Language History – A Tale of Two Countries

Bill Davey and Kevin R. Parker

[1] *School of Business Information Technology, RMIT University, Melbourne, Australia <billd@rmit.edu.au>*
[2] *Idaho State University, Pocatello, Idaho USA < parkerkr@isu.edu>*

Abstract. This paper looks at the relationships between industry computer languages and those taught in universities. By considering the differences between two of the first countries to embrace programmable computers (USA and Australia) we find patterns that seem culturally independent. History shows a set of recurring problems for academics in choosing languages. This study shows that academics should be informed by history when making those decisions.

1. Introduction

The dawn of the history of programmable computers can be traced back to Eckert and Mauchly's departure from the ENIAC project to start the Eckert-Mauchly Computer Corporation. The fourth programmable computer in the world (SCIRAC) was developed in Australia. This computer, manufactured by the government science organization (CSIRO), still exists as a complete unit at the Museum Victoria in Melbourne. The computer was used into the 1960's as a working machine at the University of Melbourne. Australia's early entry into computing makes the comparison with the United States interesting.

These early computers needed programmers, that is, people with the expertise to convert a problem into a mathematical representation directly executable by the computer. The first programmers were mostly mathematicians or engineers who programmed in machine code of some form. Many of them used hardwiring to achieve their ends. Few if any of these early programmers had any formal education in machine language programming.

vInitial computer-related offerings by universities were courses in engineering or physics. Academia's landscape changed with remarkable speed as the twin paths of computer science and information systems degrees were quickly established in most developed countries. In Australia the University of Sydney introduced a course called "The Theory of Computation, Computing Practices and Theory of Programming" in 1947 (Tatnall and Davey, 2004). The speed of the introduction of specialized degrees paralleled the introduction of hardware and software in industry. The speed of change meant that both industry and university sectors were required to make very difficult choices between hardware and software alternatives.

At that time the connection between the computing industry and academia was tight. Industry progressed due to innovation from university academics, and many industry leaders moved to teaching and research positions. In Australia the 1960's saw Gerry Maynard move from the Post Office to set up a degree at the Caulfield Technical College, Donald Overheu move from the Weapons Research Establishment to the University of Queensland, and Westy Williams leave the public service to start a program at Bendigo Technical College (Tatnall and Davey, 2004). Much of improvement of software and the emergence of languages occurred in university research departments, performed by passionate academics focused on discipline-based research rather than on industry needs.

2. The History of Language Selection

The first languages were the individual machine languages developed to control specific central processing units. UNIVAC's C-10 language was one of the first to use mnemonic instructions, like "a" for add and "b" for bring, and by the late 1950's universities had discovered (and in most cases created) higher-level languages.

> I remember a lecture given by a colleague, Peter Sefton, in the late 1950s on a new language called Fortran, which he said he thought might relieve some of the tedium of programming in machine language (Smillie, 2004).

The development of FORTRAN began in 1954 and culminated in the first release in 1957. ALGOL, released in 1958 with a major update in 1960, introduced recursion, indirect addressing, and character manipulation, among other features. Many universities adopted it as the language for use in their computer programming courses because it was a precise and useful way for capturing algorithms (Keet, 2004). COBOL was developed in 1959 and was widely used for a number of decades in business applications.

As early as 1960 there were 73 languages in existence (Sammet, 1972). By 1967 there were 117, and by 1971 there were 164 (Sammet, 1972). Of these only ten, ALGOL, APT, COBOL, Comit, FORTRAN, IPL, LISP, MAD, MADCAP, and NELIAC, appeared on all the lists. Sammet identifies the period from 1960 to 1970

as the decade in which the programming language field maturated. During this time, the battle over the use of higher-level languages was clearly won in the sense that machine coding had become the exception rather than the rule.

By 1972, most universities in Australia and the USA had established computer science or information systems (the latter often called "data processing") degree programs. Almost all computer science degree programs offered ALGOL, FORTRAN or LISP, while most data processing programs offered COBOL. In Britain BASIC was also important. During the late 60s, schools experimented with various languages like PL/I.

This situation changed most markedly with the introduction of Pascal. The first version of Pascal was released in 1970. Wirth began teaching Pascal to engineering and physics students in 1971 (Wirth, 1993), but the real impact of Pascal had to await the release of the P-machine in 1973 (Wirth, 1993).

The mid-1970s brought about another important change – the introduction of the microcomputer. These machines came with BASIC and revolutionised the teaching of computer courses in high schools. Most secondary schools immediately started using BASIC, but this trend did not impact university programs. With the introduction of Pascal in the 1970s, most universities adopted Pascal for their introductory programming course. Some authors attribute this to two pragmatic factors: the invention of the personal computer, and the availability of Pascal compilers (Levy, 1995).

Pascal compilers were always far slower than the languages used in industry, but the speed was well within the limits needed in a teaching environment. At this time academics used arguments to justify the divergence from using industrially common languages. For example, Merritt (1980) wrote

Since Pascal is a widely available and well-designed language, it was suggested that Pascal provided a unique language environment in which these features that support high quality program construction can be learned. However, it is reasonable to expect that reliable software will be a priority, that the connections between good programs and language features will continue to be made, and that language features will develop along the lines presented here. Information Systems graduates will be in systems development and management roles.

Of course Wirth himself admits that logic is not really the most important underlying cause of human decisions, "But it was probably my stubborn persistence rather than any reasoned argument that kept Pascal in use"(Wirth, 1993).

The use of Pascal in academia was eventually superseded by languages used in industry, beginning with C and C++, and eventually shifting to Java and C#. As recently as 1996 a survey of CSAB accredited programs showed the most popular first language was still Pascal at 36% of the responding institutions, followed by C++ at 32% and C at 17% (McCauley and Manaris, 1998).

The selection of programming languages in university curricula in the US and Australia is almost identical, with some interesting differences. The current distribution in Australia is shown in Table 1.

Table 1: Languages taught (de Raadt et al. 2003b)

Language	Number of courses	Weighted by students
Java	23	43.9%
VB	14	18.9%
C++	8	15.2%
Haskell	3	8.8%
C	4	5.5%
Eiffel	2	3.3%
Delphi	1	2.0%
Ada	1	1.7%
jBase	1	0.8%

This is a close approximation to the statistics in US universities. One historical difference between the countries involved Ada. When the US Department of Defense mandated Ada for their applications the language experienced a surge in US colleges, which declined after 1997 when the mandate was removed.

3. How Universities Choose Languages

The problems that must be faced in designing an introductory course are many and varied. These range from those of interdepartmental politics in the case of service courses to logistical challenges if substantial numbers of students must be accommodated (Solntseff, 1978). A cursory glance through back issues of computer-related journals such as the ACM Special Interest Group on Computer Science Education (SIGCSE) Bulletin makes it apparent that discussions about the introductory programming language course and the language appropriate for that course have been numerous and on-going (Smolarski, 2003). The selection of a programming language for instructional purposes is often a tedious chore because there is no well-established approach for performing the evaluation. The informal process may involve faculty discussion, with champions touting the advantages of their preferred language, and an eventual consensus, or at least surrender. As the number of faculty, students, and language options grows, this process becomes increasingly unwieldy. As it stands, the process currently lacks structure and replicability.

A list of the factors that affected the choice of a programming language for an introductory course at one US university is ably discussed in Smith and Rickman (1976). According to Solntseff (1978), there “appears to be no other discussion in the literature of comparable thoroughness”.

A current study carefully examines a first programming language for IT students (Gee et al., 2005). A more recent study examines over 60 papers relevant to language selection in academia (Parker et al., 2006). Over the years languages have been invented to solve problems. Other languages have been invented to make teaching algorithms easier. This has lead to two sometimes conflicting lines of arguments by academics about which languages they should use in university courses: choose a language that is commonly used or is expected to be commonly used in industry, or choose a language that best supports concept development in students. Thus, there have been two distinct arguments for language selection that have been extant throughout the history of languages: pragmatic versus pedagogical.

The pragmatic approach recommends choosing a language that will help students get a job after graduating. The pragmatic approach is impacted by a language's industry acceptance as well as the marketability of individuals proficient in its use.

3.1 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) point out that real-world acceptability is a factor that once had little weight, as indicated by the earlier use of ALGOL and Pascal, but that attitude does seem to be changing. They note that for their students being able to have an industrially accepted language on their résumé is a significant consideration for them. 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., 2003a). King (1992) agrees that many language decisions are made on the basis of current popularity or the likelihood of future popularity; he notes that choosing popular languages has a number of practical benefits, including increased student motivation to study a language that they have heard of and know is in demand, as well as a good selection of books and language implementations that will be available for a popular language.

3.2 Marketability

Marketability refers to the employability of graduates. This may include regional or national/international marketability, based on the placement of a program'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., 2003a).

Language marketability is stressed in several studies. The census of introductory programming courses conducted by de Raadt et al. (2003a) emphasizes 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., 2003a). 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. An example of this argument is:

> There is perhaps an implication here for the choice of platform and language. Extrinsically motivated students aspiring to a lucrative career will demand to be taught those tools that are currently in vogue in the industry. Universities may have to accept that pedagogical issues in the choice of platform and language must be secondary to marketing concerns. (Jenkins, 2001)

3.3 Pedagogy

Smolarski (2003), McIver and Conway (1996), and Howland (1997) question whether changes in the curriculum and programming courses should be as driven by industry as they often seem to be. They argue that decisions about the language used in an introductory course are made based on what language would be most useful for a student in finding a job, rather than on how well it underscores fundamental skills that prepare the student for subsequent courses and help to make any software being developed by the student well-written and error-free (Smolarski, 2003).

3.3.1 Avoiding the complexities of industrial environments

These arguments also call attention to the possibility that the purposes of teaching problem solving and introducing a professional grade language into the first course conflict because students end up focusing on difficulties associated with that language and its environment (Johnson, 1995; Jenkins, 2002; Gee et al., 2005; Allison et al, 2002; Kelleher and Pausch, 2005). "A language that requires significant notational overhead to solve even trivial problems forces the language

rather than the techniques of problem-solving to become the object of study" (Zelle, 1999).

3.3.2 Clear problem-solving principles

A teaching language should have attributes that help teach fundamentals of all programming tasks. This is the argument used by Wirth (1993), Kölling et al. (1995), and all the other inventors of languages designed for classroom use, and is exemplified by proponents of the various "pure" teaching languages. The argument quickly becomes one that urges use of a language not common in industry. An example of such an argument is:

> A new teaching language is required to meet the needs for teaching …. This language does not have to be a real world production language and thus can avoid the compromises in conceptual cleanness for efficiency that cause many of the problems with existing languages. (Kölling et al., 1995)

3.4 The winner?

The relevant importance ascribed to both the pragmatic and practical approaches is illustrated by a recent survey of academics, shown in Table 2. The primary reason for language selection reported by the survey is marketability, cited by 56.1% of the respondents, followed by pedagogical benefits, cited by 33.3% of the academics.

Table 2: Reasons for choosing language (de Raadt et al. 2003b)

Used in industry / Marketable 56.1%
Pedagogical benefits of language 33.3%
Structure of degree/dept politics 26.3%
OO language 26.3%
GUI interface 10.5%
Availability/Cost to students 8.8%
Easy to find appropriate texts 3.5%
OS/Machine limitations of dept 1.8%

The task of anticipating industry needs is complex. Emigh (2001) points out that four to five years pass between when a student begins a program of study and when he or she attains a position requiring 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 work force. Further, some trends are difficult to understand. Currently in Australia there seems to be a demand for multi-skilled programmers (de Raadt et al. 2003a). The average

advertisement required 1.84 languages. 48% of jobs required more than one language. C++ appeared as a requirement in around 30% of advertisements, as did Java. Visual Basic was next with 21%, followed by C with 17%.(de Raadt et al. 2003b). The Gottleibsen reports (Gottliebsen 1999; Gottliebsen 2001) on job advertisements in Australia for a sample of years shows 128 languages advertised in 1999, 3822 positions for C++, 2555 for Visual Basic, 1052 for Java, and 4678 for COBOL. By 2001 there were 206 languages in demand by industry, with 4359 positions for C++, 2680 for Java, 3369 for Visual Basic, and 1087 for COBOL.

An interesting omission from most programming language selection approaches is the ability to produce output using the language. Experiments such as that conducted by Zeigler (1995) could be used to help decide the issue. The same 60 programmers developed code in both Ada and C, the same work environment was used, as were the same debugging tools, same editors, same testing tools, and the same design methodology. Most of these programmers had masters degrees in computer science, and the more experienced programmers tended to work more in C. When first hired, 75% of the programmers knew C, while only 25% knew Ada. Despite the bias in C's favor, the experiment showed that the cost of coding in Ada is about half the cost of coding in C, because code written in Ada contained 70% less bugs discovered before product delivery and 90% less bugs discovered after product delivery (Zeigler, 1995). Note that this approach is limited by the shear quantity of programming languages available, well into the thousands today. A one-to-one comparison of all possible candidates cannot possibly be preformed.

Student perceptions also play a part in this debate. There exist several languages designed for teaching (Pascal, LOGO), but any department using one of these today would be an object of ridicule (Jenkins, 2002). It is true that programming languages designed for teaching purposes are not used to any extent by industry. Therefore student perception is that these languages are of little practical worth and further assume that, in general, they lack the advanced facilities of other languages (Gee et al., 2005). If that argument were to be carried to absurdity then the overwhelming choice would be COBOL, which now has an installed base of "more than 200 billion lines of code, and 5 billion lines of COBOL are written every year" (Langley, 2004).

Parker et al. (2006) propose a set of criteria for the selection of a programming language in an academic setting. Their work is based on papers by researchers in both Australia and the United States. Each of the criteria has been used in one or more previous studies that evaluate programming languages. This extended set of selection criteria points to a more formal and mature approach to language selection. As our current period moves into history, we may be able to see the early years of the twenty first century as a time of fundamental change in language choice.

4. Trends in Language Selection

The debate over programming language selection has been ongoing since the introduction of programming classes in university curriculums. A sampling of papers published over time provides some insights into the trends observed during given time periods.

Dijkstra (1972, p. 864) stated that

> "…the tools we are trying to use and the language or notation we are using to express or record our thoughts are the major factors determining what we can think or express at all! The analysis of the influence that programming languages have on the thinking habits of their users … give[s] us a new collection of yardsticks for comparing the relative merits of various programming languages."

Sime (1973) noted a need for an empirical approach to evaluate programming languages for unskilled users rather than experienced users, a trend that he observed in language evaluation papers prior to his work. Yohe (1974) pointed out that the development of problem-oriented languages began in the late 1950s, and they now offered an alternative to assembly language, although that was still the most basic tool available to most programmers. The availability of so many languages, however, presented a new problem in the selection of a language best suited for a particular task. Friedman and Koffman (1976) stressed the need for structured programming as a replacement to the older versions of FORTRAN, noting that "teaching disciplined programming at an elementary level is a nearly impossible task in the absence of a suitable implementation language" (p.1). Smith and Rickman (1976) were also seeking a replacement for FORTRAN, developing a well-designed set of criteria, including pedagogical factors, resource constraints, and political issues through which they "graded" ALGOL W, APL, Assembler, Basic, COBOL, EULER, Structured FORTRAN, LISP, Pascal, PL/I, and SNOBOL. In 1977, Furugori and Jalics reported that the results of their survey indicated that over half of the respondents still used FORTRAN in their introductory courses, while PL/I was used in a quarter of the schools. Finally, in 1978, Schneider indicated a trend toward the use of Pascal in classes. He pointed out that Pascal was the language that best met two critical and apparently opposing criteria – richness and simplicity. Pascal was rich in those constructs needed for introducing fundamental concepts in computer programming, but simple enough to be presented and grasped in a one-semester course.

The 1980s were marked by an increase in the number of available languages, which led to increased uncertainty about which to choose for the introductory programming course. Various paradigms were also introduced during this period. Boom and Jong (1980) performed a critical comparison of multiple programming language implementations available on the CDC Cyber 73, including Algol 60, FORTRAN, Pascal, and Algol 68. Tharp (1982) also pointed out the variety of

languages available, including FORTRAN, COBOL, Jovial, Ada, Algol, Pascal, Pl/I, and Spitbol. He discussed several recent comparisons of programming languages on the basis of their support of good software engineering practices, availability of control structures, the programmer time required for developing a representative non-numeric algorithm, and the machine resources expended in compiling and executing it. Soloway, Bonar, and Erlich (1983) discussed recent research into finding a better match between a language and an individual's natural skills and abilities. Their study explored the relationship between the preferred cognitive strategies of individuals and programming language constructs. Luker (1989) discussed the alternatives to Pascal, noting that many instructors at that time were choosing between Ada and MODULA-2. He then examined the paradigms available, including functional programming, procedural programming, object-oriented programming, and concurrent programming.

King (1992) looked at the evolution of the programming course from the *Computing Curricula 1978* to the *Computing Curricula 1991* recommendations. He noted that the 1980s saw the creation of several important languages while at the same time several languages of the 1970s became popular. He also discussed the increasing popularity of various programming paradigms during the 1980s, including the imperative or procedural paradigm, the concurrent or distributed paradigm, the database paradigm, the functional or applicative paradigm, the logic-programming paradigm, and the object-oriented paradigm. He continued by proposing a set of criteria for the selection of programming languages. Howatt (1995) also proposed an evaluation method for programming languages. His criteria included the broad categories of language design and implementation, human factors, software engineering, and application domain. He went on to provide an evaluation approach. Howland (1997) also presented an extensive list of criteria that the author felt were important in choosing a language for introductory computer science instruction, but concluded that the selection of a programming language should be made primarily on the basis of how well key programming concepts may be expressed in the language.

By the turn of the century, the object-oriented paradigm was becoming more prominent, as was the importance of security. The Ad Hoc AP CS Committee (2000) noted that in their study of language selection for CS1 and CS2 classes three main principles emerged: emphasis on object-orientation, need for safety in the language and environment, and a desire for simplicity. Wile (2002) stated that programming language choice is subject to many pressures, both technical and social. He organized the pressures into three competing needs: (1) those of the problem domain for which languages are used for problem solving; (2) the conceptual and computing models that underlie the designs of the languages themselves, independent of their particular problem domains; and (3) the social and physical context of use of the languages. He also observed a trend:

> It is clear that the role of general-purpose languages had shifted by the end of the millennium. The days of writing an entire application

> "from scratch" in a single language to build a stand-alone system that accomplishes a task are over. Modern software engineering process uses general-purpose languages as the integrating medium for extensive functionality offered by database packages, web-based services, GUIs, and myriad other COTS and customized products that interface via an application program interface (API). At the same time, "contextual concerns" for security, privacy, robustness, safety, etc., universally dominate applications across the board (p. 1027).

Roberts (2004a) observed another trend, that the growth in the popularity of the object-oriented paradigm and the decision by the College Board to move the Advanced Placement Computer Science program to Java led an increasing number of universities to adopt Java as the programming language for their introductory course. He further pointed out in (2004b) that there were two additional challenges in which dramatic increases had a negative impact on pedagogy: (1) the number of programming details that students must master has grown, and (2) the languages, libraries, and tools on which introductory courses depend are changing more rapidly than they have in the past. Finally, Gee Wills, and Cooke (2005) pointed out another trend that is becoming increasingly evident (and controversial). They mentioned several studies that the recommended the use of scripting languages to teach programming concepts because they provide "not only a proper programming environment but also an instant link into the formation of active web pages".

5. Conclusion

While there have been various differences throughout the years between Australia and the United States in the teaching of programming languages, there is a pattern that seems culturally independent. Across the two countries there has been, and still exists, a fundamental disagreement between taking a pragmatic or pedagogical approach. The pragmatic approach recommends choosing a language that will help students get a job after graduating. The pedagogical approach insists that the language used in introductory programming classes should be designed for teaching programming concepts and problem solving and should minimize complexities so that more time can be spent on developing design skills. There has been no consensus on which approach is optimal, but the ultimate lesson is that neither approach is sufficient by itself. There are additional critical factors that must be considered when selecting a programming language. Recent studies have examined a variety of factors that must be taken into account, and while pragmatic and pedagogical concerns are still near the forefront, they must be tempered by an awareness that other factors impact the selection process. The bottom line is that academics must carefully assess the best interests of the students, weigh all variables

in the language selection process such as those listed by Parker et al., (2006), and choose a language accordingly. As Johnson (1995) points out, "the greatest danger to our university system is the lemming-like rush to do the same thing, to be one with the crowd, to be part of the current fashion industry of computing."

References

[1] Allison, I., Ortin, P., and Powell, H. (2002). "A virtual learning environment for introductory programming." Proceedings of the 3rd Annual conference of the Learning and Teaching Support Network Centre for Information and Computer Sciences, Loughborough, UK.

[2] de Raadt, M., Watson, R., and Toleman, M. (2003a). "Introductory programming languages at Australian universities at the beginning of the twenty first century." Journal of Research and Practice in Information Technology 35(3): 163-167.

[3] de Raadt, M. d., R. Watson, et al. (2003b). "Language tug-Of-war: Industry demand and academic choice." Australasian Computing Education Conference (ACE2003), Adelaide, Australia., Australian Computer Society, Inc.

[4] Emigh, K L. (2001). "The impact of new programming languages on university curriculum." Proceedings of ISECON 2001, Cincinnati, Ohio, 18, 1146-1151. Retrieved July 10, 2005 from http://isedj.org/isecon/2001/16c/ ISECON.2001.Emigh.pdf

[5] Gee, Q. H., Wills, G. and Cooke, E. (2005). "A first programming language for IT students." Proceedings of the 6th Annual Conference of the Learning and Teaching Support Network Centre for Information and Computer Sciences, York, UK.

[6] Gottliebsen, C. (1999). Computer market results 1999. C. Gottliebsen. Bayswater, GIMA

[7] Gottliebsen, C. (2001). Icon index trend report 2001. Icon index Trend Report. B. Youston. Bayswater.

[8] 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, 12(4): 58-74, http://www.cs.trinity.edu/~jhowland/ ccsc97/ccsc97/

[9] Jenkins, T. (2001). "The motivation of students of programming." ACM SIGCSE Bulletin , Proceedings of the 6th annual conference on Innovation and technology in computer science education ITiCSE '01 33(3).

[10] Jenkins, T. (2002). "On the difficulty of learning to program." Proceedings of the 3rd annual conference of the Learning and Teaching Support Network Centre for Information and Computing Science, Loughborough, UK.

[11] Johnson, L.F. (1995). "C in the first course considered harmful." Communications of the ACM 38 (5): 99-101.

[12] Keet, E. E. (2004). "A personal recollection of software's early days (1960–1979): Part 1." IEEE Annals of the History of Computing (October–December).

[13] Kelleher, C. and Pausch, R. (2005). "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers." ACM Computing Surveys 37(2): 83–137.

[14] 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.

[15] Kölling, M., B. Koch, et al. (1995). "Requirements for a first year object-oriented teaching language." ACM SIGCSE Bulletin , Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education 27(1).

[16] Langley, N. (2004). "COBOL integrates with Java and .NET." Computer Weekly. http://www.computerweekly.com/articles/article.asp?liArticleID=133085

[17] 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

[18] Levy, S. P. (1995). "Computer Language Usage In CS 1: Survey Results." SIGCSE 27(3): 21-26.

[19] McCauley, R. and Manaris, B., (1998). "Computer science programs: what do they look like?" Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, February, pp. 15-19.

[20] 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.

[21] Merritt, S. M. (1980). "On the importance of teaching Pascal in the IS curriculum." ACM SIGCSE Bulletin , Proceedings of the eleventh IGCSE technical symposium on Computer science education SIGCSE '80 12(1).

[22] Parker, K. R., T. O. Ottaway, et al. (2006). "Criteria for the selection of a programming language for introductory courses." International Journal of Knowledge and Learning 2 (1/2) 119-139.

[23] Riehle, R. (2003). "SEPR and programming language selection." CrossTalk - The Journal of Defense Software Engineering 16(2): 13-17, http://www.stsc.hill.af.mil/crosstalk/ 2003/02/Riehle.html

[24] Sammet, J. E. (1972). "Programming languages: History and future." Communications of the ACM 15(7): 601.

[25] Smillie, K. (2004). "People, languages, and computers:A short memoir." IEEE Annals of the History of Computing (April–June): 60-73.

[26] Smith, C. and Rickman, J. (1976). "Selecting languages for pedagogical tools in the computer science curriculum," ACM SIGSE Bulletin 8(3): 39-47.

[27] Smolarski, D.C. (2003). "A first course in computer science: Languages and goals." Teaching Mathematics and Computer Science 1(1):137-152. Retrieved November 10, 2005 from http://math.scu.edu/~dsmolars/smolar-e.pdf

[28] Solntseff, N. (1978). "Programming languages for introductory computing courses: a position paper." ACM SIGCSE Bulletin 10(1): 119-124.

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

[30] Tatnall, A. and B. Davey (2004). "Stream in the history of computer education in Australia." History of Computing in Education. J. Impagliazzo and J. A. N. Lee, Kluwer Academic Publishers.

[31] 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

[32] Wirth, N. (1993). "Recollections about the development of Pascal." ACM SIGPLAN Notices, The second ACM SIGPLAN conference on History of programming languages HOPL-II 28(3).

[33] Zeigler, S.F. (1995). "Comparing development costs of C and Ada." Rational Software Corporation, Santa Clara, Calif., March 30.

[34] Zelle, J. M. (1999). "Python as a first language." Proceedings 13th Annual Midwest Computer Conference (MCC 99), March 18-19, Lisle, IL.