Criteria for the selection of a programming language for introductory courses

Kevin R. Parker* and Thomas A. Ottaway

Idaho State University
Campus Box 8020
Pocatello, ID 83209, USA
E-mail: parkerkr@isu.edu
E-mail: ottathom@isu.edu
*Corresponding author

Joseph T. Chao

Bowling Green State University
Bowling Green, OH 43403, USA
E-mail: jchao@cs.bgsu.edu

Abstract: Historically, the selection of a programming language for an introductory programming course has been a process consisting of faculty evaluation, discussion, and consensus. As the number of faculty, students, and language options grows, this process is likely to become increasingly unwieldy. In addition, the process lacks structure and cannot be easily replicated. The selection process will, in all likelihood, be repeated every two to three years. Providing a structured approach to the selection of a programming language would yield a more thorough evaluation of the options available and a more easily justified selection. Developing and documenting an exhaustive set of selection criteria, and an approach for the application of these criteria, will allow the process of language selection to be more easily repeated in the future. This paper presents a comprehensive set of criteria that should be considered when selecting a programming language for a teaching environment, and proposes several approaches for the application of these criteria.

Keywords: programming language selection; introduction to programming; teaching programming; programming language evaluation.

Reference to this paper should be made as follows: Parker, K.R., Ottaway, T.A. and Chao, J.T. (2006) 'Criteria for the selection of a programming language for introductory courses', *Int. J. Knowledge and Learning*, Vol. 2, Nos. 1/2, pp.119–139.

Biographical notes: Dr. Kevin R. Parker is Professor of Computer Information Systems at Idaho State University, having previously held an academic appointment at Saint Louis University. He has taught both Computer Science and Information Systems courses over the course of his 14 years in academia. Dr. Parker's research interests include e-commerce marketing, competitive intelligence, knowledge management, the semantic web, and extreme programming. He has published in such journals as *Journal of Information Technology Education*, *Journal of Information Systems Education*, *International Journal of Internet and Enterprise Management*, and *Journal of Issues in Informing Science and Information Technology*. Dr. Parker's teaching

interests include web development technologies, programming languages, data structures, and database management systems. Dr. Parker holds a BA in Computer Science from the University of Texas at Austin (1982), an MS in Computer Science from Texas Tech University (1991), and a PhD in Management Information Systems from Texas Tech University (1995).

Dr. Thomas A. Ottaway is Associate Professor in the Computer Information Systems Department at Idaho State University, having previously held academic appointments at Kansas State University and the University of Montana. He holds a BS in Computer Science from Wichita State University (1991), an MS in Management Information Systems from Texas Tech University (1993), and a PhD in Production/Operations Management from Texas Tech University (1995). He has published in such journals as *Decision Sciences*, *International Journal of Production Research*, *Journal of Economics and Finance*, and *Journal of Financial and Economic Practice*. Dr. Ottaway's teaching interests include data and telecommunications networks as well as computer programming.

Dr. Joseph T. Chao is Assistant Professor of Computer Science at Bowling Green State University. He has taught courses in all aspects of the software development life cycle including programming, systems analysis and design, database systems, usability engineering, software engineering, and agile software development. Dr. Chao has seven years of industry experience in software development, including three years as Director of Software Development. His research focus is on software engineering with special interests in programming languages, object-oriented analysis and design, software modelling, agile methodologies, and extreme programming. Dr. Chao holds an MS in Operations Research from Case Western Reserve University and a PhD in Industrial and Systems Engineering from the Ohio State University.

1 Introduction

The programming language used in teaching an introductory programming course can have a great impact on how the course is taught as well as its effectiveness. Language selection has long been a difficult and unstructured process. Fewer issues in the world of software development are as strategic, political, and contentious as the choice of programming language (Jensen, 2004). Over the years various languages have been viewed as contenders for the primary programming languages in IS and CS programmes, as seen in Wile's (2002) timeline of the succession of programming languages (and language types) throughout their evolution. Programming language selection is usually no more systematic than a series of faculty meetings focusing on informal language assessment, debate, and eventual consensus. With the diversity of high-level programming languages available, selecting the 'right' language for a computing-focused curriculum or course can be a perplexing process (Tharp, 1982). For many reasons, such as the manner in which students approach problems or scarce computing resources, the selection of a programming language has ramifications throughout the curriculum (Tharp, 1982). As the number of faculty, students, and language options grows, language selection is likely to become increasingly complex. Furthermore, the selection process currently lacks structure and thus cannot be easily replicated. Because the selection process is often repeated every two to three years, developing and documenting a set of selection criteria, and a process for the application of these criteria, will result in a process that will be more easily repeated in the future. A structured approach to the selection of a programming language will enable a more thorough evaluation of the available options and a more easily validated selection.

Several factors must be considered when selecting a programming language, and whilst different curricula place greater emphasis on different factors, all must be considered. We seek to develop a comprehensive set of selection criteria and a process for the application of these criteria to evaluate programming languages to be used in programming classes. The selection criteria must take into account the programming features of each language under consideration, the appropriateness of each of these features for beginning (and perhaps advanced) programming courses, the present and future industry acceptance of each language, the availability and quality of textbooks, the costs associated with adopting each language, the infrastructure and support implications of each language, and the impact of the decision on the tactical and strategic direction of the department and curriculum.

2 Literature review

The literature review begins by discussing the lack of experimental comparisons of the usability of programming languages, accompanied by a brief mention of those that do exist. It then points out that many factors must be considered when selecting a language, even before formal criteria can be considered. Note that whilst some of these factors will be included in the criterion, they may also be used as a pre-evaluation tool to narrow the field of choices.

McIver (2002) indicates that although there are very strong feelings on the subject of first programming languages, few evaluative studies of languages or development environments for introductory programming classes exist. Further, there have been few, if any, empirical tests comparing different languages (McIver, 2000). Wilson (1997) agrees that whilst debates over the relative merits of various programming languages are quite common, there have been relatively few experimental comparisons of the usability of different programming languages.

Wilson (1997) was able to find only one such study, an experiment by Szafron and Schaeffer (1996) to measure the usability of parallel programming systems. Another study, conducted by Murtagh and Hamilton (1998), performed a one-on-one comparison of the impact of two languages on the success of students in an introductory programming class, but their approach requires an instructor proficient in both languages and is not easily extended to more than two languages at a time. Various arguments have been made for the use of a particular language or particular paradigm in introductory classes (Kölling *et al.*, 1995; Conway, 1993). However, McIver (2002) points out that whilst anecdotal evidence from introductory programming courses is widely available (Reed, 2001; Allen *et al.*, 1996; Popyack and Herrmann, 1993), and individual language features have been studied from a cognitive point of view (Soloway *et al.*, 1989; Sime *et al.*, 1973), the determination of which language should be used for teaching introductory programming remains a contentious issue.

One explanation offered for the scarcity of studies is that the variability among programmers would render any study meaningless (Wilson, 1997). Another is that the differences between domains are too great to perform a meaningful study. Wilson (1997) disagrees with the premise that such studies are not possible but does not offer any solutions. Comparing languages is a difficult task, especially when the languages do not share the same paradigm (McIver, 2000). Establishing proper criteria for the comparison is difficult not only because a question of what to measure exists, but also because the criteria may favour one language over another (McIver, 2000). McIver (2002) points out that in educational settings the demands of various courses and curricula make it problematical, if not impossible, to compare different languages. Further, different courses generally have sufficiently different objectives to make language comparisons virtually meaningless (McIver, 2002).

2.1 Factors to be considered

As noted earlier, many factors must be considered prior to beginning the language selection process. Although these factors eventually may be included in the evaluation criteria, they may also be used to narrow the field of choices. For example, the selection will almost certainly be guided by the methodology or paradigm being taught and the number of languages to be used throughout the series of programming courses. Another decision is whether the department wishes to use a real language or a customised teaching language. Finally, the cost of changing programming languages must be considered.

2.2 Methodology or paradigm

The selection of the programming paradigm, which determines what should be taught, must precede the selection of the first language, which influences how to teach it (Esendal, 1994). The paradigm defines the framework within which the students are taught. Programming paradigms are differentiated by the concepts that they emphasise (Watt, 2000). Imperative programming emphasises procedures operating on (unencapsulated) variables; object-oriented programming emphasises methods operating on (encapsulated) objects; functional programming emphasises functions operating on immutable values; logic programming emphasises predicates operating on immutable values; concurrent programming emphasises processes exchanging messages (Watt, 2000). Bowman (1994) contends that the central consideration when evaluating introductory programming courses should not be the languages but rather the selection of a theory or methodology of programming to teach. Wolz (1997,p.12) states that the "focus on language is misguided because it forces an emphasis on the mechanics of expressing key ideas rather than focusing on the key ideas themselves". Additional considerations include whether language constructs should be learned separately from or concurrently with programme design, how to determine the appropriate balance between programming in the large and programming in the small, and whether a single language should be taught throughout the programming course sequence as opposed to multiple languages (Wallace *et al.*, 1997).

The related issue of a single language versus multiple languages is another critical decision. The number of languages to cover in a course or curriculum involves a trade-off between breadth and depth (King, 1992). Some educators prefer to cover fewer languages in more detail, assuming students will learn enough about the languages to be able to use them for nontrivial programmes, whilst other educators attempt to show students the breadth of the programming field by exposing the student to as many languages as possible (King, 1992).

2.3 Real or customised

Regardless of the paradigm, a choice must be made between professional-grade and customised languages. A professional-grade language is one used in industry and is taught in its entirety (with the exception of very advanced features), via a commercially available environment (Esendal, 1994). A professional-grade language like Java or C++ provides students with experience with a real-life environment. Any problems encountered are those that will be encountered in real life, preparing students better for their professional lives (Esendal, 1994).

A customised language is one developed especially for teaching purposes or one that is a subset of a real language (Esendal, 1994). Customised languages such as Haskell and Eiffel separate learning programming from learning the details of a particular language (de Raadt *et al.*, 2003), minimising the technical problems that may distract from the learning process. Once students learn the fundamentals, they can apply their programming knowledge to any language (Esendal, 1994).

2.4 Cost of changing

Some educators naively think that there is little cost in changing programming languages because there are no serious consequences if a decision turns out badly. Lee and Phillips (2002) point out, however, that considerable overhead occurs in adopting a particular language, including preparation of lecture materials, developing projects and student exercises, evaluating and learning development environments and installing the chosen language, requesting and evaluating textbooks, and training personnel. Such overheads indicate that care should be taken when choosing a language because that choice is likely to impact the educator for several years.

3 Selection criteria

The programming language selection criteria appear in Table 1. These criteria were derived through a thorough review of the literature, and each will be justified by a brief review of the supporting literature.

Each of the criteria in Table 1 has been used in one or more previous studies that evaluate programming languages.

Table 1 Language selection criteria

Criterion
Reasonable financial cost for setting up the teaching environment
Availability of student/academic version
Availability of textbooks
Language's stage in life cycle
System requirements of student/academic/full version
Operating system dependence
Open source (versus proprietary)
Academic acceptance
Industry acceptance
Marketability (regional and national) of graduates
Easy to use development environment
Ease of learning basic concepts
Support for target application domain (such as scientific or business)
Full-featured language (versus scripting)
Support for teaching approach (function first, object first or object early)
Object-oriented support
Good debugging facilities
Support of web development
Support for safe programming
Advanced features for subsequent programming courses
Availability of support
Training required for instructors and support staff
Anticipated programming experience level for incoming students

3.1 Reasonable financial cost

This criterion refers to the price to acquire the programming language or the development environment. This may involve individual packages or a site license for a network version. There may be an academic discount for educational institutions; there may be an alliance in which the university or department can enroll; or there may even be a free, downloadable version. Cost is often included in the overhead involved in adopting a particular language. Lawlis (1997) enumerates the various types of costs that may be associated with the choice of development products, including purchase price, training costs, installation costs, cost of additional hardware, and cost of additional people needed. Martin (2003), de Raadt *et al.* (2002; 2003), Tolmach (1999), and Stephenson (2000) all include costs in their lists of factors affecting programming language choice. Lee and Phillips (2002) expand the definition of costs to include all of the overheads associated with adopting a particular language, including preparation of lecture materials, developing projects and student assignments, installing and learning the development environment, requesting and evaluating textbooks, and training personnel.

3.2 Availability of student/academic version

Although the availability of a student version or academic version is not cited in many studies, it has always been a concern when we have been involved in programming language selection. If a student version is unavailable and the department uses a network-based version, then students may be forced to work on their assignments in campus labs, restricted by hours of operation, availability of transportation, *etc.* If the academic version is stripped down, then the benefit to the students may not be as great, but this factor should at least be evaluated.

De Raadt *et al.* (2003) consider availability and/or cost to students. A recent request for information posted on the ISWorld ListServ (Tomblin, 2002) regarding programming language choice for introductory classes elicited multiple responses that alluded to the availability of a student version or academic discounts. One posting noted that an inexpensive student version was a major benefit of a particular language. Another posting noted that if software was network-based then students would not have access to it on their home computers. That posting went on to point out that some programming languages are, however, disseminated as a student version that allows students to install and use it on their home computers.

3.3 Availability of textbooks

The availability of textbooks may be affected by many additional factors. The life-cycle stage of the language impacts the availability of textbooks, particularly when the language is relatively new. When Visual Basic.Net was first released, few quality textbooks were available, but as the language has matured more have been published. The academic acceptance of a language also plays a large role in the availability of textbooks because a larger potential market exists for a text that deals with a more widely used language. Finally, textbook availability may also be affected by the teaching approach used. For example, functions-first, objects-first, or objects-early are all approaches used to teach object-oriented languages, but few recent texts present the material from a functions-first perspective. Availability of reference books should also be taken into account (Lee and Stroud, 1996).

Watt (2000) includes textbook availability as one of the resource issues to be considered, whilst McIver and Conway (1996) include the assessment of textbook quality as one of the usual considerations when evaluating potential languages. An easy-to-find, appropriate text is one language selection criteria listed by de Raadt *et al.* (2002; 2003).

3.4 Stage in life cycle

As noted above, a programming language's stage in the programming language life cycle should also be considered. In addition to affecting textbook availability, it may also impact the widespread use of a language in both industry and academia. Universities may prefer a language that is still in its earlier stages, rather than one like COBOL or FORTRAN, which are in their declining years.

Not to be confused with the programme development life cycle, the programming language life cycle, as described by Sharp (2002), is based on the natural principles of growth, maturation, and decay. The processes of natural advantage and evolution operate in the world of programming languages in the same way that they operate in the biological domain, but in the case of languages the main forces are efficiency of expression versus profitable adoption. New languages compete with older languages. If a language is general purpose, functional, expressive, and marketed, then it will most likely be adopted. Once adopted, the application market drives language evolution.

Sharp's programming language life cycle begins with the conception stage, when a language is conceived to fill deficiencies not met by existing languages. That is followed by the adoption phase, as programmers perceive that the language will improve their efficiency. As the language stabilises and exhibits fewer defects, there is general acceptance. The maturation stage is characterised by a greater demand for functionality than for efficiency, which leads to the inefficiency stage, characterised by increased functionality but also a market fragmented by disparate vendor implementation of standards. The decline continues in the deprecation stage as development becomes more costly and inefficiencies lead to consideration of alternatives. Finally, in the decay stage, newer languages that lack some features but are faster and more efficient appear on the scene and displace their precursors.

3.5 System requirements of student/academic/full version

The system requirements of the programming language often play a role in the selection process. This includes hardware as well as operating system requirements. The amount of hard disk space needed to install the software, the operating system required, and the memory to run the software all factor into the decision. For example, some of the .Net framework requirements include Windows NT 4.0 or later, a Pentium II 450 MHz processor (minimum), 3.5 GB of available disk space, and minimum 160 MB RAM for Windows XP Professional. Many student and lab machines may be unable to meet the minimum requirements of some languages. Several studies list hardware or software requirements among the criteria to be considered. Tharp (1982), McIver and Conway (1996), Stephenson (2000), Prechelt (2000), and de Raadt *et al.* (2003) all include some variation of this factor.

3.6 Operating system dependence

This criterion refers to the dependence of a language on a particular operating system platform, often referred to as portability. For example, any of the languages supported by the .Net framework, including Visual Basic, C++, C#, *etc.*, depend on the Windows operating system. Other languages, such as Java, are platform independent, and development environments for Java can be found for a variety of operating systems. This may be of concern to faculty members who may or may not prefer to be bound to a specific operating system. Howatt (1995), Paprzycki (2002), and Riehle (2003) all consider portability from an operating system standpoint, whilst Wile (2002) refers to it as the computing platforms on which a language runs.

3.7 Proprietary/open source

This refers to the entity that controls the evolution of a language and its associated development environment. For example, Microsoft is responsible for additions, deletions, or modifications in any of the languages supported by the .Net framework. Sun is responsible for the ongoing evolution of the Java language. On the opposite end of the spectrum, PHP is an open-source language and can be easily implemented by any member of the open-source community. Both Stephenson (2000) and Riehle (2003) include open source in their criteria.

3.8 Academic acceptance

Academic acceptance refers to the popularity of a language at other academic institutions. This can be assessed by current use or projected use at other institutions. For example, the growth in popularity of object-oriented programming and the recent decision by the College Board to move the Advanced Placement Computer Science programme to Java have led to an increasing number of universities, colleges, and secondary schools adopting Java as the programming language for their introductory programming courses (Roberts, 2004). As a result, in December 2003, the ACM Education Board, in conjunction with the ACM Special Interest Group on Computer Science Education, initiated the ACM Java Task Force to study and report on how to teach the language more effectively. A 2002 survey reported that most Information Systems programmes (62%) still teach Visual Basic (VB6 or VB.Net) as their first language, whilst 51% of the respondents require Java either as a first or second language (Watson, 2002).

3.9 Industry acceptance

Industry acceptance refers to the market penetration (Riehle, 2003) of a particular language in industry, *i.e.*, the use of a language in business and industry. Often referred to as industrial relevance, this can be assessed based on current and projected usage, as well as the number of current and projected positions. Stephenson (2000) claims that this factor has the greatest influence in language selection, as indicated by 23.5% of schools that participated in his study. Lee and Stroud (1996) include a language's usefulness and acceptability to the real world. A 2001 census of all Australian universities revealed that perceived industry demand was the major factor in the choice of an introductory language (de Raadt *et al.*, 2003). McIver and Conway (1996) refer to language popularity as a typical consideration when evaluating potential teaching languages. King (1992) agrees that many language decisions are made on the basis of current popularity or the likelihood of future popularity, but Howland (1997) objects that too many languages are chosen simply because of their current popularity rather than for sound pedagogical reasons.

A 2004 study in *InfoWorld* indicated that Java is the language most used by professional developers (64%) followed by Visual Basic at 56% and C++ at 55% (McAllister, 2004). A 2005 survey commissioned by Tiobe Software (2005) provided conflicting results, reporting that C is the most used language (18.630%), followed by Java (16.981%) and Perl (10.197%). Industry acceptance affects a related criterion – marketability.

3.10 Marketability (regional and national)

Marketability refers to the employability of graduates. This may refer to regional or national/international marketability, based on the placement of a programme's graduates. Language selection is often driven by demand in the workplace, *i.e.*, what employers want. Not only are marketable skills important in future employability, but students are more enthusiastic when studying a language they feel will increase their employability (de Raadt *et al.*, 2003).

This criterion is stressed in several studies. The census of introductory programming courses conducted by de Raadt *et al.* (2003) emphasises the importance of employability. In fact, the most commonly listed factor in language selection (by 56% of the participants) was the desire to teach a language that provides graduates with marketable skills. Watt (2000) discusses the need for transferable skills that will be useful in whatever career the student chooses to pursue. Emigh (2001) agrees that the primary concern in language evaluation must be the demand in the workplace and argues that when deciding on a new language one must take into account employers' expectations of graduates. Further, graduates' marketability can be improved by exposing them to several languages (de Raadt *et al.*, 2003). They cite, for example, that a progression from C to C++ to Java will qualify a graduate for more advertised positions than exposure to any single language in isolation.

Emigh (2001) points out a caveat that must be considered when assessing both industry acceptance and marketability. Generally, four to five years pass between a student's beginning a programme of study and attaining a position using his or her programming skills. Even if a curriculum teaches a newer programming language, there is no guarantee that employers will still be looking for that language when the student enters the workforce.

3.11 Development environment

The development environment is a programmer's virtual workbench, and can improve or inhibit productivity (Jensen, 2004). Development environments range from simple text editors and command-line compilers to fully interactive and Integrated Development Environments (IDE) (McIver, 2002). Kölling *et al.* (1995) point out that the IDE should be easy to use so that the students can concentrate on learning programming concepts rather than the environment itself. Murtagh and Hamilton (1998) agree that the development environment must be one that novice students are able to figure out. Eisenstadt and Lewis (1992) take it a step farther, citing evidence that well-designed programming environments assist students in learning to programme.

Although McIver (2002) indicates that there have been few evaluative studies of development environments for introductory classes, the IDE is cited in several studies as a factor in language selection. Lee and Phillips (2002) state that evaluating various IDEs is one of the overheads in language evaluation. Howland (1997) lists the need for an IDE as one of his criteria. Jensen (2004) points out that as professional-grade languages become more sophisticated and complex, IDEs become more intimidating. Educators who experienced the transition from Visual Basic 6 to Visual Basic.Net are familiar with increasing IDE complexity.

3.12 Ease of learning fundamental concepts

The learning curve associated with each language or IDE differs greatly between languages. The most obvious recent example is the steep increase in the learning curve from Visual Basic 6 to Visual Basic.Net. Basic concepts include the sequence, selection, and iteration control structures, as well as arrays, procedures, basic input/output, and file manipulation. Kölling *et al.* (1995) note that a language should support clean, simple, and well-defined concepts; the language should have an easily readable, consistent syntax.

The ease of learning fundamental programming concepts is cited in several studies as a crucial factor in language selection. The first programming language must serve as a vehicle for exploring fundamental programming design concepts of sequence, selection, iteration, variables, and arrays (Wolz, 1997; Bishop-Clark and Donohue, 1999), so educators must select a language that supports and clearly expresses those fundamental concepts (Watt, 2000). Wang (2001) and Traxler (1994) also discuss the importance of fundamental programming concepts.

In addition to ease of learning, the language must be characterised by concise syntax and straightforward semantics (Conway, 1993). Clarity of syntax and semantics are cited by McIver and Conway (1996), Tolmach (1999), Paprzycki (2002), and Fergusson (2003) as essential considerations in the selection of a language. Milbrandt's (1993) criteria include both simplicity of syntax and ease of use. Ease of use and learning is also cited by both Howatt (1995) and Cunningham (2004). In a survey conducted by Stephenson (2000), 13.1% of the respondents indicated ease of use as a primary factor in language selection. The Ad Hoc AP CS Committee (2000) cites a need for a language and a programming environment that are reasonably simple, noting that students can be easily sidetracked by awkward syntax, complex language semantics, or expansive programming environments. The committee report suggests that a simple and clear context can encourage students to develop high-level thinking skills.

3.13 Supports target application domain

This criterion is included to assess how well a language supports programming for specific applications (Howatt, 1995). Sometimes referred to as 'problem domain', this is not to be confused with Domain Specific Languages. Examples of application domain include FORTRAN's support for scientific programming, COBOL's support for business data processing, and RPG's support for report generation.

Howatt (1995) identifies application domain criteria as one of the items in his language evaluation criteria, and defines it as the designers' intended use of the language. He points out that most discussions of language evaluation either treat this category in very general terms or fail to address it entirely. Wharton (1995) prefers the term 'problem domain' in his comparison of FORTRAN and C with respect to scientific programming. AlGhamdi and Urban (1993) propose 12 areas of analysis for comparing and assessing programming languages, including philosophy of the design, defined as the intent of the designers when designing the language. Shaw *et al.* (1981) assess the software engineering characteristics of multiple languages by rating the core of each language that captures the essential properties of a language and the intent of language designers about its intended use. Whatever term is used, this factor can play a pivotal role in language selection.

3.14 Scripting or full-featured language

Programming educators must also choose between full-featured and less complex languages. Prechelt (2000) refers to them as conventional programming languages and scripting languages. Some programming instructors prefer scripting languages like

Python because they offer sufficient richness to cover most of the requirements of an introductory course whilst reducing the complexity of the development environment and avoiding many other implementation issues. Warren (2001,p.214) states that JavaScript has "sufficient richness to cover most of what is required in an elementary course and it is a real language with immediate application for the student". He goes on to point out that JavaScript also uses a simple editing environment, reduces language complexity, and improves consistency. Full-featured languages, however, offer a more complete set of language features that an instructor may want to incorporate. Some of these issues may be related to the next criterion, the choice between teaching basic concepts and teaching a specific language.

Full-featured or conventional programming languages like C++ and Java are compiled rather than interpreted, and they require typed variable declarations (Prechelt, 2000). On the other hand, scripting languages such as Perl, Python, Rexx, and Tcl are generally interpreted rather than compiled, at least during the programme development phase, and they typically do not require variable declarations (Prechelt, 2000).

MVI Solutions (2004) points out that whilst scripting languages are growing in popularity among professional programmers, serious questions arise about performance, software reuse, and integration with components written in other languages. Some debate exists as to whether scripting languages support the learning of core programming concepts (Stephenson, 2000). However, Prechelt's (2000) comparison of the two language types reports that designing and writing programmes in scripting languages takes less than half as much time as conventional languages, and the resulting programmes are generally half as long. Prechelt (2000) also observes no clear differences in programme reliability among the language groups but notes that the typical script programme consumes about twice as much memory as a C or C++ programme, although Java programmes consume three or four times as much memory as C or C++ programmes.

Although web development features are the focus of a later criterion, a discussion of scripting languages must mention client-side or server-side scripting. As future professionals, students in the computing disciplines have to learn to develop internet applications; skills that can be acquired only when students understand client-server computing through learning HTML, JavaScript, and Java applets (Wang, 2001). As the internet continues to grow, demand for skills in scripting and markup languages also increases (de Raadt *et al.*, 2002; 2003). The many job postings requiring scripting/markup languages indicate that the modern programming degree should include these languages and the web-related concepts surrounding them (de Raadt *et al.*, 2002).

3.15 Teaching approach support

As noted above, this criterion refers to the assessment of how well a language supports the teaching approach preferred by the faculty, *i.e.*, whether the intent is to teach programming concepts, with the language simply being a vehicle through which those concepts are reinforced, or whether the intent is to teach the features of a particular language, such as the many user interface controls offered by Visual Basic. King (1992) asserts that the disagreement about whether programming courses should focus on basic programming concepts as opposed to a particular language is one of the fundamental reasons for the diversity of programming courses and textbooks.

Many studies echo the importance of concepts over language. Tomblin (2002) states that the focus of programming classes should not be so much the language as it should be the concepts and good programming practices that need to be taught. Kölling *et al.* (1995) note that the aim of their programming courses is to educate students in such a way that they understand the underlying concepts, and are thus able to write good programmes in any language. Other studies discuss the use of Java from this perspective. Warren (2001) reflects that whilst many features offered by Java are necessary for industrial-strength programmes, they are simply 'gratuitous complexity' in teaching programming concepts. Collins (2002) discusses Java's suitability to demonstrate and convey the concepts that are important across a programming curriculum.

Other programmes take the alternate approach and stress teaching individual language features over programming concepts. Lee and Phillips (2002) assert that most students regard training in a specific language more useful than an education in programming concepts. Further, Emigh (2001) notes that many universities are reacting to student demand to be taught the technicalities of a particular language rather than programming concepts.

3.16 Object-oriented support

This criterion assesses how well a programming language supports basic Object-Oriented (OO) concepts like abstraction, polymorphism, inheritance, and encapsulation. The evaluator should consider that some languages that profess to be object-oriented are merely object-based, meaning that they fail to provide support for all of the OO features listed above. Again, if an OO language is selected, the instructor must choose between an objects-first approach and an objects-early approach.

The Ad Hoc AP CS Committee (2000) emphasises object-orientation as a primary need, noting that courses should place an emphasis on higher-level abstraction, OO design, encapsulation, inheritance, and polymorphism. Kölling *et al.* (1995) are quite detailed in their specification of requirements that a first year teaching language must meet, stating that the language should exhibit 'pure' object-orientation, supporting the basic concepts of Object-Oriented Programming (OOP) such as information hiding, inheritance, type parameterisation and dynamic dispatch in a consistent and easily understood manner. Stephenson and West (1998) note that many instructors argue that the first language must be a true OO programming language with support for inheritance, polymorphism, encapsulation, *etc.* Riehle's (2003) study focuses on object-oriented languages and explicitly lists OOP as a criterion. Several others also incorporate support for the OO paradigm as an essential factor in language selection, including Watt (2000), de Raadt *et al.* (2002; 2003), Stephenson (2000), Howland (1997), Murtagh and Hamilton (1998), Wang (2001), Paprzycki (2002), and Voegele (2004).

3.17 Debugging facilities

Whilst this criterion is considered part of the IDE, when assessing a programming language one should evaluate the debugging facilities that accompany the language, *i.e.*, the existence of adequate diagnostic aids (Tharp, 1982). The Ad Hoc AP CS Committee (2000) report states that programming environments should contain extensive tools for tracing and debugging. The error diagnostics should be clear and meaningful (McIver and Conway, 1996), and the language should be robust as well as graceful in failure (Conway, 1993).

3.18 Support of web development

One criterion that may not be applicable to every curriculum but critical in others is the level of web development support that a particular language provides. This is not limited to scripting languages, as discussed in a previous section, because some languages like ASP.Net provide a high level of support for web development but are at the same time considered full featured. Many programmes consider it essential that today's students have the skills to develop web-based applications. Wang (2001) notes that future IS and CS professionals must acquire the skills to develop computer applications in the internet environment. Fortunately, web application development has evolved to the use of high-level development tools that focus on the integration of varying components (Courte, 2004). Further, there are many IDEs and programmes designed to generate useful web pages.

Martin (2003) compares the features of two web development languages, PHP and Perl. Haga and Fustos (2002) list several skills needed by a web developer, including writing server-side and client-side application programmes, web page design and development, visual design of web pages with graphical and multimedia applications, database integration, site configuration, management, maintenance, and writing and editing for the web. The skills most requested in their analysis of position announcements are server-side scripting (70%), programming languages (68%), database (55%), markup languages (51%), and client-side scripting (46%).

3.19 Coding safety

This criterion can be used to assess two important factors. The first considers whether the language offers features like strong type checking and array bounds checking, whilst avoiding features like variants and pointers in unsafe mode. Kölling *et al.* (1995,p.174) note that a language "should avoid concepts that are likely to result in erroneous programmes. In particular it should have a safe, statically checked (as far as possible) type system, no explicit pointers and no undetectable uninitialised variables". They further state that a language should provide support for correctness assurance, such as assertions and pre and post conditions. The Ad Hoc AP CS Committee (2000) report cites a need for safety in a language and environment. Several other studies, including Watt (2000), Milbrandt (1993), Murtagh and Hamilton (1998), Fergusson (2003), and Riehle (2003) also mention this as a criterion in language selection.

The second factor, which is closely related to the first, is the inclusion of security-related features like Java's sandbox, which is intended to limit the memory addresses that a Java programme can access. Another example, cited by Voegele (2004), points out that Java applets are considered untrusted, and thus are limited in the actions they can perform when executed from a user's browser. The Princeton Secure Internet Programming Team (1998) details the requirements for a secure system, including type safety, modular programming, and security policies.

3.20 Advanced features for subsequent programming courses

If multiple programming courses are included in a computing curriculum, whether or not a programming language offers adequate advanced features to support an advanced programming course may be an issue. Whilst some programmes prefer to teach multiple languages in their curriculum, other programmes prefer to introduce basic programming language features in an introductory course and defer advanced features of the language, like multithreading, to a subsequent course. Lee and Stroud (1996) include whether a language provides a basis for subsequent courses that require use of a programming language, *e.g.*, compiler construction, operating systems, and concurrent programming. In either case, introductory programming courses cannot be considered in isolation from the remainder of the courses required in a curriculum (de Raadt *et al.*, 2003).

3.21 Availability of support

This criterion refers to the availability of support staff, including computer lab staff and/or network administrators, to support the teaching and administration of a language. Both Tolmach (1999) and Watt (2000) list availability of trained personnel among their selection criteria. The evaluators must consider the likelihood that their language questions will be answered (Cunningham, 2004), and should also take into account the availability of support through forums or listservs on the internet, as well as vendor support (Tharp, 1982). The evaluators may want to consider the availability of other resources like teachers' guides, example programmes, student workbooks, and programming assignments. Both Collins (2002) and Stephenson (2000) include the availability of instructional and technology resources.

3.22 Qualified instructors and staff

This refers to the training required for instructors and support staff as well as the time needed to learn a language or its IDE. It also takes into account the availability of qualified instructors to teach a particular language. Emigh (2001,p.2) points out that adopting a new language requires a willingness on the part of the university to invest in the education of its educators because instructors:

> "Must continuously enrich their qualifications, implement new training methods and techniques supplemented with practical methods and techniques supplemented with practical experience; while teaching a new language that is as new to them as it is to their class."

Lee and Phillips (2002) include training as one of the overheads associated with language adoption, as do Tolmach (1999) and Wile (2002). Stephenson (2000) reflects on a need for continuous training to aid instructors in keeping up with constantly changing technology. Lawlis (1997) says that there is no substitute for good education and training, so the availability of language-related education and/or training courses must be a part of language selection.

3.23 Anticipated programming experience level for incoming students

The final criterion is the anticipated programming experience level for incoming students. This is important because students' previous experience and training skews their understanding of new programming paradigms and languages (Traxler, 1994). If students coming into a programme consistently exhibit the same traits such as previous exposure to Java, then it may play a role in language selection. Figures released by the College Board in 2004 indicate that 20% of college-bound students had taken at least one computer programming course in high school (CollegeBoard.com, 2004). Lee and Phillips (2002) and Kölling *et al.* (1995) discuss the increase in student experience levels. The proportion of students with programming experience has increased significantly over time although not sufficiently to require prior experience as a prerequisite for the course (Lee and Stroud, 1996). Still, if a programme consistently sees students with uniform programming experience, it may be able to adjust its requirements and its programming language selection accordingly.

4 Practical aspects

Although language selection is highly subjective, a thorough list of criteria makes an objective selection process possible. However, the process may still vary drastically due to the differences in culture, strategy, or even politics at each institution. The following steps provide a systematic approach in a general selection process:

1. Compile a list of criteria

   The criteria proposed by this study can be adapted to fit the needs of most departments or programmes.

2. Weight each of the criteria

   Ask each evaluator to weight, specific to the department's needs, the value of importance for each criterion. For example, the weight may range from zero (do not care) to ten (extremely important). If there are multiple evaluators, either a consensus can be reached or the weights assigned by each evaluator can be averaged.

3. Determine a list of candidate languages

   The list should comprise of languages nominated by the faculty rather than a complete list of available languages. Having sub-lists may be desirable so that a subset of candidate languages can be compared at one time to narrow down the choices, and comparing several similar languages may also be desirable.

4. Rate the language

   Each candidate language should be assigned a rating for each criterion. The score may range from zero (extremely low) to ten (extremely high). Again, with more than one evaluator, a consensus should be reached or average scores could be calculated.

5. Calculate weighted score

   For each candidate language, a weighted score can be calculated by adding together the language score multiplied by the weight assigned to each criterion. The language with the highest weighted score is the optimal choice based on the evaluators' assessments.

The process is fairly mechanical and can be easily adapted to fit the needs of individual departments. It may be better to begin the selection process with a brief introduction to the procedure. A language selection committee may be formed to evaluate and adapt the selection criteria and to assign a weight to each criterion for the department.

Not every faculty member in the department may have expertise in or even familiarity with all the languages to be evaluated. One solution is to provide evaluators with programme code samples for each language to be evaluated, or different groups of evaluators could assess each subset of language candidates. Another alternative would be to require each evaluator to state his or her confidence level on each language evaluated.

5 Summary and conclusion

In this paper we have presented the relevant and extant literature on the selection of a programming language for use in an introductory programming course. We have developed a comprehensive set of criteria to be used in evaluating a programming language. Finally, we have proposed a process by which these criteria may be used to compare programming languages to facilitate the selection of a language.

By constructing an exhaustive set of evaluation criteria and using these criteria in a structured manner we have set forth a means by which much of the subjectivity in the selection process may be removed. In addition, the approach presented is extensible. As new programming paradigms and languages are introduced and old ones fall out of favour, the criteria and associated process may easily be revised. The objectivity and extensibility of this approach yield the repeatability sought in the original research objectives.

In practice, the choice of a programming language for an introductory course is often a compromise. Economic, political, and pedagogical factors may all be relevant to the decision-making process. Whilst the importance of each of these factors may depend on the specific aims and priorities of the institution, educator, or course, educators must be certain that the factors in the above criteria are not neglected or sacrificed to more highly visible concerns (McIver and Conway, 1996).

Future research will include refinement of the selection criteria, formalisation of the selection process, and application of the process in a variety of settings. A large number of selection criteria have been presented so as to develop the most comprehensive selection instrument possible. Some of these criteria are likely more relevant than others. Future research will capture the relative importance of each of these criteria across different languages, decision makers, and decision-making environments. Formalisation of the selection mechanism will draw on methods used in multicriteria decision-making. Finally, the refined selection criteria and process will be applied and evaluated in a variety of academic settings and the results will be evaluated for use in further refining both the process and the instrument.

References

Ad Hoc AP CS Committee (2000) *Round 2: Potential Principles Governing Language Selection for CS1-CS2*, http://www.cs.grinnell.edu/~walker/sigcse-ap/99-00-principles.html

AlGhamdi, J. and Urban, J. (1993) 'Comparing and assessing programming languages: basis for a qualitative methodology', *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*, Indianapolis, Indiana, pp.222–229.

Allen, R.K., Grant, D.D. and Smith, R. (1996) 'Using Ada as the first programming language: a retrospective', *Proceedings of Software Engineering: Education and Practice*, Dunedin, New Zealand, pp.234–241.

Bishop-Clark, C. and Donohue, C. (1999) 'Comparing changes in attitude in three different introductory computing courses', *Journal of Educational Technology Systems*, Vol. 27, No. 4, pp.305–317.

Bowman, H. (1994) 'A perspective on language wars', *2nd All-Ireland Conference on the Teaching of Computing*, Dublin, Ireland, http://www.ulst.ac.uk/cticomp/papers/ bowman.html

CollegeBoard.com (2004) '2004 college-bound seniors: a profile of SAT test takers', *College Entrance Examination Board*, Online, http://www.collegeboard.com/prod_downloads/about/news_info/cbsenior/yr2004/2004_CBSNR_total_group.pdf

Collins, D. (2002) 'Java second. The suitability of Java as a first programming language', *The Sixth Java and the Internet in the Computing Curriculum Conference Proceedings*, London, UK, http://www.ics.ltsn.ac.uk/pub/jicc6/collins.doc

Conway, D. (1993) 'Criteria and considerations in the selection of a first programming language', *Technical Report 93/192*, Department of Computer Science, Monash University.

Courte, J.E. (2004) 'The difficulties of incorporating web development into a university curriculum', *Paper Presented at the New Society of the WWW Conference*, Rose-Hulman Institute of Technology, Terre Haute, Indiana, USA, 30 September–2 October, http://www10.cs.rose-hulman.edu/Papers/Courte.pdf

Cunningham, W. (2004) 'Language comparison framework', *Portland Pattern Repository*, 29 November, http://c2.com/cgi/wiki?LanguageComparisonFramework

Eisenstadt, M. and Lewis, M.W. (1992) 'Errors in an interactive programming environment: causes and cures', in M. Eisenstadt, M.T. Keane, and T. Rajan (Eds.) *Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence*, Hillsdale, NJ: Lawrence Erlbaum Associates, http://citeseer.ist.psu.edu/cache/papers/cs/3586/http:zSzzSzkmi.open.ac. ukzSzmarczSzpaperszSzBookCh5.pdf/errors-in-an-interactive.pdf

Emigh, K.L. (2001) 'The impact of new programming languages on university curriculum', *Proceedings of ISECON 2001*, Vol. 18, Cincinnati, Ohio, http://isedj.org/isecon/2001/16c/ISECON.2001.Emigh.pdf

Esendal, H.T. (1994) 'The selection of first programming language', *2nd All-Ireland Conference on the Teaching of Computing*, Dublin, Ireland, http://www.ulst.ac.uk/cticomp/esendal.html

Fergusson, K. (2003) 'Anti compiler: an educational tool for first year programming', *Honors Thesis*, Department of Computer Science, Monash University, http://www.nifwlseirff.net/honours/litreview/final/lit-review.pdf

Haga, W.A. and Fustos, J.T. (2002) 'Weaving a web development curriculum', *Proceedings of Informing Science and IT Education Conference*, Cork, Ireland, pp.629–643, http://proceedings.informingscience.org/IS2002Proceedings/papers/ Haga155Weavi.pdf

Howatt, J.W. (1995) 'A project-based approach to programming language evaluation', *ACM SIGPLAN Notices*, Vol. 30, No. 7, pp.37–40, http://academic.luther.edu/~howaja01/v/lang.pdf

Howland, J.E. (1997) 'It's all in the language: yet another look at the choice of programming language for teaching computer science', *Journal of Computing in Small Colleges*, Vol. 12, No. 4, pp.58–74, http://www.cs.trinity.edu/~jhowland/ccsc97/ccsc97/

Jensen, C. (2004) 'Choosing a language for .NET development', *Borland Developer Network*, http://bdn.borland.com/article/0,1410,31849,00.html

King, K.N. (1992) 'The evolution of the programming languages course', *Proceedings of the Twenty-Third SIGCSE Technical Symposium on Computer Science Education*, Kansas City, Missouri, pp.213–219.

Kölling, M., Koch, B. and Rosenberg, J. (1995) 'Requirements for a first year object oriented teaching language', *Proceedings of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, Nashville, Tennessee, pp.173–177.

Lawlis, P.K. (1997) *Guidelines for Choosing a Computer Language: Support for the Visionary Organization*, 2nd edition, Ada Information Clearinghouse, http://archive.adaic.com/docs /reports/lawlis/content.htm

Lee, P.A. and Phillips, C. (2002) 'An assessment of c++ as an introductory teaching language', *Technical Report CS-TR: 777*, Department of Computing Science, University of Newcastle, http://www.cs.ncl.ac.uk/research/pubs/trs/papers/777.pdf

Lee, P.A. and Stroud, R.J. (1996) 'C++ as an introductory programming language', in M. Woodman (Ed.) *Programming Language Choice: Practice and Experience*, London: International Thomson Computer Press, pp. 63–82, http://www.cs.ncl.ac.uk/old/publications /books/apprentice/InstructorsManual/C++_Choice.html

Martin, M. (2003) 'PHP: putting perl in a jam? The battle for web programming', *Computer Bits*, Vol. 13, No. 4, http://www.mindbridge.com/news/PerlinaJam.htm

McAllister, N. (2004) 'What do developers want?', *InfoWorld*, http://www.infoworld.com /article/04/09/24/39FErrdev_1.html

McIver, L. (2000) 'The effect of programming language on error rates of novice programmers', *Proceedings of the Twelfth Annual Meeting of the Psychology of Programming Interest Group*, Corigliano Calabro, Italy, pp.181–192, http://www.ppig.org/papers/12th-mciver.pdf

McIver, L. (2002) 'Evaluating languages and environments for novice programmers', *Proceedings of the Fourteenth Annual Meeting of the Psychology of Programming Interest Group*, London, UK, pp.100–110, http://www.ppig.org/papers/14th-mciver.pdf

McIver, L. and Conway, D.M. (1996) 'Seven deadly sins of introductory programming language design', *Proceedings of Software Engineering: Education and Practice (SE:E&P'96)*, Dunedin, NZ, pp.309–316.

Milbrandt, G. (1993) 'Using problem solving to teach a programming language in computer studies', *Journal of Computer Science Education*, Vol. 8, No. 2, pp.14–19.

Murtagh, J.L. and Hamilton, J.A. (1998) 'A comparison of Ada and Pascal in an introductory computer science course', *Proceedings of the Annual ACM SIGAda International Conference on Ada*, Washington, DC, pp.75–80.

MVI Solutions (2004) *Computer Programming*, http://www.mediavue.net/programming /programLanguage/computer_programming.html

Paprzycki, M. (2002) *Programming Languages*, http://www.cs.okstate.edu/~marcin/mp/teach /spring01/408/csc408.html

Popyack, J.L. and Herrmann, N. (1993) 'Mail merge as a first programming language', *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education*, Indianapolis, Indiana, pp.136–140.

Prechelt, L. (2000) 'An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl', *IEEE Computer*, Vol. 33, No. 10, http://page.mi.fu-berlin.de/~prechelt/Biblio/jccpprt _computer2000.pdf

Princeton Secure Internet Programming Team (1998) *Programming Language Support for Security*, http://www.cs.princeton.edu/sip/language/

de Raadt, M., Watson, R. and Toleman, M. (2002) 'Language trends in introductory programming courses', *Proceedings of Informing Science and IT Education Conference*, Cork, Ireland, pp. 329–337, http://www.proceedings.informingscience.org/IS2002Proceedings /papers/deRaa136Langu.pdf de Raadt, M., Watson, R. and Toleman, M. (2003) 'Introductory programming languages at Australian universities at the beginning of the twenty first century', *Journal of Research and Practice in Information Technology*, Vol. 35, No. 3, pp.163–167.

Reed, D. (2001) 'Rethinking CS0 with JavaScript', *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, Charlotte, North Carolina, pp.100–104.

Riehle, R. (2003) 'SEPR and programming language selection', *CrossTalk – The Journal of Defense Software Engineering*, Vol. 16, No. 2, pp. 13–17, http://www.stsc.hill.af.mil/crosstalk/2003/02/Riehle.html

Roberts, E. (2004) 'Resources to support the use of java in introductory computer science', *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, pp.233–234.

Sharp, R. (2002) 'Programming language lifecycles – where's Java at?', *Software Reality*, http://www.softwarereality.com/programming/language_lifecycles.jsp

Shaw, M., Almes, G.T., Newcomer, J.M., Reid, B.K. and Wulf, W.A. (1981) 'A comparison of programming languages for software engineering', *Software – Practice and Experience*, Vol. 11, No. 1, pp.1–52.

Sime, M.E., Green, T.R.G. and Guest, D.J. (1973) 'Psychological evaluation of two conditional constructions used in computer languages', *International Journal of Man-Machine Studies*, Vol. 5, No. 1, pp.105–113.

Soloway, E., Bonar, J. and Ehrlich, K. (1989) 'Cognitive strategies and looping constructs: an empirical study', in E. Soloway and J.C. Spohrer (Eds.) *Studying the Novice Programmer*, Hillsdale, NJ: Lawrence Erlbaum Associates, pp.853–860.

Stephenson, C. (2000) 'A report on high school computer science education in five US states', http://www.holtsoft.com/chris/HSSurveyArt.pdf

Stephenson, C. and West, T. (1998) 'Language choice and key concepts in introductory computer science courses', *Journal of Research on Computing in Education*, Vol. 31, No. 1, pp.89–95.

Szafron, D. and Schaeffer, J. (1996) 'An experiment to measure the usability of parallel programming systems', *Concurrency: Practice and Experience*, Vol. 8, No. 2, pp.147–166.

Tharp, A.L. (1982) 'Selecting the "right" programming language', *Proceedings of the 13th SIGCSE Technical Symposium on Computer Science Education*, Indianapolis, Indiana, pp.151–155.

Tiobe Software (2005) *TIOBE Programming Community Index for April 2005*, http://www.tiobe.com/tpci.htm

Tolmach, A.P. (1999) *Evaluating Programming Languages*, http://www.cs.pdx.edu/~apt/cs301_1999/lecture3/

Tomblin, S. (2002) 'Summary of MIS and programming responses', *ISWorld Listserv*, 19 April, http://www.marshall.edu/ctc/ctc2/MIS_programming_responses.htm

Traxler, J. (1994) 'Teaching programming languages and paradigms', *2nd All-Ireland Conference on the Teaching of Computing*, Dublin, Ireland, http://www.ulst.ac.uk/cticomp/traxler.html

Voegele, J. (2004) *Programming Language Comparison*, http://www.jvoegele.com/software/langcomp.html

Wallace, C., Martin, P. and Lang, B. (1997) 'Not whether Java but how Java', *CTI Computing Monitor*, No. 8, http://www.scism.sbu.ac.uk/jfl/conference/uwe.html

Wang, S. (2001) 'An approach to teaching multiple computer languages', *Journal of Information Systems Education*, Vol. 12, No. 4, pp.201–211.

Warren, P. (2001) 'Teaching programming using scripting languages', *The Journal of Computing in Small Colleges*, Vol. 17, No. 2, pp.205–216.

Watson, H. (2002) 'Programming languages summary', *ISWorld Listserv*, 28 December http://www.isworld.org/isworldarchives/Teachingmessagedisplay.asp?message=237

Watt, D.A. (2000) 'Programming languages – trends in education', *Proceedings of Simposio Brasileiro de Linguagens de Programacao*, Recife, Brazil, http://www.dcs.gla.ac.uk/~daw/publications/PLTE.ps

Wharton, L. (1995) *Should C Replace FORTRAN as the Language of Scientific Programming?*, http://www.cs.colorado.edu/~zorn/cs5535/Fall-1995/projects/wharton.ps

Wile, D.S. (2002) 'Programming languages', in J.J. Marciniak (Ed.) *Encyclopedia of Software Engineering*, 2nd edition, Hoboken, NJ: John Wiley and Sons, pp.1010–1023.

Wilson, G.W. (1997) 'Tools, languages, and interacting with machines', *Dr. Dobb's Journal*, July, http://www.ercb.com/ddj/1997/ddj.9707.html

Wolz, U. (1997) 'Language considerations in a goal-centered approach to CS I and II: Java, C, or what?', *Journal of Computing in Small Colleges*, Vol. 12, No. 5, pp.12–20.