THE EVOLUTION OF THE PROGRAMMING LANGUAGES COURSE

K. N. King
Department of Mathematics and Computer Science
Georgia State University
University Plaza
Atlanta, Georgia 30303

ABSTRACT
This paper examines the past, present, and future of the programming languages course as reflected by its textbooks.

1. INTRODUCTION
A course in programming languages is now a mainstay of most computer science curricula. This paper traces the changing nature of this course over the last decade, as reflected by the textbooks used to teach the course.

"Organization of Programming Languages," as the course is named in the Curriculum '78 report [2], is one of the most flexible in the computer science curriculum. Different instructors cover different languages, different topics, and use different course organizations.

There are several reasons for the many approaches to the course. The subject of programming languages is a broad one. In particular, there are numerous languages worthy of study. Furthermore, there are many ways to organize and present the basic ideas. In the words of [17], the course is characterized by a "diversity of material and [a] lack of unifying concepts." Another reason for variations in the course: disagreements over the goals of the course give rise to different organizations and topic selections.

The flexibility of the programming languages course is mirrored by the variety of textbooks that are available to teach it. Many computer science courses are dominated by one or two textbooks. There are numerous texts for the languages course, however, with no single market leader.

In recent years, a new breed of books has emerged—books that emphasize language paradigms instead of individual languages. These new books provide instructors with yet more options.

This paper compares the programming language course of today with the course as described in the Curriculum '78 report. It also examines how the course will evolve in light of the *Computing Curricula 1991* recommendations.

As a method of examining the evolution of the programming languages course, this paper compares major textbooks with respect to their organization and their coverage of languages, paradigms, and topics. The books to be discussed, all published within the last ten years, are:

- Appleby, *Programming Languages: Paradigm and Practice* [1]
- Dershem/Jipping, *Programming Languages: Structures and Models* [4]
- Friedman, *Comparative Programming Languages: Generalizing the Programming Function* [5]
- Ghezzi/Jazayeri, *Programming Language Concepts* [6]
- Horowitz, *Fundamentals of Programming Languages* [7]
- Kamin, *Programming Languages: An Interpreter-Based Approach* [8]
- MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation* [10]
- Marcotty/Ledgard, *The World of Programming Languages* [11]
- Pratt, *Programming Languages: Design and Implementation* [12]
- Schneider, *Problem Oriented Programming Languages* [13]
- Sebesta, *Concepts of Programming Languages* [14]
- Sethi, *Programming Languages: Concepts and Constructs* [15]
- Tennent, *Principles of Programming Languages* [16]
- Tucker, *Programming Languages* [18]
- Wilson/Clark, *Comparative Programming Languages* [23]

A secondary purpose of this paper is to provide guidance to instructors who teach the programming languages course. With so many texts available, it often difficult to choose the right one. The information provided in this paper will help an instructor decide what the goals of his or her course should be and which books are best suited to achieving those goals.

This paper is not a book review. It concentrates solely on issues of coverage and organization. Other matters of interest to instructors—accuracy, clarity, and quality of examples, for example—are not considered.

2. HISTORY
In the Curriculum '78 report, Organization of Programming Languages (CS 8) is one of eight core courses. In the years since this report was published, most computer science departments have adopted a course resembling CS 8.

However, the versions of CS 8 currently offered (as well as the books used) rarely match the description in [2]. For example, much of the course should be devoted to language implementation

issues, according to [2]. But many CS 8 texts downplay implementation issues or omit them entirely.

What are the reasons for the changes to CS 8? One is the need to devote more space to new languages. During the 1980s, several important languages were created and several languages of the 1970s became popular. A second reason is the need to discuss not just individual languages but entire programming paradigms. The most striking trend in the field of programming languages over the past ten years has been the rise of paradigms, of which the object-oriented paradigm is the best-known.

But there is a more fundamental reason for the diversity of the programming languages course: the fact that instructors—and authors—often disagree on the goals of the course. This disagreement is reflected in the titles of current texts. Should the course cover "principles" and "concepts"? Should it be a "comparative" study of languages? To what extent should it cover "design" and "implementation"?

3. GOALS

The goals of CS 8, according to [2], are "(a) to develop an understanding of the organization of programming languages, especially the run-time behavior of programs; (b) to introduce the formal study of programming language specification and analysis; (c) to continue the development of problem solution and programming skills introduced in the elementary level material."

The goals of textbooks for CS 8 often differ considerably from those stated in [2]. Here are the goals that have been cited by various authors; no single book promises to meet all of them:

- *Knowledge of specific languages.* Students should be introduced to languages that are in wide use.
- *Deeper understanding of language issues.* By comparing features across languages, students should arrive at a deeper understanding of the issues that drive programming language design. At the same time, students should come to better appreciate the languages they already know.
- *Ability to critique languages.* Students should learn how to evaluate the strengths and weaknesses of a language.
- *Ability to compare languages.* Students should develop the ability to choose the right language for a particular application.
- *Knowledge of features and paradigms.* Students should be exposed to a wide range of language features. According to Kamin, the course should "foster an appreciation of the range of programming languages and styles that are possible." A broad knowledge of language features will expand students' horizons (in Sebesta's words, students will experience an "increased capacity to express ideas"). This knowledge will prove helpful even when students later use a language that doesn't support a given feature, because they can often simulate it. (For example, students who are familiar with pointers or recursion could simulate those features in Fortran, say.) Students should also be exposed to language paradigms beyond the usual imperative approach. New languages often incorporate the features of older ones, so a broad knowledge of features and paradigms will help prepare students to learn new languages in later years.
- *Knowledge of language-related theory.* Students should be exposed to the theory that underlies the design of programming languages and the construction of compilers and interpreters. Such coverage allows students to see applications of subjects that might otherwise seem impractical.
- *Appreciation for the historical development of programming languages.* Students should learn why some languages have prospered while others fell into disuse, which language features have become popular and which are now regarded as obsolete, and what historical reasons underlie decisions made by language designers. Students should become aware of how our view of the role of programming languages has changed over time and how modern-day languages are designed to support the software engineering process.
- *Preparation for later courses.* According to the Curriculum '78 report, CS 8 "should provide appropriate background for advanced level courses involving formal and theoretical aspects of programming languages and/or the compilation process." The programming languages course is most often a stepping-stone to a course in compiler construction.
- *Knowledge of language implementation techniques.* Students should learn how high-level languages are implemented. This knowledge is crucial for all majors, not just those who will later take a course in compiler construction. One reason: programmers should be aware of the cost of features they use. Also, knowledge of implementation issues is often necessary to understand the reasoning behind language design decisions (restrictions on language features in particular).
- *Ability to design languages.* Although few students will ever design general-purpose programming languages, many will design language-based interfaces to other kinds of software.

Sebesta mentions an especially ambitious goal: "overall advancement of computing." In his view, certain languages (Fortran, for example) have prospered despite technical deficiencies, simply because the people who selected languages were unable to appreciate the advantages of alternative languages (Algol 60, say). If those who select programming languages are better informed, Sebesta argues, technically superior languages will win out, thus advancing the entire computer field.

4. PARADIGMS

For the past few years, Peter Wegner and others have promoted the concept of "language paradigm" as a way of classifying languages (see [20] and [21], for example). Generally accepted paradigms include the imperative or procedural paradigm (of which Pascal is an example), the concurrent or distributed paradigm (CSP), the database paradigm (SQL), the functional or applicative paradigm (Lisp), the logic programming paradigm (Prolog), and the object-oriented paradigm (Smalltalk).

Recent papers ([9] and [22] are but two examples) have discussed where language paradigms should be introduced in the curriculum. At present, the programming languages course provides most students with their first taste of alternatives to the imperative paradigm.

Table 1 shows paradigm coverage in various texts. The books are arranged by publication date, making it easy to see the increasing emphasis on covering multiple paradigms.

All but one of the books covers the imperative paradigm in detail. (Kamin, the exception, focuses on non-imperative languages, "since they present a stark aesthetic contrast with what the reader is assumed already to know—namely, Pascal.") Concurrent programming is the second most popular paradigm, despite the opinion of some instructors that concurrency is best taught in an operating systems course. (See [24] for a discussion of the merits of teaching concurrency in the programming languages course.) The functional, logic programming, and object-oriented paradigms narrowly trail the concurrent paradigm in popularity.

	Ten	*Hor*	*Pra*	*Sch*	*Tuc*	*Ghe Jaz*	*Mac*	*Mar Led*	*Wil Cla*	*Seb*	*Set*	*Der Jip*	*Kam*	*App*	*Fri*
Paradigm	*81*	*84*	*84*	*84*	*86*	*87*	*87*	*87*	*88*	*89*	*89*	*90*	*90*	*91*	*91*
Concurrent	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓✓	✓		✓✓	
Database												✓		✓	
Functional		✓	✓		✓✓	✓✓	✓		✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	
Imperative	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓	✓✓	✓✓
Logic					✓	✓✓	✓		✓	✓✓	✓✓	✓✓	✓	✓✓	✓
Object-oriented		✓					✓		✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓

✓✓ = substantial discussion of paradigm (typically a chapter or more, with two or more languages discussed); ✓ = some discussion of paradigm (typically one language). *Note:* Older texts often do not mention paradigms by name; they are assigned a ✓✓ or ✓ based on their coverage of languages that support a particular paradigm.

Table 1. Paradigm coverage

Although the object-oriented paradigm is covered in just nine books, it is likely to be a fixture of all future texts. Some authors substitute an object-*based* paradigm, allowing coverage of languages such as Ada and Modula-2 that support data abstraction but not inheritance.

5. LANGUAGES

When it comes to choosing languages to cover, the instructor of a programming languages course faces some difficult decisions.

5.1. Real Versus Simplified Languages

Should the course cover actual programming languages or languages that have been simplified—or even created—by an author? The latter approach allows the instructor to concentrate on key features while ignoring irrelevant and distracting details. Furthermore, a simplified language can be small enough to permit an author to provide a complete description and a student to gain a full understanding.

On the minus side, using artificial languages may give the student a simplistic view of languages; in particular, the student may not recognize the importance (and danger) of interaction among language features. Also, the student has nowhere to turn for additional information; studying real languages allows the student to consult other books for additional information or a different perspective.

Using actual languages has several practical advantages. The student can use widely available compilers and interpreters to test programs. Knowledge of actual languages can immediately be put to use in other courses (or in the workplace). Best of all, the student gains confidence in his or her ability to master genuine languages.

Most authors have adopted the "real language" approach, with Marcotty/Ledgard and Kamin as the chief exceptions. Marcotty/Ledgard describes a series of extremely simple "mini-languages," each designed to show off a single kind of language feature (types or procedures, say). Kamin discusses real languages, but in a "syntactically and semantically simplified form." Both books include coverage of real languages as well. Marcotty/Ledgard briefly describes how each mini-language relates to one or more actual languages. Kamin includes some discussion of each language "as it really is."

Dershem/Jipping uses artificial languages to introduce the logic and object-oriented paradigms, then switches to real languages.

5.2. Few Versus Many Languages

The number of languages to cover involves a trade-off between breadth and depth. Some authors attempt to show students the breadth of the programming languages field by exposing the student to as many languages as possible. Unfortunately, students may be overwhelmed by the sheer number of languages and features. And since few authors know many languages well, books that describe a large number of languages tend to have more errors.

The alternative is to cover fewer languages in more detail. Students then learn enough about the languages to be able to use them for nontrivial programs. Students also get a better chance to study the interaction of features—to see how the features in a language "fit together."

The number of languages covered varies considerably from book to book. At one extreme, Friedman covers only six languages. At the other, Appleby covers 16 languages and discusses many more in passing. The average is ten languages, not counting ones that are mentioned only briefly.

5.3. Which Languages?

Some authors prefer to choose languages based on their current popularity or the likelihood of their future popularity. It is interesting to note that most recent books cover C while texts over three years old generally do not, despite the fact that C has been around for about 20 years. Clearly the reason for including C now is its popularity.

Choosing popular languages has a number of practical benefits. Students are usually better motivated to study a language that they have heard of and know is in demand. Also, a good selection of books and language implementations will be available for a popular language.

Popularity is not the only reason for selecting a language, however. Some languages are chosen because they are typical of a particular paradigm (Lisp is a common representative of the functional paradigm) or implementation technique (interpretation, say). Other authors cover languages because they best illustrate a particular feature or stage in the development of programming languages. A language might even be chosen because the sheer number of features that it offers provides a bountiful source of examples (Ada falls into this category). Some languages (Algol 60) are covered because of their influence on later languages.

Table 2 shows the variety of languages covered by the books in our sample. The languages most likely to be covered in a programming languages text are Ada (because of its many features), Pascal (because of its popularity and historical significance), and Lisp and Prolog (as representatives of the functional and logic programming paradigms, respectively). The table also shows

Language	Ten 81	Hor 84	Pra 84	Sch 84	Tuc 86	Ghe Jaz 87	Mac 87	Mar Led 87	Wil Cla 88	Seb 89	Set 89	Der Jip 90	Kam 90	App 91	Fri 91
Ada		✓✓	✓✓	✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓	✓	✓✓	✓	✓	
Algol 60	✓	✓		✓			✓✓	✓	✓					✓	
Algol 68	✓	✓		✓		✓✓		✓	✓						
APL			✓✓	✓	✓✓	✓							✓✓*	✓	
Basic				✓											
BCPL														✓	
C					✓✓	✓			✓	✓	✓✓	✓		✓	✓✓
C++									✓		✓✓	✓	✓	✓	
Clu						✓							✓✓*		
Cobol			✓✓	✓	✓✓			✓						✓	✓✓
Euclid		✓													
Fortran	✓		✓✓	✓	✓✓	✓	✓✓	✓	✓	✓					
FP						✓			✓			✓			
Hope									✓						
Lisp	✓	✓✓	✓✓	✓	✓✓	✓	✓✓		✓	✓		✓	✓✓*	✓✓	
Logo									✓					✓	
ML											✓		✓		
Modula		✓													
Modula-2					✓✓	✓			✓	✓	✓✓	✓			✓✓
Object Pascal														✓	
occam									✓						
Pascal	✓✓		✓✓	✓	✓✓	✓✓	✓✓	✓	✓	✓				✓	✓✓
PL/I	✓	✓	✓✓	✓	✓✓	✓		✓✓	✓	✓					
Prolog					✓✓	✓	✓✓		✓✓	✓	✓✓	✓	✓✓*	✓✓	✓✓
Sasl													✓✓*		
Scheme											✓		✓✓*		
Setl														✓	
Simula	✓	✓				✓✓				✓				✓	
Smalltalk		✓✓				✓	✓✓			✓	✓	✓	✓✓*		✓✓
Snobol	✓		✓✓	✓	✓✓									✓	
SQL												✓		✓	
Val		✓													

✓✓ = extensive coverage (an overview of the entire language or coverage of numerous features, often occupying an entire chapter or several sections in different chapters); ✓ = some coverage (often a detailed discussion of a small number of language features). Brief summaries (a page or two) and passing references are not counted. *Presents a simplified version of the language.

Table 2. Language coverage

which languages are gaining in popularity as examples (C, C++, Modula-2, and Smalltalk) and which are fading (Algol 60, Algol 68, Fortran, PL/I, and Snobol).

6. TOPICS

According to Curriculum '78, CS 8 should cover the following topics:

- Language definition structure
- Data types and structures
- Control structures and data flow
- Run-time consideration
- Interpretative languages
- Lexical analysis and parsing

The topics actually covered in texts for CS 8 vary quite a bit. Some topics are found in most books, some in a few books, and some only rarely. The most common topics include:

- *Evaluation criteria.* Criteria for evaluating a programming language design.
- *Syntax.* Lexical issues, BNF and other formalisms for defining syntax.
- *Data objects.* Constants, variables, assignment, lifetime.
- *Data types.* Simple types, structured types, pointer types, strong typing, type equivalence, type conversion.
- *Expressions.* Operator precedence and associativity, operator overloading, short-circuit evaluation.
- *Control structures.* Selection, iteration, branching, exception handling.
- *Subprograms.* Parameter-passing, recursion, overloading, generics, coroutines.
- *Scope.* Static versus dynamic scoping, block structure, modules.

- *Abstraction.* Data abstraction, control abstraction.
- *Input/output.* Formatting, files.

Although these topics are found in most CS 8 texts, there are wide variations in the amount of space devoted to each topic. Coverage of input/output, for example, is often skimpy, probably because I/O models vary so much from language to language. Wilson/Clark is one of the few books to devote much space to I/O.

The following topics are found in some texts, but not all:

- *History.* Many books contain concise histories of individual languages or a brief overview of the history of language development over the past thirty or forty years. A few provide considerably more detail, however. Appleby, for example, includes fifteen "historical vignettes," most dealing with specific languages, but some with pioneering figures in computer science, mathematics, or philosophy. MacLennan not only provides historical information about each language, but also covers languages in chronological order, allowing the student to trace their development over time.
- *Language implementation issues.* Despite the clear emphasis of CS 8 on implementation issues, some books ignore the topic or downplay it considerably. There are clear advantages to including this information in a programming languages course. Students can better appreciate the true cost of a language feature. Also, students will be better prepared for a later course in compiler construction. But there are disadvantages as well. Discussing implementation issues leaves less time for other material. Some authors feel that high-level languages exist to insulate the programmer from machine-specific details, hence it should not be necessary to discuss how a language is implemented in order to understand the features of the language. In Kamin, on the other hand, implementation issues are an integral part of the book, which is organized around a series of interpreters for the (simplified) languages studied, with Pascal source code for each included.
- *Theory.* Many books limit their treatment of the theoretical foundations of programming languages to a brief discussion of BNF. Appleby is at the other end of the spectrum. The author makes an ambitious attempt to explain the theoretical underpinnings of programming language design, even including an entire chapter on formal languages (grammars, automata, and the complete Chomsky hierarchy). Future books will likely move in this direction (see Section 8). Other books with a theoretical bent include Kamin and Sethi, which discuss the lambda calculus, and Ghezzi/Jazayeri and Tennent, which cover formal semantics.

Table 3 shows the coverage of these three topics in the books under discussion. The table shows that coverage of history and implementation issues is more common in recent books than in older ones.

Some topics are relatively rare. Friedman covers several topics that are unusual for a programming languages book: program design (a topic more often found in software engineering texts), fourth-generation languages, and expert systems. Pratt devotes a chapter to programming environments (including batch processing, interactive environments, and embedded systems). Ghezzi/Jazayeri covers both design methodologies and programming environments.

7. ORGANIZATION

It has long been known that there are several ways to organize a programming languages course. The Curriculum '78 report itself acknowledged that different organizations were feasible: "... programming languages could be specified and analyzed one at a time in terms of their features and limitations based on their run-time environments. Alternatively, desirable specification of programming languages could be discussed and then exemplified by citing their implementations in various languages." More recently, interest in paradigms has led some authors to adopt a paradigm-based organization.

There are three basic methods of organization:

- *By language.* Languages are discussed one by one. (MacLennan calls this a "horizontal" organization.) This organization gives students a better appreciation for the "look and feel" of individual languages. It is also good for discussions of how language features interact with each other. If languages are discussed in chronological order, this organization is best for showing the historical development of languages. On the other hand, there is some redundancy involved in describing and analyzing similar features in different languages. Books that employ this approach include Kamin, MacLennan, and Tucker. MacLennan, for example, covers languages one by one, using each as a springboard for a discussion of more general issues.
- *By feature.* Features are grouped into categories (control structures or data structures, say); categories are then discussed one by one. (MacLennan calls this a "vertical" organization.) As MacLennan notes, a book organized in this fashion "is apt to degenerate into a catalog of features." Schneider and Tennent are organized in this way.
- *By paradigm.* Paradigms are introduced one by one, each illustrated by one or more languages. Appleby adopts this organization.

Friedman and Pratt hedge their bets by offering two organizations: one part of the book is organized by feature, while another part is organized by language. Pratt even suggests two different ways to teach the course (by language or by feature), recom-

	Ten	*Hor*	*Pra*	*Sch*	*Tuc*	*Ghe Jaz*	*Mac*	*Mar Led*	*Wil Cla*	*Seb*	*Set*	*Der Jip*	*Kam*	*App*	*Fri*
Topic	*81*	*84*	*84*	*84*	*86*	*87*	*87*	*87*	*88*	*89*	*89*	*90*	*90*	*91*	*91*
History		✓					✓✓		✓✓	✓✓		✓		✓✓	✓✓
Implementation			✓✓				✓✓			✓		✓	✓✓		
Theory	✓✓		✓			✓					✓		✓	✓✓	

✓✓ = extensive coverage (an entire chapter or numerous shorter discussions); ✓ = some coverage (more than a brief summary).

Table 3. Topic coverage

mending the former for undergraduates and the latter for graduate students.

A number of books (including Dershem/Jipping, Ghezzi/Jazayeri, Horowitz, Sebesta, Sethi, and Wilson/Clark) use a hybrid organization in which early chapters are organized by feature (for discussing imperative languages) and later chapters are organized by paradigm (for discussing non-imperative languages). One reason for this organization is that the non-imperative languages are often difficult to fit into the framework used for comparing imperative languages.

Marcotty/Ledgard offers a unique solution to the problem of organization: each chapter covers a single kind of language feature, which the authors illustrate using a mini-language they invented.

8. THE FUTURE

With the publication of *Computing Curricula 1991* [19], the days of CS 8 are numbered.

The 1991 curriculum specifies a number of "knowledge units"—collections of related material to be taught as a whole. It also lists "common requirements" for computer science majors (a set of knowledge units covering the areas in the Denning report [3]). The common requirements in the area of programming languages consist of the following knowledge units:

PL1. History and overview of programming languages

PL2. Virtual machines

PL3. Representation of data types

PL4. Sequence control

PL5. Data control, sharing, and type checking

PL6. Run-time storage management

PL7. Finite-state automata and regular expressions

PL8. Context-free grammars and pushdown automata

PL9. Language translation systems

PL10. Programming language semantics

PL11. Programming paradigms

PL12. Distributed and parallel programming constructs

Although computer science programs are expected to include all these units, there is no requirement that they be covered in a single course. (PL7 and PL8, for example, could be taught as part of a course on theoretical computer science.) However, it seems likely that programming languages books of the future will include all twelve knowledge units, on the principle that omitting any could eliminate their book from consideration by some instructors.

Most books already cover the material in units PL1–PL6. Although they usually contain some coverage of context-free grammars (in the guise of BNF), most books lack further coverage of PL7 and PL8. Coverage of PL9–PL12 varies considerably. Some books omit material on compilation (PL9) and semantics (PL10), while others devote considerable space to these topics. Recent books are strong on paradigms (PL11). Most books contain some discussion of concurrency (PL12), although it may need to be expanded and updated.

The net effect of the 1991 curriculum recommendations will likely be to make programming languages books more uniform (in topic coverage, at least) and more theoretical, as authors scramble to add material on automata. Books will also likely become longer, thanks to the broader coverage suggested by the new curriculum.

Today's books, of course, were written before the 1991 curriculum was finished. Still, some have already moved in that direction. Appleby, for example, cites the Denning report—the starting point for the 1991 curriculum—as a source of topics.

9. CONCLUSION

There is no single best way to teach a programming languages course, as the diversity of texts illustrates. Many factors influence the organization and topic coverage, including:

- The goals of the course.
- The instructor's preferences.
- The language implementations available.
- The relationship of the courses to others in the curriculum. A programming languages course must take into account which languages students have seen in previous courses (either to avoid duplicating discussion of these languages or to exploit this prior knowledge by comparing new languages and paradigms with those the student already knows). In many departments, a programming languages course must serve a second purpose. It may introduce a language that is used in later courses or that is thought to be important for students to know before graduation.

Because of these factors, it is unlikely that the programming languages course will ever become as uniform as other courses in the computer science curriculum. Textbooks will likely become more similar in topic selection to conform with the 1991 curriculum recommendations, but instructors will still have considerable leeway in choosing languages and organizing the course.

REFERENCES

1. Appleby, D., *Programming Languages: Paradigm and Practice*, McGraw-Hill, New York, 1991.
2. Austing, R. H., B. H. Barnes, D. T. Bonnette, G. L. Engel, and G. Stokes, Curriculum '78: recommendations for the undergraduate program in computer science—a report of the ACM Curriculum Committee on Computer Science. *Communications of the ACM* **22**, 3 (March 1979): 147–166.
3. Denning, P. J., D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, Report of the ACM task force on the core of computer science. *Communications of the ACM* **32**, 1 (January 1989): 9–23.
4. Dershem, H. L., and M. J. Jipping, *Programming Languages: Structures and Models*, Wadsworth, Belmont, Calif., 1990.
5. Friedman, L. W., *Comparative Programming Languages: Generalizing the Programming Function*, Prentice-Hall, Englewood Cliffs, N.J., 1991.
6. Ghezzi, C., and M. Jazayeri, *Programming Language Concepts*, Second Edition, Wiley, New York, 1987.
7. Horowitz, E., *Fundamentals of Programming Languages*, Second Edition, Computer Science Press, Rockville, Md., 1984.
8. Kamin, S. N., *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, Reading, Mass., 1990.
9. Luker, P. A., Never mind the language, what about the paradigm? *SIGCSE Bulletin* **21**, 1 (February 1989): 252–256.
10. MacLennan, B. J., *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt, Rinehart and Winston, New York, 1987.
11. Marcotty, M., and H. Ledgard, *The World of Programming Languages*, Springer-Verlag, New York, 1987.

12. Pratt, T. W., *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, N.J., 1984.
13. Schneider, H. J., *Problem Oriented Programming Languages*, Wiley, Chichester, England, 1984.
14. Sebesta, R. W., *Concepts of Programming Languages*, Benjamin/Cummings, Redwood City, Calif., 1989.
15. Sethi, R., *Programming Languages: Concepts and Constructs*, Addison-Wesley, Reading, Mass., 1989.
16. Tennent, R. D., *Principles of Programming Languages*, Prentice-Hall International, Englewood Cliffs, N.J., 1981.
17. Trenary, R., A project centered programming language course. *SIGCSE Bulletin* **19**, 1 (February 1987): 67–69.
18. Tucker, A. B., *Programming Languages*, Second Edition, McGraw-Hill, New York, 1986.
19. Tucker, A. B., ed., *Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force*, ACM Press, New York, 1991.
20. Wegner, P., Introduction to the special issue on programming language paradigms. *Computing Surveys* **21**, 3 (September 1989): 253–258.
21. Wegner, P., Concepts and paradigms of object-oriented programming. *OOPS Messenger* **1**, 1 (August 1990): 7–87.
22. Wells, M. B., and B. L. Kurtz, Teaching multiple programming paradigms: a proposal for a paradigm-general pseudocode. *SIGCSE Bulletin* **21**, 1 (February 1989): 246–251.
23. Wilson, L. B., and R. G. Clark, *Comparative Programming Languages*, Addison-Wesley, Wokingham, England, 1988.
24. Yeager, D. P., Teaching concurrency in the programming languages course. *SIGCSE Bulletin* **23**, 1 (March 1991): 155–161.