A Survey of Literature on the Teaching of Introductory Programming

Arnold Pears,
Uppsala Uni., Sweden
Arnold.Pears@it.uu.se

Stephen Seidman,
Uni. of Central Arkansas, USA
sseidman@uca.edu

Lauri Malmi,
Helsinki Uni. of Tech., Finland
lma@hut.fi

Linda Mannila
Åbo Akademi Uni., Finland
Linda.Mannila@abo.fi

Elizabeth Adams
James Madison Uni., USA
adamses@jmu.edu

Jens Bennedsen
IT Uni. West, Denmark
jbb@it-vest.dk

Marie Devlin
Newcastle Uni., UK
Marie.Devlin@newcastle.ac.uk

James Paterson
Glasgow Caledonian Uni., UK
James.Paterson@gcal.ac.uk

ABSTRACT

Three decades of active research on the teaching of introductory programming has had limited effect on classroom practice. Although relevant research exists across several disciplines including education and cognitive science, disciplinary differences have made this material inaccessible to many computing educators. Furthermore, computer science instructors have not had access to a comprehensive survey of research in this area. This paper collects and classifies this literature, identifies important work and mediates it to computing educators and professional bodies.

We identify research that gives well-supported advice to computing academics teaching introductory programming. Limitations and areas of incomplete coverage of existing research efforts are also identified. The analysis applies publication and research quality metrics developed by a previous ITiCSE working group [74].

Categories and Subject Descriptors

K.3 [**Computing Milieux**]: Computers and Education; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

THEORY

Keywords

computing education research, teaching, literature survey, bibliography, introductory programming

1. INTRODUCTION

The goal of this paper is to answer the question "What body of research literature can inform instructors when designing a new introductory programming course?" The literature discussed here can help instructors to answer questions such as: "What programming language should be used?", "What tools and environments support learning, and how?", "What pedagogies have been tested, and what are the outcomes?", and "How does the new course fit into a larger computing curriculum?"

The audience for this paper includes computing faculty who are planning, designing, developing, revising or implementing a new course that introduces programming concepts to novice programmers. Design and development of such a course may be influenced by many factors, including the history and culture of the department offering the course, the needs of prospective employers, the requirements of accreditation or assessment bodies, as well as national and international perspectives on computing education.

We discuss some of these issues below. Within the local context created by these factors, the results of decades of computing education research can provide important input to the course design process. This paper seeks to provide course designers with an overview of evidence-based papers in the computing education research literature that have had an influence on the practice of teaching. Evidence-based research is defined as the use of literature and results from relevant areas of the scholarship of teaching and learning to motivate and support teaching practice. The nature and quality of evidence is determined by the level of agreement of research literature as to what is known in the area. We assume that readers of this paper have little experience or exposure to general literature in the theory of teaching and learning in higher education.

It is unlikely that a new introductory course can be planned, designed, developed, or implemented on the initiative of a single faculty member. Such an effort generally requires the creation of consensus at multiple levels: committee, department, college, university, and even possibly the concurrence of a ministry of education. As noted above, there are many actual or potential constraints on new course creation. For example, a US university may be constrained by articulation agreements with other universities or with two-year colleges, or by the curricula for "advanced placement" examinations taken by secondary students. European universities may be constrained by national agreements relating to the Bologna Accord[1]. More generally, a local employer may try to in-

[1]"The Bologna Process aims to create a European Higher Education Area by 2010, in which students can choose from a wide and transparent range of high quality courses and benefit from smooth recognition procedures.", see http://ec.europa.eu/education/-

fluence a university to structure a curriculum around its employment needs, or the department may have a strong position in a particular research area.

Increasing focus on teaching practice over the last fifteen years has resulted in a worldwide shift in education from instructor-centered to student-centered learning. Boyer's discussion of the "Scholarship of Teaching" in [16] gave rise to a worldwide discussion about academic engagement in teaching practice and to an increased emphasis on a scholarly approach to teaching. The scope has since expanded to encompass learning as well. Such an approach requires instructors to be aware of two distinct categories of research literature: work that deals with theories of teaching, learning and knowledge acquisition relevant to the discipline (e.g. [95, 94, 65, 63]), and work dealing specifically with teaching issues in the subject matter of instruction (e.g. [70, 81]).

This report provides an overview of research in both of these categories. It highlights publications which we have identified as particularly relevant to teaching and learning designs for introductory programming courses. Specifically, we identify and comment on research in the following categories:

- Curricula
- Pedagogy
- Language choice
- Tools for teaching

Where possible, we summarise existing research and make concrete recommendations for teaching based on empirical research results in tertiary computing education. In addition, we identify shortcomings in the research to date and recommend that future research in such areas should be given a high priority.

2. METHOD

Papers for consideration by the working group were solicited from the community by sending a direct email request for contributions to approximately 100 academics active in computing education research worldwide who were known to the working group coordinators. The request for input was also distributed via the SIGCSE mailing list to 1077 people. This resulted in 62 paper nominations from 10 individuals outside the working group. These papers were combined with additional paper nominations solicited from colleagues or nominated directly by the working group participants to form a total pool in excess of 180 papers.

Area	Considered	Reviewed	Included
Curriculum	21	21	8
Pedagogy	48	21	11
Languages	50	48	12
Tools	60+	60	14

Figure 1: Paper selection process summary

The papers were assigned by the working group leaders to one of the following four sub-areas: Curricula, pedagogy, languages, tools. An overview of the numbers of papers nominated, selected for review and finally included in the annotated bibliography is given in figure 1. Each paper selected for review was read by at least two (usually three) working group members and an annotation and motivation for why the paper should be included was prepared. Additionally the papers were classified using a previously published taxonomy, and citation information collected using Google Scholar. A taxonomic breakdown of the papers finally selected is presented using a modified version of the Lisbon taxonomy [74] in figure 2.

policies/educ/bologna/bologna_en.html

Area/Category	Emerging	Influential	Synthesis
A: Small Scale	10	1	–
B: Institutional	3	6	2
C: Prob/Solns	5	6	7
D: CER	–	4	1

Figure 2: Characteristics of Papers Included in the Annotated Bibliography

The annotated bibliography comprises 45 papers selected from the nominations. In selecting papers the following criteria were applied. A paper should be

- relevant and informative to practitioners
- representative of the area
- provide, as far as possible, good coverage of appropriate scholarship from the literature on teaching and learning of introductory programming.

2.1 Methodological considerations

The majority of studies reviewed were located in area A of the Lisbon taxonomy table. The concentration of small-scale studies is understandable, since it is easy for practitioners to study their own instructional settings. However, it is difficult to aggregate results from such papers to produce well-supported arguments for particular approaches to teaching. A more consistent approach to reporting the results of individual studies would facilitate generalizability. In addition, there is a clear need for larger scale studies that address underlying issues associated with programming language learning. In selecting papers for the annotated bibliography, we tried to avoid a concentration of papers in category A.

Factors that contribute to determining the level of "influence" of a paper, tool or language can vary widely. When considering the impact or influence of a paper on tools or languages, we have considered the distribution and use of particular tools and languages, as well as the citations received by papers dealing with the use and evaluation of these artifacts. Further criteria for selection and evaluation papers specific to the classes of paper considered in each sub-area were developed by the group members. These criteria are more fully discussed later.

3. A GUIDE TO THE LITERATURE

3.1 Curricula

An introductory programming course should always be considered within the context and structure of a computing curriculum. Traditionally, most computing students were enrolled in a computer science curriculum. Other long-standing curricular possibilities include computer engineering, information systems (computer applications to business) and information science (computer applications to libraries). In recent years, many more computing curricula have been proposed and implemented, including software engineering, information technology, multimedia applications, and games programming. How does the introductory programming course fit into such a wide variety of computing curricula? How does the distinction between programming and coding play out in this relationship? What are the key national and international trends in university computing curricula?

In the United States, the major professional computing societies (the ACM and IEEE Computer Society) have worked jointly for many years on recommendations for computing curricula. Their most recent computer science curriculum recommendation was released in 2001; this recommendation is currently under revision. The two societies have also developed curriculum recommendations in computer engineering, software engineering, and information systems. An information technology curriculum is currently under development. Given so many similarly titled, but seemingly different, computing programs and curricula it is easy for faculty, administrators, students, and parents to become confused. In order to reduce the confusion, the two professional societies have prepared an overview document [21] that compares and contrasts the curricula for which they have made recommendations.

In Europe, the Bologna Accord establishes protocols and processes governing university degree programs within the European Higher Education Area (EHEA). Each signatory to the Bologna Accord has undertaken to work towards establishing procedures and guidelines under which universities will move toward harmonisation of their educational programmes. As a consequence, the development of a new introductory computing course may be constrained by national arrangements relating to the Bologna Accord. For a summary of the effects of movement towards the Bologna model for the European Higher Education Area see [40].

Any introductory computing course embodies, formally or informally, a number of assumptions. First, a computing course always makes assumptions about its underlying computational metaphor and programming paradigm. Course designers should be explicitly aware of the choices made with respect to programming metaphor and paradigm. They must also be aware of the spectrum of possible choices. For example, while most introductory courses assume a "computation as calculation" metaphor [91], this is not the only possibility. Stein has made a strong argument for replacing "computation as calculation" with "computation as interaction" [90]. She has also designed and implemented an introductory course based on this metaphor [33]. As another example, many introductory courses now adopt the object-oriented programming paradigm, and it is often assumed that this is the only available paradigm. However, some introductory courses have always chosen to use the functional programming paradigm. An important recent case in point is the TeachScheme project [38]. An introductory computing curriculum may also make an explicit choice to emphasize a particular theme that will recur throughout the course. Possibilities for such themes include abstraction [60] and components [51]. Finally, it is also important to consider the target audience for a new course. [39] explores the ways in which an introductory computing course was received by students enrolled in different programs.

3.2 Pedagogy

The landscape in which a language and/or tool is deployed consists of the subject matter and skills acquisition associated with taking the course, and the approaches to teaching and learning programming adopted by the instructor. While the curriculum defines what is to be taught, pedagogy deals with the manner in which teaching and learning are managed in order to facilitate desired learning outcomes. The approach taken to structuring the learning situation influences both the expected outcomes and the techniques adopted to achieve them.

There is a large body of general literature on the roles and methods teachers and students encounter in higher education. As noted in the introduction, Boyer's work [16] on *Scholarship of Teaching*[2] has created considerable interest in academic teaching practice. There is also a significant body of literature on approaches to teaching which considers both teacher and learner perspectives. Some of this literature, while not central to the teaching of computing, can be of great assistance to teachers. Student approaches to learning [95] and the influence of learning environments on how students relate to learning tasks [94] is a focus of the work of Trigwell, Prosser, Ramsden and others. They have also studied how university teachers approach and experience their teaching [63]. The principle of constructive alignment, which discusses the relationship between expected learning outcomes, instructional design and assessment practices is also highly relevant to the teaching of computing [14]. Recent contributions of Meyer and Land [65] deal with student learning difficulties and how students deal with learning troublesome topics. The insight that can be gained from Meyer and Land's approach is valuable when teaching a topic (in this case introductory programing) that the computing community recognises as problematic for many students. The context of the teacher and the role of professional experience in professional development of tertiary educators, with particular relevance to educational reform, has also been examined by Driel et al. [34].

While it is important for computer science teachers to be aware of this literature, much of it is not directly relevant to teaching introductory programming, nor is the research situated in the computing education research (CER) discourse. Consequently, this paper concentrates on research papers situated in and informing the practices associated with teaching introductory programming. Our intention is to provide an overview of what is known about two major aspects of learning about programming: gaining broader perspectives on and deeper insight into the learner's experience, and gaining familiarity with the didactic techniques that have been demonstrated to have a positive impact on learning outcomes.

One difficulty encountered in identifying relevant literature is trying to define what "learning programming" means to our community. Some prominent early members of the computing research community espouse the view that programming is mathematically based, and that good programming requires programmers to understand how to prove prop-

[2]Now usually termed "Scholarship of Teaching and Learning"

erties of the programs they write [29, 50]. Textbook writers typically focus on syntax and language features [81], while others focus on skills associated with problem formulation, requirements analysis and problem solving [70].

Studies of programming courses can be loosely grouped, based on the primary emphasis of the instructional setting. In the following discussion we define and explore three such emphases: problem solving, learning a particular programming language, and code/system production. Each of these areas is represented in the annotated bibliography in appendix A.

A strong movement in computing and informatics education considers programming to be an application of skills in problem solving. Many educators therefore feel that learning which addresses the development of problem solving skills should be a major goal of the instructional design. Palumbo [70] conducted an extensive review of literature regarding the connections between learning programming languages and problem solving skills. He summarises many studies (of imperative programming, mostly in primary or secondary school settings) on the transferability of learning a programming language to problem solving skills. A key aspect of problem solving is the ability to solve two classes of problem, those similar to problems one has solved previously, and new problems, perhaps in a different application domain. Palumbo distinguishes between these classes and goes on to discuss two important dimensions of skill transfer: the distance and generalizability of transfer. Distance distinguishes between *near transfer* (the ability of the learner to apply "skills and expertise to a new problem solving domain that is similar in its stimulus features to the domain in which proficiency and skill have already been obtained and established"(p. 70)), and *distant transfer* ("the transfer of skills and expertise to a new problem-solving domain that is distinctly different in its stimulus features to the problem-solving domain where expertise had already been established" (p.71)). Generalizability distinguishes between *specific transfer*, which "can be defined as transfer of one or more specific skills to a new problem-solving domain" (p.70) and *generalized transfer*, which "can be defined as the transfer of general problem-solving strategies and procedures from one problem solving context to another."(p. 70). Palumbo's conclusion is that one can expect neither distant nor general transfer to happen in an introductory course, since the time spent in practice is simply too limited.

Perhaps one of the most traditional views of learning programming prioritises the structure and syntax of the language itself. Most introductory programming textbooks are structured according to the constructs of the particular programming language used. Robins, Robins and Rountree [81] conclude that "Typical introductory programming textbooks devote most of their content to presenting knowledge about a particular programming language" (p.141). Approaches in this tradition are not necessarily limited to direct instruction related to the syntax, but may have to do with features of a family of languages. Some of the most influential movements include the Object Oriented early initiative and the component-first approach advocated in [51]. Both approaches focus on constructs central to expressing solutions in specific programming paradigms.

The desired outcomes of programming courses as well as many of the assessment artifacts created by students during the learning process are simple programs or software systems. Another way to approach teaching programming is to base learning in the context of code and system development. If one takes this approach it is useful to be aware of research that deals with novice and expert approaches to code development. In this paper we treat this approach as distinct from approaches to problem solving, though there may be relationships between the two.

Soloway [86] did some significant early work on the theory of plans and schemas. The application of this theory to teaching lies in making the plans and schemes visible to students during instruction. The idea is that this will assist students in recognising appropriate plans and schemes when they encounter code development requirements with similar characteristics. This has similarities to arguments advanced by Green and Petre [44, 43] on the cognitive load associated with languages and the extent to which the structure of programming languages acts to obscure or elucidate the underlying plans and cognitive structures of the original author.

Whatever approach is adopted, it is clearly important to decide what one thinks programming is about before setting out to teach others how to do it. There is almost certainly no single answer to the question "What does it mean to teach/learn programming?". But we should give serious consideration to what we want, can and should achieve in the first course.

3.3 Language choice

During the last four decades, many languages have been used for teaching introductory programming. The language choice is usually made locally, based on factors such as faculty preference, industry relevance, technical aspects of the language, and the availability of useful tools and materials. The process has become increasingly cumbersome as the number of languages has grown. The aim of this section is to survey literature that can help educators make informed decisions when choosing a programming language for an introductory course, and/or to understand difficulties that may occur with the use of a particular language. We begin by reviewing language and paradigm trends in introductory programming courses. We consider paradigm along with language because languages may be taught in a manner that does not take advantage of the paradigm to which a language belongs [27, 28]. We also look at trends in industry. Next, we review and discuss the types of language-related papers found in the literature.

Today, C, Java and C++ top the list of the most widely used programming languages, both in industry and education. The Tiobe programming community index[3] is updated once a month. It uses search engines to give an indication of the popularity of programming languages. There are also a number of surveys assessing the usage of programming languages in introductory programming courses. Paul Tymann's data comes from an online survey of the SIGCSE community carried out in 2005 [96]. deRaadt et al [27, 28] conducted similar surveys in Australia and New Zealand in 2001 and 2003, and a much earlier one was done by Stephenson and West in 1998 [92]. These surveys indicate that the most widely used languages have not changed during this time span - Java, C++ and C have continuously been among the top four languages. deRaadt et al [27, 28] also studied

[3]http://www.tiobe.com

the relationship between the paradigms and languages used. They conclude that most institutions use an object-oriented language, but many use such a language to teach procedural programming, and less than 10% of institutions teach functional programming.

Despite the popularity of languages such as Java, C and C++, there has been much debate about the suitability of these languages for education, especially when introducing programming to novices (for example [66, 47, 12, 22, 23]). These languages have not been designed specifically for educational purposes, in contrast to others that have been designed with this specific purpose in mind (e.g., Python, Logo, Eiffel, Pascal).

As early as 1970, many writers had pointed out the disadvantages and limitations of available languages [78, 30]. Since then, there has been much research addressing these limitations as well as presenting different perspectives on the advantages of a given language. In this section of the paper, we review language-related papers which address the merits and demerits of specific language choices. Most of the papers fall into one of the following three categories: papers focusing on a single language; papers comparing two or more languages; and papers presenting criteria and/or pedagogical foundations for facilitating language selection.

Many of the papers we discuss appear in the form of experience reports (e.g. "We have used this language in our introductory course, and this is what we found") [79, 80, 47, 37]. These papers are usually motivated by faculty members' dissatisfaction with the language used. Dissatisfaction with the use of C++ or Java as the language in the first course has caused faculty to both look back at other languages used in the past [8] and/or to consider adopting languages such as Scheme or Python. For example, the TeachScheme Project takes an existing language and teaches it using an approach designed explicitly around pedagogy. The roots of this project can be found in the ideas of Abelson and Sussman [1]. However, the TeachScheme project is asserted to have a "cognitive development environment specifically developed for beginners ... containing a full-fledged suite of tailor-made tools." [38] We also reviewed papers comparing two or more languages. The papers in this category are interesting, as they can inform teachers who are thinking about languages and need guidance on which one to choose. Examples include [41, 62, 8, 18, 17, 58].

The papers in a third category are language-independent. They present intrinsic and extrinsic criteria that characterize a good introductory language [45, 85, 64, 43]. Intrinsic criteria are directly related to the language itself. They include technical aspects (compiled vs. interpreted, dynamic vs. static typing, visual vs. textual, ...), purpose-related aspects (pedagogical vs. general) and paradigm. Extrinsic criteria, on the other hand, relate to external factors (industry demands, trends, student demand), accessibililty (availability of good texts, other supporting material, tools, ...) and the focus of the introductory programming course (algorithmic thinking, social skills, design, ...).

Schneider [85] agrees with Gries [45] that the targets of an introductory programming course should be problem solving and algorithm development. According to Schneider, a programming language "should be based on two critical and apparently opposing criteria: richness and simplicity - 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." (p. 110) If a language has a small set of constructs/features, although it may be possible to say what one wants, it may be excessively complicated to do so. On the other hand, if the language has a very large set of constructs/features, it may be difficult to assimilate them all. Green [43] and McIver [64] discuss other contradictions that lead to the tradeoffs that must be considered when choosing a programming language.

Intrinsic criteria may be supported by research in pedagogy or psychology. For instance, one of the most common criteria listed in most of the papers is syntactic simplicity. Java, C++ and C are often criticized for being too verbose, enforcing notational overhead that has little to do with learning to think algorithmically and writing structured programs. Palumbo [70] cites Linn and Dalbey, who described an ideal chain of cognitive accomplishments for programming instruction as follows: a novice starts with learning individual language constructs, progresses to being able to combine these constructs to solve specific programming problems, and finally to acquire general problem-solving skills that can be transferred to other domains not related to programming. The idea of having a simple syntax is clearly related to this chain; limiting the amount of specific programming features required to use the programming language effectively "allows for a smoother and faster transition to the second link in the cognitive chain when students can begin to use these commands to effectively create templates for solving programming problems" (p.79). Green [43] provides a "cognitive dimension framework" which can be interpreted as mapping intrinsic criteria to cognitive dimensions. Each of the seven steps towards more teachable languages in [64] is preceded by a pedagogical rationale.

The striking correspondence between the languages used in education and in industry is an example of the operation of an extrinsic criterion. Studies have found that market appeal/industry demand/student demand is one of the most important factors affecting language choice in computer science education [31, 27, 28]. As an example, Rochester Institute of Technology dropped Eiffel as its introductory programming language primarily because of student demand to learn a language that would be useful after graduation.[4] Following trends in programming language development and popularity might seem a reasonable and motivated approach to choosing an appropriate language for teaching introductory programming. However, faculty should be aware of the fact that popularity of a language does not necessarily mean that it is good for teaching purposes. Gries [45] cites McCracken's 1973 remark (which was based on McCracken's survey of 40 representative universities) "Nobody would claim that FORTRAN is ideal for anything, from teachability, to understandability of finished programs, to extensibility. Yet it is being used by a whopping 70% of the students covered by the survey, and the consensus among the university people who responded to the survey is that nothing is going to change much anytime soon" (p. 88). In fact, FORTRAN continued to be widely used in education until Pascal appeared. Böszörményi [15] makes similar negative remarks about Java, yet Java has been used from its release in 1995 to the present day. Perhaps this is because, as Böszörményi [15] says, "the approach we think today to be the best (if there is one) was not the first and probably

[4]According to a conversation with Paul Tymann at ITiCSE2007. June 2007, Dundee, Scotland.

will not be the last. The university should not try to teach ultimate wisdom: it should rather teach students to think about different possibilities" (p. 142).

The goal of this section was to provide a simple categorization of papers dealing with language choice for the introductory programming course. The language-specific papers provide the reader with experience-based information. The comparison papers serve a similar purpose but also provide models for how languages can be compared. Finally, the criteria papers provide checklists that can be used to decide whether a language has the features of a good teaching language. In the end, the choice of a programming language should be based on its suitability to the purpose. An appropriate language can only be chosen after course goals and learning outcomes have been specified. As Böszörményi [15] states, a choice of language cannot be made in isolation, but must be made in the context of the entire curriculum. As there are no worldwide accepted standards for the CS curriculum or the goals for the courses in that curriculum, one cannot expect there to be a single language that would fit all curricula and courses.

3.4 Tools research

Programming tools are generally developed to meet professional programmers' needs. They often have extensive sets of concepts and features that are problematic for novices, and their error and warning messages may be hard for novice users to understand. For this reason, there have been efforts to develop tools specifically designed for the needs of beginning programmers. Another motivation for tool development projects has been to reduce or simplify teacher workload. Typical targets of these projects are automatic assessment, plagiarism detection and course management.

Many research projects have been devoted to tool development. Valentine's survey [97] classified papers on topics related to CS1 and CS2 presented at SIGCSE Technical Symposium conferences between 1984 and 2003. 22% of these papers (99 out of 444) presented software tools designed to aid teachers and/or students. However, in spite of this large body of research and development work, only a small number of tools have been used outside their home institutions. Very few tools have seen wide adoption, even although most important tools can now be downloaded from websites, are easily found using search engines and are often supported on most common platforms.

There are several reasons for this. First, tools research projects often originate in a desire to solve a local problem, either in a specific institution or a specific course. As a result, multiple tools have been developed for similar purposes. Furthermore, local differences in instruction have made it difficult for other teachers to adapt tools to their needs; support for tool modification is rare.

A second reason for limited tool dissemination is a consequence of the research process. Tools are often developed using research funds or as a part of doctoral research; in this case, the primary goal is to investigate novel approaches and methods; the tool is a research prototype that faces a total lack of support after the developer has moved on. Such tools typically need much work before they can be easily adopted for use in different institutions, but there is no one to do this work. This leads to the third issue: research funding does not generally support customization and generalization of tools to meet the needs of a larger set of teachers and institutions. Such work is essential for dissemination but is rarely considered worthy of scientific publication.

On the other hand, there are several excellent examples of tools that have been widely adopted and that have had considerable effect on teaching practices. These include the BlueJ programming environment [59], the Karel family of microworlds of robots [72], and the CourseMarker automatic assessment tool and its predecessors [49]. These tools have been developed by larger groups of teachers and researchers, so a critical mass of people has been available to maintain and develop them over a long period.

The purpose of this paper is not to survey tools research. Rather, we aim to give an outline of the field and to point out sub-areas that may be particularly relevant to teachers of introductory programming courses. We have identified literature surveys for particular topics. The annotated bibliography includes a small sample of papers that describe some specific tools. These papers provide examples of work in several sub-fields of tools research that have had demonstrated success or that we consider potentially important in the future.

3.4.1 Visualization tools

Humans are very good at processing visual information. On the other hand, most programming concepts, algorithms and data structures are abstract concepts with no obvious graphical form. Moreover, programs and algorithms are dynamic artifacts; capturing their essentials is a challenging task for novices. It is therefore not surprising that much research has been devoted to the visual presentation of the structure and operation of programs and algorithms.

However, much research has shown that simple visualization is not enough to support successful learning of dynamic algorithmic processes. Students may look at dynamic visualizations without understanding the context or deeper meaning. The key result of the meta-analysis in [52] was that when students are engaged with a tool in some kind of activity, such as responding to questions, making predictions or experiments, they learned better. An ITiCSE 2002 working group [69] developed a taxonomy of student engagement in algorithm visualization. This taxonomy has had a considerable effect on how algorithm animation tools are being developed and how they are used in practice.

[76] gives a good if somewhat dated survey of software visualization. Such tools can be categorized in several ways. A key distinction is that between tools which visualize the structure or execution of code, such as Jeliot and jGRASP [67, 54], and concept or algorithm animation tools which may omit the code or make it available as pseudocode, such as JHAVÉ and MatrixPro [68, 56].

Code visualization tools focus on visualization of *static* structures or display the *dynamic* aspects of program execution, and are often associated with programming environments, discussed below. Tools that reflect code-level aspects of program behavior, showing execution proceeding statement by statement and visualizing the stack frame and the contents of variables (e.g., DDD [101]), are sometimes called *visual debuggers*, since they are directed more toward program development rather than understanding program behavior.

Algorithm animation tools capture dynamic behavior of code in different ways. Possibilities include library calls (e.g. Tango, Polka) [87, 89] and script languages (e.g. Animal, Samba, JHAVÉ) [88, 68]. Another possibility is to use graphical interaction (as in MatrixPro [56]); such *algorithm simulation* allows a teacher to demonstrate algorithm execution directly. For example, a key can be inserted into an AVL tree by dragging the key from the array and dropping it as a leaf into the depicted tree. ALVIS Live [53] provides an environment which allows novice programmers to construct algorithms and visualizations graphically.

3.4.2 Automated assessment tools

The need for automated assessment tools comes from the large numbers of students enrolling in programming courses at many institutions. In this situation, checking assignments can become unmanageable, limiting the number and character of assignments and impeding learning for students. Automated assessment can yield benefits for both teachers and students.

Two survey papers summarize the field in some detail. Ala-Mutka [3] describes the methods and techniques used by automated assessment tools and shows how they are generally used. Douce et al. [32] gives a historical perspective on the development of the field from the 1960s to the present. One additional useful reference is a 2005 special issue of the ACM Journal on Educational Resources in Computing devoted to automated assessment (vol. 5, no. 3).

From the teacher's point of view, current tools provide a variety of techniques for analyzing program features. Most tools check the correctness of program execution (e.g., CourseMarker [49], BOSS [55]). Typically, this is done by comparing a student's program output with results provided by the teacher's model solution program. Other approaches compare internal data representations [83], or do more detailed program testing [35]. In addition to determining correctness, an assessment tool may also analyze program efficiency and/or complexity, usage of particular language features, coding style, and some aspects of program design.

The tools can be used in different ways. Some teachers use them to fully automate the grading process, while others use the tools to generate an initial evaluation that can indicate problems with a student's program. Automated assessment tools can also reduce the subjectivity resulting from grading by multiple teachers. Some tools summarize grades given by different teaching assistants, allowing the teacher to modify them if inconsistencies are found [2].

From the student's point of view, these tools can provide valuable formative evaluation in a timely manner, making it possible to learn from errors and improve work before final submission. However, the availability of instant feedback can encourage a trial-and-error approach to programming. This problem can be mitigated by limiting the number of submissions permitted, or by specifying a minimum delay between subsequent submissions. Edwards [35] has developed WebCAT, an innovative tool that deals with this problem in quite a different way. WebCAT requires students to submit test data with each program and assesses how well the program passes these tests, as well as the extent of the program's test coverage.

Finally, we note that automatic assessment is not limited to program assessment. CourseMarker supports the assessment of flow charts [49] by transforming them to programs that can be executed. [36] presents a tool for assessing graphical user interfaces. TRAKLA2 [61] allows students to simulate the operation of algorithms by direct interaction. The students drag and drop objects on the screen to simulate how a specific algorithm would modify the data structure. The simulation sequence is recorded and feedback can be given.

3.4.3 Programming Environments

Programmers at all levels of experience need to work within environments which give them access to the tools which they must use to accomplish their tasks. This implies that an environment must provide the capability to build and execute a program. For Java, the most basic environment would consist of a simple text editor for editing Java files and a Java Software Development Kit which provides command line tools to compile and execute programs. Most experienced programmers prefer to work within a more sophisticated Integrated Development Environment (IDE) which enhances productivity by providing access to additional capabilities, such as

- organization of program components and resources into projects
- advanced language-specific editing features (e.g., syntax highlighting, code completion)
- integration of additional support tools (e.g., visual debuggers, test tools, documentation generators, code analyzers)

IDEs are available for virtually all widely used programming languages. An IDE designed for professional use can be used in an educational environment, and some excellent free and/or open-source tools are available (e.g., Eclipse, NetBeans). However, for introductory courses, the advantages of a professional IDE may be outweighed by its complexity, requiring students to spend excessive time learning the tool. To counter this, a range of integrated environments have been designed specifically for novice programmers. These fall into two broad categories:

- Programming support tools, which support learners in the creation of programs within a standard execution environment.
- Microworlds, which provide environments based on physical metaphors; program execution is reflected in the visual state of concrete objects within the environment.

Kelleher and Pausch [57] give an extensive taxonomy of programming environments for novices. Much of the literature on programming environments consists of papers which describe the rationale for and the use of a particular tool. However, some detailed evaluations of the educational impact of specific tools have been published. Gross and Powers [46] have conducted a meta-study of such evaluations.

3.4.3.1 Programming support tools.

These tools typically offer a limited subset of the capabilities of a professional IDE. In some cases, (GILD [93], NetBeans BlueJ Edition [5]) an IDE is modified to hide its more advanced features. Other tools (BlueJ [59], jGRASP [54], Dr Java [4] and JPie [42]) are designed specifically to support

[5]http://edu.netbeans.org/bluej

the learning of particular programming concepts. It is interesting to note that programming support tools which appear to have been widely adopted are predominantly aimed at Java. This may reflect community acceptance of Java, but it may also respond to a sense that Java learners need more support.

These tools have facilities that are shared with the professional environments. However, each tool also has one or more specific programming support features. Examples of these features given below.

Interactive incremental code execution: It can be helpful for students learning Java to have access to an interface which allows the student to evaluate expressions, instantiate objects and call methods in a similar way to an interpreted language such as Python. Dr Java [4] provides a good example of this feature.

Visualization: Ragonis and Ben-Ari [77] describe advantages and disadvantages of static visualization, and conclude that its use should be augmented with dynamic visualization. BlueJ offers only static visualization of and interaction with classes and objects, but its extensibility has enabled the integration of Jeliot for dynamic visualization. jGRASP [54] is an example of a tool which provides both static and dynamic visualization capabilities.

Editing and syntax support: Student-oriented IDEs tend to provide fairly simple editors with limited code-completion capabilities. Syntax construction is supported by direct manipulation programming environments (DMPEs) such as JPie, which allow language elements to be manipulated with gestures such as drag-and-drop rather than by editing code; by tools which assist program construction by allowing students to generate code segments from templates based on established patterns or code frameworks [26, 5]; or by graphical development tools which allow the user to construct programs graphically, for example by using flowcharts [20]. An interesting example of templating is the support included in BlueJ for generating and running unit test code [71], which gives novice programmers access to a fairly complex code framework (jUnit).

3.4.3.2 Microworlds.

The physical metaphor in a microworld environment is intended to decrease the distance between students' mental models and the programming language [100]. The 'world' can be a visualization on a computer screen or a real physical environment.

The recent literature on microworlds is directed towards supporting the objects-first approach [25, 100, 7, 48, 84]. Microworlds differ in the programming languages they use, the nature and scope of their metaphors and the level of programming support which they provide.

One distinguishing characteristic is language. The original *Karel the Robot*, in which the program controls robots that live in a grid of streets and intersections, used its own language, *Karel* [72]. More recently, Karel's conceptual framework has been extended to object-oriented languages [11, 7, 19]. *Alice* [25] is both a popular language and an environment.

A second characteristic is the underlying metaphor. The Karel, Jeroo [84] and Robocode [6] frameworks are built on specific metaphors. In contrast, Alice and Greenfoot [48] provide frameworks in which many different microworlds can be created. The M.U.P.P.E.T.S. system [13] allows users to create objects within a multi-user graphical 3D environment.

The level of coding support varies widely. At one extreme, Alice provides an integrated environment with DMPE features: at the other, some Karel variants are implemented as program libraries leaving the choice of coding environment open. An alternative approach to the Karel concept is taken by *objectKarel* [100],which provides a didactic development environment with a range of tutoring features.

3.4.4 Other tools

Some other categories of tools may be relevant for teachers. Plagiarism is often a serious concern, and tools have been developed to cope with the problem. Some automatic assessment tools like Boss have integrated plagiarism detection. On the other hand, several specific tools are available for this purpose, such as JPlag [7], MOSS [8], and YAP [9].

An area of research with some relevance for teaching is *intelligent tutoring systems* for introductory programming. These expert systems were developed to support students in learning. A recent survey by Pillay [75] summarizes some of this work. Few of these systems, though, have seen wide use. Many have remained prototypes or are now difficult to access. Some tools in this category are worth noting, such as the LISP Tutor [6] and ELM-ART [99] dedicated to introductory LISP programming.

4. CONCLUDING REMARKS

The goal of this paper has been to give an overview of research issues relating to the teaching of introductory programming, with specific reference to curriculum, pedagogy, and languages and tools for supporting learning. It has provided material from the body of discipline-specific research in each of these areas. We conclude that despite the large volume of literature in this area, there is little systematic evidence to support any particular approach. For that reason, we have not attempted to give a canonical answer to the question of how to teach introductory programming. Rather, the paper seeks to provide support for a wide variety of approaches, and to help teachers to provide scholarly motivations for course design choices with respect to learning situation, instructional approach, selection of a language, and use of support tools and technologies.The annotated bibliography is intended to support practitioners seeking to inform themselves on these issues.

After decades of exploratory research on teaching programming, it has become clear that studying individual learning settings is no longer sufficient. Our literature review leads us to believe that larger-scale systematic studies are vital to a better long-term understanding of how to teach this key element of the field. We suggest that such studies commence with a comprehensive overview of the available evidence, extending the pioneering work of Robins et al. [81]. In addition, they should provide strategically significant new empirical results, allowing the community to use research to begin to resolve key questions regarding what to teach beginning programmers and how to teach programming.

[6]http://robocode.sourceforge.net

[7]https://www.ipd.uni-karlsruhe.de/jplag

[8]http://theory.stanford.edu/ aiken/moss

[9]http://luggage.bcs.uwa.edu.au/ michaelw//YAP.html

5. REFERENCES

[1] H. Abelson, J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.

[2] T. Ahoniemi and T. Reinikainen. ALOHA - A Grading Tool for Semi-Automatic Assessment of Mass Programming Courses. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research*, pages 139–140. Department of Information Technology, Uppsala University, 2006.

[3] K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[4] E. Allen, R. Cartwright, and B. Stoler. DrJava: a lightweight pedagogic environment for Java. *SIGCSE Bulletin*, 34(1):137–141, 2002.

[5] C. Alphonce and B. Martin. Green: a customizable UML class diagram plug-in for Eclipse. In *Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 168–169. ACM Press, 2005.

[6] J. R. Anderson and E. Skwarecki. The automated tutoring of introductory computer programming. *Communications of the ACM*, 29(9):842–849, 1986.

[7] B. W. Becker. Teaching CS1 with Karel the Robot in Java. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, pages 50–54. ACM Press, 2001.

[8] K. Becker. Back to Pascal: Retro but not backwards. *Journal of Computing in Small Colleges*, 18(2):17–27, 2002.

[9] M. Ben-Ari. Constructivism in computer science education. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 257–261. ACM Press, 1998.

[10] J. Bennedsen and M. E. Caspersen. Programming in context: a model-first approach to CS1. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 477–481. ACM Press, 2004.

[11] J. Bergin, J. Roberts, R. Pattis, and M. Stehlik. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[12] R. Biddle and E. Tempero. Java pitfalls for beginners. *SIGCSE Bulletin*, 30(2):48–52, 1998.

[13] K. Bierre, P. Ventura, A. Phelps, and C. Egert. Motivating OOP by blowing things up: an exercise in cooperation and competition in an introductory Java programming course. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pages 354–358. ACM Press, 2006.

[14] J. Biggs. What the student does: teaching for enhanced learning. *Higher Education Research and Development*, 18(1):57–75, 1999.

[15] L. Böszörményi. Why Java is not my favorite first-course language. *Software-Concepts & Tools*, 19(3):141–145, 1998.

[16] E. Boyer. *Scholarship Reconsidered: Priorities of the Professoriate*. Jossey-Bass, Hillsdale, NJ, 1997.

[17] S. Brilliant and T. Wiseman. The first programming paradigm and language dilemma. *SIGCSE Bulletin*, 28(1):338–342, 1996.

[18] B. M. Brosgol. A comparison of Ada and Java as a foundation teaching language. *Ada Letters*, 18(5):12–38, 1998.

[19] D. Buck and D. J. Stucki. JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, pages 16–20, 2001.

[20] M. C. Carlisle, T. A. Wilson, J. W. Humphries, and S. M. Hadfield. Raptor: introducing programming to non-majors with flowcharts. *Journal of Computing in Small Colleges*, 19(4):52–60, 2004.

[21] Computing curricula 2005; the overview report, 2005.

[22] D. Clark, C. MacNish, and G. F. Royle. Java as a teaching language: opportunities, pitfalls and solutions. In *Proceedings of the 3rd Australasian Conference on Computer Science Education*, pages 173–179, 1998.

[23] R. Close, D. Kopec, and J. Aman. CS1: perspectives on programming languages and the breadth-first approach. In *Proceedings of the 5th annual CCSC Northeastern Conference on Computing in Small Colleges*, pages 228–234. Consortium for Computing Sciences in Colleges, 2000.

[24] A. Collins, J. Brown, and S.E.Newman. *Knowing, learning and instruction: Essays in honour of Robert Glaser*, section Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. Erlbaum, San Francisco, 1989.

[25] S. Cooper, W. Dann, and R. Pausch. Using animated 3d graphics to prepare novices for CS1. *Computer Science Education*, 13(1):3–30, 2003.

[26] L. N. de Barros, A. P. dos Santos Mota, K. V. Delgado, and P. M. Matsumoto. A tool for programming learning with pedagogical patterns. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology Exchange*, pages 125–129. ACM Press, 2005.

[27] M. de Raadt, R. Watson, and M. Toleman. Language Trends in Introductory Programming. In *Proceedings of Informing Science and IT Education Conference*, pages 329–337. InformingScience.org, June 2002.

[28] M. de Raadt, R. Watson, and M. Toleman. Introductory programming: what's happening today and will there be any students to teach tomorrow? In *Proceedings of the 6th Conference on Australasian Computing Education*, pages 277–282. Australian Computer Society, Inc., 2004.

[29] P. J. Denning. A debate on teaching computing science. *Communications of the ACM*, 32:1397–1414, 1989.

[30] E. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[31] A. Dingle and C. Zander. Assessing the ripple effect of CS1 language choice. In *Proceedings of the 2nd Annual CCSC Computing in Small Colleges Northwestern Conference*, pages 85–93. Consortium for Computing Sciences in Colleges, 2000.

[32] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: a review. *ACM Journal of Educational Resources in*

Computing, 5(3):4, 2005.

[33] A. B. Downey and L. A. Stein. Designing a small-footprint curriculum in computer science. In *Proceedings of the 36th Frontiers in Education Conference*, pages 21–26. IEEE Computer Society, 2006.

[34] J. Driel, D. Beijaard, and N. Verloop. Professional development and reform in science education: the role of teachers' practical knowledge. *Journal of Research in Science Teaching*, 38:137–158, 2001.

[35] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. *ACM Journal of Educational Resources in Computing*, 3(3):1, 2003.

[36] J. English. Automated assessment of GUI programs using JEWL. In *Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education*, pages 137–141. ACM Press, 2004.

[37] M. B. Feldman. Ada experience in the undergraduate curriculum. *Communications of the ACM*, 35(11):53–67, 1992.

[38] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The TeachScheme! Project: computing and programming for every student. *Computer Science Education*, 14(1):55–77, 2004.

[39] A. Forte and M. Guzdial. Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses. *IEEE Transactions on Education*, 48(2):248–253, 2005.

[40] U. Fuller, A. Pears, J. Amillo, C. Avram, and L. Mannila. A computing perspective on the Bologna process. *SIGCSE Bulletin*, 38(4):142–158, December 2006.

[41] E. Giangrande, Jr. CS1 programming language options. *Journal of Computing in Small Colleges*, 22(3):153–160, 2007.

[42] K. J. Goldman. A concepts-first introduction to computer science. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 432–436. ACM Press, 2004.

[43] T. R. G. Green. Instructions and descriptions: some cognitive aspects of programming and similar activities. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 21–28. ACM Press, 2000.

[44] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[45] D. Gries. What should we teach in an introductory programming course? In *Proceedings of the 4th SIGCSE Technical Symposium on Computer Science Education*, pages 81–89. ACM Press, 1974.

[46] P. Gross and K. Powers. Evaluating assessments of novice programming environments. In *Proceedings of the First International Workshop on Computing Education Research*, pages 99–110. ACM Press, 2005.

[47] S. Hadjerrouit. Java as first programming language: a critical evaluation. *SIGCSE Bulletin*, 30(2):43–47, 1998.

[48] P. Henriksen and M. Kölling. Greenfoot: combining object visualisation with interaction. In *Companion to the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 73–82. ACM Press, 2004.

[49] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas. Automated assessment and experiences of teaching programming. *ACM Journal on Educational Resources in Computing*, 5(3):5, 2005.

[50] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[51] E. Howe, M. Thornton, and B. W. Weide. Components-first approaches to CS1/CS2: principles and practice. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 291–295. ACM Press, 2004.

[52] C. Hundhausen, S. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

[53] C. Hundhausen and S. A. Douglas. SALSA and ALVIS: A language and system for constructing and presenting low fidelity algorithm visualizations. In *Proceedings of the IEEE International Symposium on Visual Languages*, pages 67–68. IEEE Press, 2000.

[54] J. Jain, I. James H. Cross, T. D. Hendrix, and L. A. Barowski. Experimental evaluation of animated-verifying object viewers for Java. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, pages 27–36. ACM Press, 2006.

[55] M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. *ACM Journal of Educational Resources in Computing*, 5(3):2, 2005.

[56] V. Karavirta, A. Korhonen, L. Malmi, and K. Stålnacke. MatrixPro - A tool for on-the-fly demonstration of data structures and algorithms. In *Proceedings of the 3rd Program Visualization Workshop*, pages 26–33, The University of Warwick, UK, July 2004.

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

[58] E. B. Koffman. The case for Modula-2 in CS1 and CS2. In *Proceedings of the 19th SIGCSE Technical Symposium on Computer Science Education*, pages 49–53. ACM Press, 1988.

[59] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13, Dec 2003.

[60] J. Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007.

[61] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267 – 288, 2004.

[62] L. Mannila, M. Peltomäki, and T. Salakoski. What about a simple language? Analyzing the difficulties in learning to program. *Computer Science Education*, 16(3):211–227, 2006.

[63] E. Martin, M. Prosser, K. Trigwell, P. Ramsden, and J. Benjamin. What university teachers teach and how they teach it. *Intructional Science. Special issue: Teacher Thinking, Beliefs and Knowledge in Higher Education*, 28((5-6)):387–412, 2000.

[64] L. McIver and D. Conway. Seven deadly sins of introductory programming language design. *Proceedings of the 1996 International Conference on Software Engineering Education and Practice*, pages 309–316, 1996.

[65] J. Meyer and R.Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49(3):725–734, 2003.

[66] R. P. Mody. C in education and software engineering. *SIGCSE Bulletin*, 23(3):45–56, 1991.

[67] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 373–376. ACM Press, 2004.

[68] T. L. Naps. JHAVÉ – Supporting Algorithm Visualization. *IEEE Computer Graphics and Applications*, 25(5):49–55, 2005.

[69] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. In *Working Group Reports from the 2002 Conference on Innovation and Technology in Computer Science Education*, pages 131–152. ACM Press, 2002.

[70] D. Palumbo. Programming language/problem-solving research: a review of relevant issues. *Review of Educational Research*, 60(1):65–89, 1990.

[71] A. Patterson, M. Kölling, and J. Rosenberg. Introducing unit testing with BlueJ. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, pages 11–15. ACM Press, 2003.

[72] R. E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1981.

[73] R. E. Pattis. The “procedures early” approach in CS 1: a heresy. In *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education*, pages 122–126. ACM Press, 1993.

[74] A. Pears, S. Seidman, C. Eney, P. Kinnunen, and L. Malmi. Constructing a core literature for computing education research. *SIGCSE Bulletin*, 37(4):152–161, 2005.

[75] N. Pillay. Developing intelligent programming tutors for novice programmers. *SIGCSE Bulletin*, 35(2):78–82, 2003.

[76] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[77] N. Ragonis and M. Ben-Ari. On understanding the statics and dynamics of object-oriented programs. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 226–230. ACM Press, 2005.

[78] A. Ralston. Fortran and the first course in computer science. *SIGCSE Bulletin*, 3(4):24–29, 1971.

[79] S. Reges. Back to basics in CS1 and CS2. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pages 293–297. ACM Press, 2006.

[80] E. S. Roberts. Using C in CS1: evaluating the Stanford experience. In *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education*, pages 117–121. ACM Press, 1993.

[81] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: a review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[82] H. Roumani. Practice what you preach: full separation of concerns in CS1/CS2. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pages 491–494. ACM Press, 2006.

[83] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, pages 133–136. ACM Press, 2001.

[84] D. Sanders and B. Dorn. Jeroo: a tool for introducing object-oriented programming. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pages 201–204. ACM Press, 2003.

[85] G. M. Schneider. The introductory programming course in computer science: ten principles. In *Papers of the 9th SIGCSE/CSA Technical Symposium on Computer Science Education*, pages 107–114. ACM Press, 1978.

[86] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[87] J. T. Stasko. TANGO: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, 1990.

[88] J. T. Stasko. Using student-built algorithm animations as learning aids. In *The Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pages 25–29. ACM Press, 1997.

[89] J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[90] L. Stein. What we've swept under the rug: radically rethinking CS1. *Computer Science Education*, 8(2):118–129, 1998.

[91] L. Stein. Challenging the computational metaphor: implications for how we think. *Cybernetics & Systems*, 30(6):473–507, 1999.

[92] C. Stephenson and T. West. Language Choice and Key Concepts in Introductory Computer Science Courses. *Journal of Research on Computing in Education*, 31(1):89–95, 1998.

[93] M.-A. Storey, D. Damian, J. Michaud, D. Myers, M. Mindel, D. German, M. Sanseverino, and E. Hargreaves. Improving the usability of Eclipse for novice programmers. In *Proceedings of the 2003*

OOPSLA Workshop on Eclipse Technology Exchange, pages 35–39. ACM Press, 2003.

[94] K. Trigwell and M. Prosser. Development and use of the approaches to teaching inventory. *Educational Psychology Review*, 16(4), December 2004.

[95] K. Trigwell, M. Prosser, and F. Waterhouse. Relations between teachers' approaches to teaching and students' approaches to learning. *Higher Education*, 37:57–70, 1999.

[96] P. Tymann, June 2007. http://www.cs.rit.edu/ ptt/apac06/Life_After_CS.pdf.

[97] D. W. Valentine. CS educational research: a meta-analysis of SIGCSE technical symposium proceedings. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 255–259. ACM Press, 2004.

[98] T. Vilner, E. Zur, and J. Gal-Ezer. Fundamental concepts of CS1: procedural vs. object oriented paradigm - a case study. In *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education*, pages 171–175. ACM Press, June 2007.

[99] G. Weber and P. Brusilovsky. ELM-ART: An adaptive versatile system for web-based instruction. *International Journal of Artificial Intelligence in Education*, 12:351–384, 2001.

[100] S. Xinogalos, M. Satratzemi, and V. Dagdilelis. An introduction to object-oriented programming with a didactic microworld: objectKarel. *Computers and Education*, 47(2):148–171, 2006.

[101] A. Zeller and D. Lütkehaus. DDD, a free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, 31(1):22–27, 1996.

6. ACKNOWLEDGEMENTS

The authors would like to acknowledge the contribution of Mark Guzdial, Raymond Lister, Bruce Weide, Päivi Kinnunen, Evgenia Vagianou, David Kay, Murali Sitaraman, Matthias Felleisen, Anthony Robbins and Valentina Dagiene, who responded to the public call for contributions. Thanks are due both to them and to our colleagues at our home institutions for their nominations of papers and for other input during the compilation of this report. We are also indebted to those attending the ITiCSE 2007 meeting for their input and discussions during the conference.

APPENDIX

A. RECOMMENDED LITERATURE ON INTRODUCTORY PROGRAMMING

[Ala-Mutka:2005] Kirsti Ala-Mutka, **'A Survey of Automated Assessment Approaches for Programming Assignments'**, *Computer Science Education*, 15(2):83–102, 2005.
Classification: Tools, C, Synthesis. Citations: 7

Annotation

This wide survey covers what can be assessed automatically and how assessment tools are used in practice. It emphasizes the need for careful pedagogical design of assignments and assignment settings. The paper contains an in-depth treatment of the types of automated and semi-automated assessment and the assessment criteria that are appropriate for programming assignments, including static and dynamic assessment of programs. The author points out that in order for developers to share the body of knowledge and the good assessment solutions resulting from the use and evaluation of these tools, they must be made more portable and interoperable. The primary conclusion of this paper is that automated assessments must be educationally sound and consistent with the course context; they must not be solely inspired by the technology.

[Becker:2002] Katrin Becker, **'Back to Pascal: Retro but not backwards'**, *Journal of Computing in Small Colleges*, 18(2):17–27, 2002.
Classification: language choice, A, Emerging. Citations:1

Annotation

This paper is a report from a department that decided to return to Pascal after experiencing problems teaching C and C++ in the introductory programming course. The author discusses the rationale behind the decision. The paper serves as a good example of how the language chosen for an introductory programming course does not necessarily have to be one of those currently most widely used. The author also makes the interesting observation that the difficulties experienced in switching to a second language in the second programming course (in this case, from Pascal to Java) may not be due to the language change but rather to the change of paradigm.

[Ben-Ari:2001] Mordechai Ben-Ari, **'Constructivism in Computer Science Education'**, *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.
Classification: Pedagogy, D, Influential. Citations: 3 (203 for the SIGCSE conference paper version)

Annotation

This paper introduced constructivism to the community of computing education researchers. It is an extension of Ben-Ari's 1998 conference paper [9]. The paper treats constructivism as an educational paradigm, giving readers tools to help them recognize the differences between constructivism and more classical educational paradigms. It discusses different dimensions of constructivism (as a theory of learning, as a theory of teaching, as a theory of education, and as a theory of cognition), as well as varieties of constructivism. The paper also discusses constructivism in science education and computer science education contexts and gives some guidelines for educators. It lists several issues that must be addressed by those seeking to use constructivism in an introductory programming course.

[Boszormenyi:1998] L. Böszörményi, **'Why Java is not my favorite first-course language'**, *Software-Concepts & Tools*, 19(3):141–145, 1998.
Classification: language choice, A, Emerging. Citations:14

Annotation

The author presents reasons why object-oriented development is not the right paradigm for the first programming course. He asserts that modularization is more important than object-orientation. The author points out and discusses weaknesses in Java that make it inappropriate as a first programming language. Although the paper is rather Java-centric, it points out many important issues that should be considered when choosing a language to use in introductory programming courses.

[Brilliant:1996] S.S. Brilliant and T.R. Wiseman, **'The first programming paradigm and language dilemma'**, *SIGCSE Bulletin*, 28(1):338–342, 1996.
Classification: language choice, B, Synthesis. Citations:17

Annotation

This article presents the results of a small survey of universities, conducted in 1996, conducted to determine the paradigm then most widely used. Although the situation has changed since the paper was published, it provides basic information about the three paradigms (object-oriented, procedural and functional) and discusses pros and cons of each of them as reported by survey respondents. For a more current discussion on the object-oriented and procedural paradigms, see [98].

[Brosgol:1998] Benjamin M. Brosgol, **'A comparison of Ada and Java as a foundation teaching language'**, *Ada Letters*, 18(5):12–38, 1998.
Classification: language choice, B, Emerging. Citations:8

Annotation

This paper identifies the intrinsic and extrinsic criteria relevant to choosing a programming language for an introductory programming course. The author evaluates the suitability of Java and Ada in education based on these criteria. Although both languages have changed since the paper was written, the paper is still interesting because of its direct comparison of an object-oriented and a procedural language.

[CC2005] **'Computing Curricula 2005; The Overview Report'**, 2005.
Classification: curricula, B, Influential. Citations=0

Annotation

While the report is not specifically relevant to those planning a new introductory programming course, its description of alternative computing curricula focuses attention on the fact that not all students in an introductory course are traditional computer science students/majors. In this context, [39] paper is also significant.

[Cooper:2003] Stephen Cooper, Wanda Dann, and Randy Pausch, **'Using animated 3D graphics to prepare novices for CS1'**, *Computer Science Education*, 13(1):3–30, 2003.
Classification: Tools, C, Influential. Citations: 8

Annotation

This paper introduces *Alice*, a language/tool which provides 3D animation and direct manipulation of elements of a programming language. The tool removes the need for students to type code and deal with syntax, allowing them to focus on learning concepts. The paper outlines the pedagogical reasons for the use of visualization and graphics in a programming course for novices, and then describes the tool in detail. It shows how the tool and language deal with graphical and animation concepts and how it helps students to learn about programming constructs and programming style. The paper also gives an overview of course practice and some student evaluation of the tool. Alice has been adopted widely, and many recent papers evaluate its use in the classroom.

[Denning:1989] P. J. Denning, **'A debate on teaching computing science'**, *Communications of the ACM*, 32:1397–1414, 1989.
Classification: Pedagogy, D, Influential. Citations: 25

Annotation

This paper contains the text of E. J. Dijkstra's "On the cruelty of really teaching computer science", as well as comments from distinguished computer science researchers. Dijsktra's point is that programs essentially are functions that produce a given output based on the input. Consequently, programming should be taught as a mathematical discipline with a strong focus on proofs of correctness. Dijkstra points out that the concept of 'bugs' in computer programs is wrong; a program is either correct or incorrect, so the correct term should be 'errors'. David Parnas, commenting on Dijkstra's manifesto, draws attention to the fact that teaching (software) engineering also includes dealing with problems where exact mathematical models are intractable. Furthermore, it is not uncommon to have errors in a formal proof. William Scherlis, another commentator, points out that one of the biggest problems is actually to create a formal model of the problem that the program is supposed to solve. This corresponds to requirements modeling, an area not mentioned by Dijkstra.

[Douce:2005] Christopher Douce, David Livingstone, and James Orwell, **'Automatic test-based assessment of programming: a review'**, *ACM Journal of Educational Resources in Computing*, 5(3):4, 2005.
Classification: Tools, C, Synthesis. Citations: 2

Annotation

The domain of interest for this survey is automated assessment of student attempts to solve programming problems. It identifies three generations of test-based assessment systems and provides many examples of approaches to automated assessment. The reader can then form conclusions on the success of these approaches and get a better idea about how contemporary systems could and should be constructed. The paper examines the nature of the programming tasks that can be assessed, the considerations the teacher must make when choosing a tool, and the shortcomings of existing tools. The final section of the paper deals with development directions for automated assessment tools. It emphasizes a key challenge for developers: how to expand the accessibility of automatic assessment systems so that they can easily be adopted by institutions. This requires that tools be standardized with regard to interoperability and learning design.

[Downey:2006] A. B. Downey and L. A. Stein, **'Designing a small-footprint curriculum in computer science'**, In *Proceedings of the 36th Frontiers in Education Conference*, pages 21–26. IEEE Computer Society, 2006.
Classification: curricula, A, Emerging. Citations: 1

Annotation

This paper describes a computing curriculum built around alternative computational metaphors and paradigms.The descriptions of the "introductory programming" and "software design" courses are particularly relevant to those proposing new introductory programming courses.

[Eckerdal:2005] Anna Eckerdal and Anders Berglund, **'What does it take to learn 'programming thinking'?'**, In *Proceedings of the First International Workshop on Computing Education Research*, pages 135–143. ACM Press, 2005.
Classification: Pedagogy, A, Emerging. Citations: 10

Annotation

Eckerdal and Berglund propose a five-level hierarchy of "what does it mean to learn to program?": 1) learning to read and write a programming language, 2) learning a way of thinking aligned with a programming language, 3) gaining an understanding of computer programs as they appear in everyday life, 4) learning a way of thinking that enables problem solving, 5) learning a skill that can be used outside the programming class. These categories were derived from interviews with 14 students by applying phenomenographical analysis to the transcripts. This analysis produces a number of qualitatively different ways of understanding "what does it take to learn program thinking?". The authors describe their hierarchy and give examples from the interviews illustrating the five levels.

[Edwards:2003] Stephen H. Edwards, **'Improving student performance by evaluating how well students test their own programs'**, *ACM Journal of Educational Resources in Computing*, 3(3):1, 2003.
Classification: Tools, C, Emerging. Citations: 14

Annotation

Web-CAT is an innovative automatic assessment tool that emphasizes how well students test their programs. The tool has been demonstrated to have considerable positive effect on students' program quality and on their attitude to programming. The paper also reviews some related background work on automatic grading systems and contains some useful discussion and references.

[**Felleisen:2004**] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, **'The TeachScheme! Project: computing and programming for every student'**, *Computer Science Education*, 14(1):55–77, 2004.
Classification: language choice, B, Emerging. Citations:4

Annotation

The paper describes using Scheme for teaching programming in secondary schools. There are suggestions that it will also be effective for use at tertiary institutions. The roots of the project can be traced back to the ideas to the ideas of Abelson and Sussman [1], but the authors of this paper have taken an existing language and designed a teaching approach explicitly for pedagogical reasons. The approach builds on the authors' ideas that programming education should focus on three aspects: 1) a simple, stratified language, 2) an enforcing programming environment and 3) a rational design recipe. The project serves as a model for a pedagogical approach to teaching programming using a series of sublanguages based on Scheme. The approach can be translated to other languages as well but it should be noted that providing a similarly user-friendly environment could require enormous effort).

[**Forte:2005**] A. Forte and M. Guzdial, **'Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses'**, *IEEE Transactions on Education*, 48(2):248–253, 2005.
Classification: Curricula, A, Emerging. citations:7

Annotation

This paper contrasts student motivations and reactions to three introductory programming courses: a conventional course, one directed at engineers, and one directed at general audiences. Anyone planning a new introductory course needs to think about the issues of audience and motivation. The paper provides an excellent introduction to these issues.

[**Giangrande:2007**] Ernie Giangrande, Jr., **'CS1 programming language options'**, *Journal of Computing in Small Colleges*, 22(3):153–160, 2007.
Classification: language choice, B, Synthesis. Citations:0

Annotation

This is a historical survey. The author reviews the programming languages that have been used in introductory programming education, starting with assembly language. He discusses the reasons for the changes that have taken place over period of the survey. The author poses a list of questions that faculty should address when choosing the most suitable language for an introductory course. He points out that the first step in the selection process should be the specification of course goals. He gives examples showing how different goals can determine the choice of language.

[**Gries:1974**] David Gries, **'What should we teach in an introductory programming course?'**, In *Proceedings of the 4th SIGCSE Technical Symposium on Computer Science Education*, pages 81–89. ACM Press, 1974.
Classification: language choice, D, Influential. Citations:32

Annotation

The paper includes ideas on how to make a big language small and how to teach problem solving and algorithm development. The author divides the language into basic (i.e. elementary) statements and control statements and suggests that these are the only ones necessary for implementing simply structured algorithms. He also presents reasons for why he finds algorithms written in programming languages difficult to write, read and understand. The paper is dated and therefore lacks a discussion about, for example, modularization. The paper is still worth reading as it has served as a base for much work over the ensuing decades.

[**Gross:2005**] Paul Gross and Kris Powers, **'Evaluating assessments of novice programming environments'**, In *Proceedings of the First International Workshop on Computing Education Research*, pages 99–110. ACM Press, 2005.
Classification: Tools, D, Synthesis, Citations: 3

Annotation

The paper discusses the lack of a model for the development of programming knowledge in novices and how this limits our ability to fully assess novice programming tools. The study analyzes assessment studies of some of these tools - BlueJ, Alice, Jeliot, Lego Mindstorms and RAPTOR, and broadly categorizes the techniques that were used for the assessments as anecdotal, analytical and empirical techniques. It then outlines an evaluation rubric for the assessments and evaluates the assessments that have been carried out on each of the tools, documenting the areas that have not been covered.

[**Higgins:2005**] Colin A. Higgins, Geoffrey Gray, Pavlos Symeonidis, and Athanasios Tsintsifas, **'Automated assessment and experiences of teaching programming'**, *ACM Journal on Educational Resources in Computing*, 5(3):5, 2005.
Classification: Tools, C, Influential. Citations:0

Annotation

Program development is rarely completed satisfactorily on the first attempt; it is a process of revision and refinement that consists of many levels of development and testing. This paper describes how automated assessment tools can help with this process, summarizes the historical development of the CourseMarker tool, and gives a brief overview of related tools. The paper then gives a detailed overview of the improved CourseMarker architecture and student interface. It also includes a review of the marking system in earlier versions of the system (Ceilidh) and the subsequent changes that have been made to improve it. The latter part of the paper contains an in-depth discussion of the tool's marking mechanism, feedback detail and grading styles, and gives a review of the use of CourseMarker at Nottingham University. This review includes details of course content, exercises and assessment types, and course evolution. Results show that students that became good programmers achieved better marks as a result of CourseMarker's on-demand feedback, and that the use of CourseMarker plus multiple submissions is very beneficial for students and has received very good feedback. The paper concludes with proposals for future work on the system. CourseMarker has been distributed to 50 universities worldwide, and continuing development and improvements should make it more accessible to other institutions. The paper is important because it gives a good overview of the pedagogical considerations needed when using automatic assessment, and demonstrates how the use of CourseMarker has helped students improve their programming skills.

[**Howe:2004**] Emily Howe, Matthew Thornton, and Bruce W. Weide, **'Components-first approaches to CS1/CS2: principles and practice'**, In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 291–295. ACM Press, 2004.
Classification: Curricula, B, Emerging. Citations: 3

Annotation

The paper describes a different approach to teaching object-oriented programming, an approach the authors call "components first". It compares two independently developed approaches to the introductory teaching of programming. The two instructional designs are compared and contrasted. The comparison is used to motivate some underlying principles and to link to the literature in problem solving and cognition. The main message of the paper is that the students should start by being 'clients' by using components that have already been developed. From this perspective, the paper can be seen as an object-oriented version of Pattis's 'read before write' [73] or of the model-based approach [10].

[**Jain:2006**] Jhilmil Jain, II James H. Cross, T. Dean Hendrix, and Larry A. Barowski, **'Experimental evaluation of animated-verifying object viewers for Java'**, In *Proceedings of the 2006 ACM Symposium on Software Visualization*, pages 27–36. ACM Press, 2006.
Classification: Tools, C, Emerging. Citations:3

Annotation

jGRASP is a visualization tool that generates dynamic state-based visualizations of objects and primitive variables in Java. It is unique in that it provides object viewers. The paper describes the tool and reports on formal experiments that investigated the effect of the object viewers on student performance. The findings show that student performance when using jGRASP shows a statistically significant improvement over traditional methods of visual debugging that use break points. The studies and experiments show that students were more productive and were able to detect and correct logical bugs more accurately using the jGRASP viewers.

[**Kay:1996**] David G. Kay, **'Bandwagons considered harmful, or the past as prologue in curriculum change'**, *SIGCSE Bulletin*, 28(4):55–58, 1996.
Classification: curricula, C, Influential. Citations:9

Annotation

This paper warns the developer of a new introductory programming course to beware of the 'dead hand of the past'. It describes some issues that have arisen in previous curriculum transitions, and gives a list of goals for such a course.

[**Kelleher:2005**] Caitlin Kelleher and Randy Pausch, **'Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers'**, *ACM Computing Surveys*, 37(2):83–137, 2005.
Classification: Tools, C, Synthesis. Citations:27

Annotation

This paper presents a taxonomy of languages and programming environments that have been developed to make programming accessible to a larger number of people, and uses this taxonomy to discuss and classify a wide range of tools. The authors first classify the tools into two broad categories - teaching systems and empowering systems. Teaching systems are those that teach programming for its own sake, and empowering systems are those that teach programming in pursuit of another educational goal (e.g., to teach cognitive modelling to psychology students). The taxonomy further classifies these broad categories of tools with respect to their specific pedagogical goals (e.g., teaching systems that simplify the entering of code, and empowering systems that allow students to build models from other domains of knowledge). The latter section of the paper describes the design influences on tools in each category, and the lessons learned from tools that preceded them. It outlines future issues that need to be considered and addressed by designers of these tools in order to make programming more accessible to students. This paper is useful for the teacher as it gives an overview of the available systems, their pedagogic goals and targets, and an overview of the pedagogic issues considered in their design.

[**Kolikant:2005**] Y. Ben-David Kolikant, **'Students' alternative standards for correctness'**, In *Proceedings of the First International Workshop on Computing Education Research*, pages 37–43. ACM Press, 2005.
Classification: Pedagogy, A, Emerging. Citations: 3

Annotation

The author uses the results of a student questionnaire to examine students' view of program correctness. She concludes that students have a pragmatic view of correctness. Specifically, students have a concept of "relative correctness" that was satisfied if a program exhibited reasonable output for many legal inputs.

[**Koelling:2003**] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg, **'The BlueJ system and its pedagogy'**, *Computer Science Education*, 13, Dec 2003.
Classification: Tools, C, Influential. Citations: 94

Annotation

Industrial-strength object-oriented development IDEs are often too complex for novice programmers. BlueJ simplifies the development environment, directly reflecting the development paradigm. It allows visualization of objects and class structures and provides a minimal set of interface components, so that students can focus on concepts rather than on mastering the tool. BlueJ provides a unique mechanism of direct parameterized method calls, which allows the teacher to delay the introduction of text-based interfaces, GUIs or applets until an appropriate point in the course. This paper gives a good overview of the pedagogical reasons for the tool's development and some guidelines for developing assignments. The discussion section focuses on the syntax problem and explains how BlueJ, through interaction with objects first, helps students to deal with concepts rather than tackle syntax as their first task. The paper describes potential problem areas for those using the tool and makes recommendations for the structure of a first programming course. Full evaluation of the tool is left to subsequent papers. The paper has been influential - at the time of its writing there were few student-oriented tools available for teaching an introductory object-oriented programming course.

[**Kuittinen:2004**] M. Kuittinen and J. Sajaniemi, **'Teaching roles of variables in elementary programming courses'**, *SIGCSE Bulletin*, 36:57–61, 2004.
Classification: Pedagogy, A, Emerging. Citations: 13

Annotation

The authors describe stereotypical variable usages in programs. They have found that 10 roles can describe the uses of 99% of all variables in students' introductory programs. They name and give brief descriptions for each of the roles, and show how relationships between the roles can be used as a basis for incremental knowledge construction. Finally, they show how tools can be used to visualize the roles of variables and discuss how variable roles can be used in teaching introductory programming to novices.

[**Linn:1992**] Marcia C. Linn and Michael J. Clancy, **'The case for case studies of programming problems'**, *Communications of the ACM*, 35(3):121–132, 1992.
Classification: Pedagogy, A, Influential. Citations: 89

Annotation

The article argues for the use of case studies in teaching programming. A case study includes a) a statement of the programming problem, b) a description of the process used by an expert to solve the problem, c) a listing of the expert's code, d) study questions, and e) test questions. The reliance on expert solutions links to the schema/plan idea of Soloway [86]. Linn and Clancy note that many programming textbooks tend to emphasize the product (i.e. the program text) but not the programming process itself. They argue that this is reinforced by the typical assignment's focus on running code, rather than on the process that produced it.

[**Long:1998**] Timothy J. Long, Bruce W. Weide, Paolo Bucci, David S. Gibson, Joe Hollingsworth, Murali Sitaraman, and Stephen Edwards, **'Providing intellectual focus to CS1/CS2'**, In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 252–256. ACM Press, 1998.
Classification: Curricula, B, Influential. Citations: 23

Annotation

The authors describe computer science as a discipline that poses deep philosophical questions, and emphasize that it is not just about programming. Based on the definition that computer science is "the study and application of languages and methods for making precise and understandable descriptions of things" they describe several objectives for their CS1/2 course: systems thinking, formal mathematical models, describing design-time behaviour, reasoning about modularity about usage of abstract components and design, specifying and implementing components. They describe how components and relations between these objectives can be made concrete in different paradigms. The paper concludes with a description of the authors' CS1 and CS2 courses.

[**Malmi:2004**] Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppälä, and Panu Silvasti, **'Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2'**, *Informatics in Education*, 3(2):267 – 288, 2004.
Classification: Tools, C, Emerging. Citations: 12

Annotation

The paper describes TRAKLA2 - a web-based learning environment for teaching data structures and algorithms. The system allows the user full control of data structure manipulation through GUI interaction using a method called visual algorithm simulation. This method allows students to manipulate conceptual visualizations of data structures to simulate the working of algorithms. The tool is unique among tools that offer algorithm simulation in that it also facilitates learning management and provides automatic grading and storing of submissions. The learning management facility supports both summative and formative assessment and facilitates randomized input values to the assignments so that each student can get a personal version. It also provides automatic feedback and grading and allows resubmission, which means students can correct their errors in real time. The paper gives a detailed description of the tool and the Matrix framework upon which it is based. It gives an overview of the system architecture and shows how a user can use the system and create new exercises. The authors report on their experiences in using the tool in the classroom, give some positive student evaluations, and point to further references on use of the tool.

[Mannila:2006] L. Mannila, M. Peltomäki, and T. Salakoski, **'What about a simple language? Analyzing the difficulties in learning to program'**, *Computer Science Education*, 16(3):211–227, 2006.
Classification: language choice, A, Emerging. Citations:1

Annotation

The paper evaluates the use of a 'simple' language (Python) as a first language and its potential effect on future learning of more complex languages. The paper begins with motivating the use of a 'simple' language in introductory courses and then presents results from two studies. The first study compares errors found in code constructed by secondary level students after learning programming either using Java or Python, and the other examines how students who learned Python managed the transition to Java.

[McIver:1996] L. McIver and D. Conway, **'Seven deadly sins of introductory programming language design'**, *Proceedings of the 1996 International Conference on Software Engineering Education and Practice*, pages 309–316, 1996.
Classification: language choice, B, Influential. Citations:15

Annotation

The authors present seven undesirable features common to programming languages used in introductory programming courses. Difficulties that originate from these features are illustrated with examples from many different programming languages and paradigms. The authors use the list of weaknesses to derive a list of design principles/criteria that characterize a language suitable for teaching programming to novices.

[Naps:2005] Thomas L. Naps, **'JHAVÉ – Supporting Algorithm Visualization'**, *IEEE Computer Graphics and Applications*, 25(5):49–55, 2005.
Classification: tools, C, Influential. Citations:3

Annotation

The author asserts that many teachers working on algorithm visualization tend to concentrate on the graphics rather than how to employ the graphics for pedagogical advantage. It refers to [52]; this meta-analysis found that visualizations that involved students more actively were more pedagogically effective. The author maintains that algorithm visualization can be used effectively to walk students through algorithms and promote discussion and questioning. He asserts that tools need hooks to engage the student, since viewing graphics is not sufficient to promote effective learning. The paper describes JHAVÉ, a support framework that allows students to step through an algorithm's visual display. It allows the visualization designer to insert 'stop and think' questions in a variety of formats that can pop up at an algorithm's key stages. The paper describes how this is implemented and how the tool can be used by teachers. The remainder of the paper outlines future directions for the research with the goal of making algorithm visualization a good pedagogical tool.

[Naps:2002] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide, **'Exploring the role of visualization and engagement in computer science education'**, In *Working Group Reports from the 2002 Conference on Innovation and Technology in Computer Science Education*, pages 131–152. ACM Press, 2002.
Classification: Tools, C, Synthesis. Citations:76

Annotation

The paper summarizes a rather wide range of research on how visualizations are used. In addition it emphasizes the role played by interaction when visualizations are used in education, and ways learning outcomes could be evaluated in the framework of Bloom's taxonomy. The authors present their own taxonomy of learner engagement with visualization technology and describe three surveys on current practice in using algorithm visualizations in teaching, with a focus on dynamic visualization. They found that dynamic visualization is most frequently used for demonstration, and that in most teaching situations and that students had relatively passive interactions with visualization. The review of experimental studies of visualization effectiveness show that those experiments that had a more active focus on learner engagement had a greater impact on student learning. The paper presents ways for CS teachers to measure the relationship between a learner's form of engagement with a visualization and the types of understanding that are affected by that engagement

[Palumbo:1990] D.B. Palumbo, **'Programming language/problem-solving research: a review of relevant issues'**, *Review of Educational Research*, 60(1):65–89, 1990.
Classification: Pedagogy, C, Synthesis. Citations: 23

Annotation

This is a review and analysis paper that examines the evidence and models associated with using "programming languages as a vehicle for learning problem solving skills". The paper draws on a large body of relevant work in problem solving and cognition.

[Pattis:1993] Richard E. Pattis, **'The "procedures early" approach in CS 1: a heresy'**, In *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education*, pages 122–126. ACM Press, 1993.
Classification: didactics, B, Influential. Citations: 27

Annotation

Pattis describes four eras of Pascal textbooks as a way to capture the development stages of programming courses: pre-Pascal book, early Pascal book, mature Pascal book and Pascal books 'today' (1993). He describes the placement of procedures in books from the different eras, and argues that students needs to use procedures before actually creating them (read/call before write). The same line of reasening is applied in object-first approaches as well (e.g. [82]).

[Powers:2007] Kris Powers, Stacey Ecott, and Leanne M. Hirshfield, **'Through the looking glass: teaching CS0 with Alice'**, In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, pages 213–217. ACM Press, 2007.
Classification: language choice, C, Emerging. Citations:0

Annotation

The paper discusses the 3D graphical programming environment Alice and its use in introductory programming courses. The authors share their experiences in teaching programming using Alice and discuss difficulties their students experienced in the transition from Alice to either of the higher-level languages Java or C++. They suggest that Alice's object model may lead to misconceptions that can be disadvantageous for learning other languages later.

[Ragonis:2005] Noa Ragonis and Mordechai Ben-Ari, **'On understanding the statics and dynamics of object-oriented programs'**, In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 226–230. ACM Press, 2005.
Classification: Tools, C, Synthesis. Citations:7

Annotation

The paper presents a study of the student learning process over two academic years and indicates the main areas where students had difficulty in understanding. It includes detailed examples of questions and code segments. Findings from the study indicate that the topic of program flow needs to be treated explicitly and that the distinction between the BlueJ environment and the execution of a program must be clarified. The authors emphasise that developers of visualizations should combine both static and dynamic aspects and direct the reader to a version of Jeliot that can be used as a plug in to BlueJ for this purpose.

[Reges:2006] Stuart Reges, **'Back to basics in CS1 and CS2'**, In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pages 293–297. ACM Press, 2006.
Classification: language choice, A, Emerging. Citations: 5

Annotation

The paper provides an example of how the introductory programming course can be redesigned by changing the paradigm but not the language. The author discusses his experience from switching from teaching Java as an object-oriented to a procedural language, and provides the main reasons for the change. He argues that teaching "objects first" takes away much of the problem solving required from the students: for example, while an assignment asking students to write a procedural program (given these input data, produce the following output) immediately presents students with problems to solve, students are less challenged by object-oriented assignments that are constrained by specifying the classes and methods to be used.

[Roberts:1993] Eric S. Roberts, **'Using C in CS1: evaluating the Stanford experience'**, In *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education*, pages 117–121. ACM Press, 1993.
Classification: language choice, A, Emerging. Citations:12

Annotation

The paper asserts that a language that has traditionally been viewed as unsuitable for novices can be used in introductory programming. According to the author, the key to writing good programs does not lie in the language used, but in good discipline: "If we teach that discipline, we will produce good programmers. If we do not, the choice of language barely matters" (p.121).

[Robins:2003] A. Robins, J. Rountree, and N. Rountree, **'Learning and teaching programming: a review and discussion'**, *Computer Science Education*, 13(2):137–172, 2003.
Classification: Pedagogy, C, Synthesis. Citations: 59

Annotation

This paper gives an overview of (mainly early) research in psychology of programming. It makes a distinction between novice and expert programmers. It describes research done in the area of cognitive structures (the schemas and plans possessed by expert programmers), the difference between program comprehension (reading a program) and generation (creating a program), and briefly discusses differences between object-orientation and procedural programming. The authors discuss problems with teaching programming to novices.

[Schneider:1978] G. Michael Schneider, **'The introductory programming course in computer science: ten principles'**, In *Papers of the 9th SIGCSE/CSA Technical Symposium on Computer Science Education*, pages 107–114. ACM Press, 1978.
Classification: language choice, D, Influential. Citations:11

Annotation

This paper is not language-specific but gives a more general overview of what should be in a programming language. The author presents ten basic principles, among which he suggests that class time should concentrate primarily on semantics and program characteristics (i.e. how to use the language to produce programs with desirable properties). He argues that the selection of a programming language for educational purposes "should be based on two critical and apparently opposing criteria: richness and simplicity - 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." (p. 110). The paper is a bit dated as it was written before the appearance of Java and C++, but it is still worth reading for its general discussion about features of programming languages, and for its emphasis on the importance of non-coding skills that students should acquire from an introductory programming course (e.g., team working and debugging).

[Soloway:1986] E. Soloway, **'Learning to program = learning to construct mechanisms and explanations'**, *Communications of the ACM*, 29(9):850–858, 1986.
Classification: Pedagogy, C, Influential. Citations: 112

Annotation

This article gives an overview of the theory of programming schemas and plans. Soloway argues that you should annotate a program with the schema used, in order to help novices to grasp the plans applied by experts. Soloway discusses ways to "glue together" plans: abutment, nesting, merging and tailoring. The suggestion to make the plans explicit in the program can be seen as one way to operationalize cognitive apprenticeship. [24]

[Stein:1998] L.A. Stein, **'What we've swept under the rug: radically rethinking CS1'**, *Computer Science Education*, 8(2):118–129, 1998.
Classification: curricula, B, Influential. Citations: 28

Annotation

This paper describes an introductory computing course that uses the fundamental metaphor "computation as interaction". Anyone proposing a new introductory programming course should be aware of the existence of courses using alternative metaphors and paradigms.

[Stein:1999] L.A. Stein, **'Challenging the computational metaphor: implications for how we think'**, *Cybernetics & Systems*, 30(6):473–507, 1999.
Classification: curricula, B, Influential. Citations: 31

Annotation

This paper challenges the formerly dominant metaphor that regards computation as a function converting inputs to outputs. A new introductory course must be built on a fundamental computational metaphor, and this paper will make readers aware of their unspoken assumptions.

[Xinogalos:2006] Stelios Xinogalos, Maya Satratzemi, and Vassilios Dagdilelis, **'An introduction to object-oriented programming with a didactic microworld: objectKarel'**, *Computers and Education*, 47(2):148–171, 2006.
Classification: Tools, C, Emerging. Citations:1

Annotation

ObjectKarel supports program animation and explanatory visualization, and provides a structure editor, a recording facility for recording student submissions and a series of e-lessons and hands-on activities. The paper describes the didactic rationale for the design of the tool. The primary rationale was that novice programmers are often incapable of creating a clear mental model of program execution. Since objectKarel does not hide the process of program execution, it helps prevent students from developing an 'input-output' understanding that is often results from using other programming environments. The tool is part of the Karel family. It provides tracing and step-by-step execution, as well as program animation and visualization. It allows students to focus on the solution of the problem and the acquisition of concepts, rather than on the syntactic details of the programming language, and it encourages them to experiment with their solutions during program execution.