

# Learning to Program: Spreadsheets, Scripting and HCI

Peter Warren,  
Computer Science Discipline  
University of Natal,  
King Edward Ave,  
Pietermaritzburg,  
3209

peterw@cs.unp.ac.za, +27 33 2605646

## Abstract

An examination of some recent programming texts indicates that the complexities of modern programming languages and environments have replaced the study of algorithms and programming principles. Additionally such an approach simply slows the development of the learner in the end. Consequently the paper examines the basic principles of minimalism as espoused by Carroll and the human computer interaction heuristics advanced by Nielsen to argue that scripting languages and spreadsheets provide a better approach. It has been used over the last three years at the University of Natal, Pietermaritzburg. Unfortunately quantitative results are difficult to obtain, but searching, sorting and the use of objects is obtainable after 40 notional study hours for first year students who have never programmed before.

*Keywords:* JavaScript, Human Computer Interaction, Programming, Minimalism, Heuristic Evaluation

## 1 Introduction

A typical modern programming text (Bell & Parr 1998) starts off with objectives something like this:-

This book explains

- how to run Java programs
- how to run Java programs as free standing programs
- how to invoke a Java program from a World Wide Web browser.

This is not Computer Science nor is it about programming. Although the above may appear sensationalist and anecdotal, a survey of ten undergraduate texts taken off the author's shelf shows that the introduction to teaching programming lies with programming language issues, compilation techniques or computer hardware. The introductory programs presented deal with input and output rather than any significant algorithms, although there are notable exceptions. Even the ACM curricula (ACM 2000) seem to drive the discussion from a programming constructs point of view rather than the computational process. There seems to be little attempt to abstract the notion of a

computation from the machine that is going to execute the computation, the language that is going to express it or the management of the development. It is as if a course in mathematics somehow emphasized the internals of Mathematica<sup>©</sup> rather than the axioms and the theorems. This does not mean that there are no exceptions, but in the majority of texts the material is inward looking and computer orientated. The complexities of modern languages and environments are to be blamed for this situation.

There is a need for an approach that allows the Computer Science issues to be in the foreground and the language issues in the background. That in turn requires a simple notation (read programming language) that will enable this. The quest is not new, as it is the ideal that inspired languages such as Pascal and BASIC. Conversely, students do need to have skills in Java and C++ in order to read the textbooks on many other Computer Science related topics, not to mention getting jobs. We face a dilemma as how best to bridge these two worlds.

The simplest approach is to begin with a language that is not tied to the complexities of having to implement the solution on a computer, and thus flow charts or pseudo code are often advocated, for instance (Schneider & Gersting 1999) for a recent example. Even so, at some stage the programmer needs to implement the algorithms on a physical computer, and thus interact with real programming environments and languages. As soon as this is done the principles of human computer interaction (HCI) come into play, and need to be taken cognisance of. This paper presents an approach that is based on HCI principles rather than a focus on the computer, the compiler or the software production process. Driving the analysis from established principles provides a firm basis for the discussion, and establishes this contribution as different from others in the field which tends to be a *post hoc* justification of the actions already taken, or at best a set of criteria designed to justify the position adopted.

In the sections below we will review the major issues in JavaScript as they apply to the teaching of programming. We will then look at minimalism (Carroll 1990) and Nielsen's heuristics (Nielsen 1993). The issue is what is needed to get started, and *not* the final outcome. The point is often misunderstood in informal discussions. The assumption is that, in the end, the learners need to be able program in one of the C-family of languages such as Java, C, C++ or C#. This is a critical part of the argument, as there is no point in an introductory language if it does not ease that transition, at least in our case.

## 2 Related work

There is a huge literature advocating one or other programming language as suitable for teaching beginners, and reviewing it is quite beyond the scope of this contribution. It suffices to note that even those who advocate using Java or C++, comment on the difficulties of using such complex environments to teach programming. Hong (1998) goes even further by insisting that students have a pre-course which discusses functions and arrays. Whether it can then be claimed that Java is being used as an introductory programming language is a moot point.

The literature on the use of scripting languages (Barron 1999) for teaching programming is much smaller. The important point made in all the contributions, is that scripting languages tend to be simpler than the “system programming languages” (Ousterhout 1998) and thus easier for both student and faculty to learn. The issue of dynamic typing appears to be the most controversial issue raised in opposition, but this is largely due to ignorance because dynamic typing can be just as safe as static typing (Ousterhout 1998). Most of the issues are explored by Zelle (2003) who also comments on the problem of the distance between the teaching language and the final target language, as well as the concern that students have for the relevance of the skills learned. These are important considerations. Despite the hope that students can transfer programming skills from one language to another, anecdotal experiences and the literature do not confirm this (Scholtz & Wiedenbeck 1989). Scheme, particularly, founders on both relevance and transferability.

There appear to be considerably fewer papers on the use of JavaScript (Warren 2001, Wilkens 2002, Zachary & Jensen 2003, Popyack, Shokoufandeh & Zoski 2000, Mercuri, Herrmann & Popyack 1998, Ward & Smith 1998, Brady, McDowell & Schultz 2002, Reed 2001). These reiterate the advantages of scripting languages, but further emphasize the real nature of JavaScript and its syntactic similarity to Java/C++. Most authors regard JavaScript as a simple language however those who combine it with HTML are not quite so certain (Zachary & Jensen 2003). There is some reservation about the generality of JavaScript (Ward & Smith 1998) but this is in part due to a misunderstanding of the language. Authors also comment that the association of JavaScript with the Web is an additional attraction (Gurwitz 1998). JavaScript has primarily been used to teach novices and non computer science majors but it has also been used to teach more advanced topics such as data structures and objects (Warren 2001, Wilkens 2002).

However, the JavaScript teaching literature is essentially descriptive of courses that have been given, and mostly reporting positive experiences with the language. These have to be balanced against an even larger number of papers extolling the virtues of C++ or Java as an initial programming language. Without a set of principles it is difficult to judge the various arguments. This work is designed to close the gap.

Outside the literature on teaching using one of the C-family of programming languages lies the vast literature on the psychology of programming, eg (Hoc, Green, Samurcay & Gilmore 1990, Winslow 1996). This literature is consistent with this paper and provides many useful insights. Particular interesting are the studies on the “transfer of competence” where, at risk of oversimplification, the greater the distance between the learned and required activity the greater the problem of applying it (Hoc *et al.* page 75). Similar point was made over 30 years ago by Weinberg in his pioneering book “The Psychology of Computer

Programming” where he discusses the issue of Tradition and Innovation. Another important point made is that simplicity, in its self, is not advantageous but rather the problem is to rid the systems of unnecessary complexity, Hoc *et al.* page 24.

A more radical approach is taken by authors who advocate the use of visual programming environments. The recent work (Pane 2002, Pane, Myers & Miller 2002) is interesting as it uses HCI techniques to design the programming language. Although this addresses the problem of the transition from no programming to programming it suffers from the same problem as Scheme namely the “transfers of competence” difficulty. Noteworthy is that he sees Nielsen’s (1993) heuristics “Consistency and Standards” as relating to internal consistency and consistency with how children think. However, this is not the way it was originally conceived which included consistency with other systems, in this case, programming languages. Neither his HANDS system widely available. These studies appear to be of more long term interest than immediately applicable.

Neither the psychological nor the visual programming literature seems to address the phenomenon of the rise in popularity of scripting languages, and JavaScript in particular. Hopefully this paper will address this issue and why it is timely for educators to take note of these trends.

## 3 JavaScript

JavaScript (Flanagan 1998) is not Java; the name is just a marketing ploy. Nevertheless the language was designed to make the transition to the C-family as easy as possible, and in order to help achieve this, the control structures and operators are pure C. This includes the use of = for assignment, and probably worse, + for string concatenation. Consequently it is relatively simple to take a function written in C++ and convert it to JavaScript.

Unlike the C-family however, JavaScript is a dynamically typed language. This does not mean that there are no types, but rather that it is the contents, and not the declaration, that determines the type. The language supports very few types viz number, string, boolean and objects. Types are completely uniform, a function can be an array element and any type can be returned as a function result, including a function. There is no character type; if one wants to look at a single character then a string of length one can be used. The type system looks a lot like that of a spreadsheet, and the range of types are both necessary and sufficient for any but the most specialized applications; novices do not encounter these.

Variables do not have to be declared, which is an issue that can create considerable heat. In conventional languages variables have to be declared, but errors associated with uninitialized variables are not trapped. Conversely, in JavaScript variables need not be declared, but they do have to be initialized. This detects many spelling errors as well as the uninitialized variable error. Unfortunately failure to declare variables inside a function makes them global and engendering a habit of declaration is probably worth while.

Like Java, in JavaScript primitive types are addressed by value while composite objects are addressed by reference, and there are no explicit reference parameters. This is a complexity one would wish to avoid. The solution is to pass all information back to the caller via the function result rather than through parameters. Simple object creation in JavaScript makes this technique viable in the language in a way in which it is not in the C-family. Nev-

ertheless, more experienced C and Pascal programmers in a novice class, may have to be persuaded to try a more functional approach to programming than they are used to.

Another objective of JavaScript was to make object creation easy but not mandatory. Objects can be created without requiring a prior declaration of a class. For example an object representing a point is created as follows, `{x:10, y:3*q}` and additional properties and methods can be added at any stage without having to get involved in inheritance.

Here is an example of one of the first functions presented to students. "Find a telephone number in a telephone directory. This one is a bit hard at present but we will give it anyway"

```
function telephone(person, book){
  for (var i=0; i < book.length; i++)
    if (book[i].name == person)
      return book[i].number;}

```

Surprisingly the function does not contain an error; JavaScript returns `undefined` if nothing is returned explicitly, but discussion the "not found" appeared to be premature. Here is an example of its use. Note how the object and array literals reduce the syntactic overhead.

```
me = telephone("Peter",
  [{name:"Peter", number:1},
  {name:"Paul", number:2},
  {name:"Mary", number:3}]);

```

JavaScript is not only suitable for the teaching of elementary programming but more advanced data structures such as linked lists and trees are equally well handled, as are object oriented techniques. It supports, but does not require classes, however these do look quite different from C++ and Java. Objects are created by a constructor functions, e.g. for a `Person` object:

```
function Person(name, address){
  this.name = name;
  this.address = address;}

```

Inheritance is also supported but looks quite different from the C++ model. Consider a `Student` that descends from `Person` by also having a `mark`. Each function has a prototype, and if a property is not found in the object itself, a search is made in the constructor's prototype, and if not in the prototype in the prototype's prototype, etc. Thus the prototype of the `Student` is made to be an instance of a `Person` object in order to create this chain.

```
function Student(name, address, mark){
  this.mark = mark;
  this.Person(name, address);}
Student.prototype = new Person;
Student.prototype.Person = Person;

```

Note how the constructor of `Person` can be used inside `Student` to initialize the `name` and `address` properties. We could have made the `Person` constructor as part of the `Person` prototype instead of the `Student` prototype, by writing `Person.prototype.Person = Person` instead of `Student.prototype.Person = Person`. The constructor for `Student` remains unchanged, however, as the `Person` constructor function will simply be found further up the inheritance chain.

When a non-existent property is used as an *lvalue* the property is created, while if used as a *rvalue* it returns `undefined`; but either way it is not an error. This allows a very simple binary tree constructor. Noting that a binary tree is either empty, or it is a node consisting of a value along with a left and right subtree. An empty tree is thus created with an empty constructor:-

```
function BST(){
  BST.prototype.empty = function ()
    {return ! this.left;}

```

However do not be misled that this is a nonexistent object, it has a prototype and a known constructor. We have already given it one method, the `empty()` method. Inserting an object into the binary search tree is easy enough:-

```
BST.prototype.insert = function (value){
  if (this.empty()){
    this.left = new BST;
    this.right = new BST;
    this.value = value;}
  else{
    if (value < this.value)
      return this.left.insert(value);
    if (value > this.value)
      return this.right.insert(value);}
}

```

The function is polymorphic, the `value` only needs to be able to respond to a `valueOf()` method in order to make comparisons. Contrast this with C++ where a whole range of operators need to be overloaded.

Exception processing is available through the `try/catch` mechanism, which means we do not need to follow the weak error handling technique of writing an error message. The syntactic overhead is considerably less than in C++, nor is it mandatory as in Java.

One point must, however, be emphasized. JavaScript consists of two parts, the core language and the API for the Document Object Model (DOM) which is essentially the programmer's interface to HTML. Mixing up the teaching of programming with the teaching of anything, especially HTML and the DOM, is a recipe for failure and we have experienced those problems in Pietermaritzburg. In the theoretical part of the course all reference to input and output are banished while these are taken up in the laboratories. Unfortunately all texts I have examined see the teaching of JavaScript as the objective and not the teaching of programming, a typical example being that by Deitel and Deitel (2002) . As we have observed, this situation is little different from other languages.

One further tactical point needs to be made. JavaScript is not a teaching language; it is a real one that is widely used and probably has to be learned by the student anyway (ACM 2000). This is important to motivate the students and the staff. Nor is it a toy language; it is a modern language supporting object oriented concepts but not enforcing them. This allows objects to be introduced as they are needed to simplify programming and not artificially to complicate the issue as is prevalent in many Java texts. It is available on any computer that supports a browser. Finally the dynamic typing is different from static typing which adds a new dimension to those who can already program.

#### 4 Minimalism

Carroll (Carroll & Mack 1984) noted that self instruction manuals for software packages, and word processors in particular, were ineffective. Carroll makes a number of points which are germane to the learning of programming.

- Complexity hurts because it inhibits exploration and adds to the burden of learning. It takes twice as many tokens to program the sieve of Eratosthenes in Java as it does in JavaScript (Warren

2001). Novices are likely to be totally confused by `public static void main (String[] argv)` (Hong 1998). The beginner pays for this by learning half as many real Computer Science concepts. Inconsistency is a form of complexity and hinders in integrating the bits and pieces. Complexity not only refers to the language but the help and the environment – the less said about many of the C-family systems the better.

The advanced student may need the concepts that lead to the additional syntactic clutter, and if so, that is the time it should be introduced.

- The Zen aphorism that “less is more” is central to minimalism. It says the novice should not be confused with unnecessary complexity over and above what is needed for the task; Laurel (1993) refers to it as “gratuitous complexity”. The fact that they need to know the concepts eventually is no reason for explaining it to them at the outset. Novices do not need 12 different kinds of numbers where one will do, or to know only integers can be used to index arrays because it’s really an index register. They do not need to understand the difference between reference and value types to write even quite complex algorithms nor do they have to be involved in their own memory management. They do not even need to understand the compilation process in order to understand the operation of an algorithm.
- Carroll (1990) proposes a “staged” sequence with the initial system trading function for simplicity; this is sometimes called the *training wheels* approach. The idea is to enable only a *subset* of the total system to prevent novices getting tangled in complexity and to encourage exploration. In order to teach a complex language a suitable subset needs to be chosen rather than something different. The teaching of flow charts (Ramsey, Atwood & Doren 1983, Waddell & Cross 1988) or LISP derivatives (Sanders & Mueller 2000) is not a good way to teach C++ or Java but JavaScript is.
- Learner’s thinking is routed in the task domain and not in the system domain. Learning takes place by addressing the tasks, in this case the tasks of learning about algorithms, and not the learning about the influences of 1970’s architectures on programming languages. Excuses about having to understand what really happens inside the computer must slow the learners down and deflect them from their tasks. Bottom up approaches are excluded by this principle.

Minimalist principles apply not only to the programming languages but to the integrated programming environments. If it contains complexity that is not needed to complete the task in hand the novice will be slowed. I am not aware of any studies in this regard with programming environments although there have been many studies of the difficulties of using word processors that are too complex (e.g. Carroll, 1990) and the principles are the same.

To conclude, minimalist principles predict that students are going to have difficulties in learning to program, especially with texts that are driven from the language and machine perspective rather than from the algorithm point of view. They also predict that the more complex the environment and the language the greater the difficulties or the fewer real computer science issues will be tackled. This is exactly what is observed.

## 5 Nielsen’s Heuristics

Nielsen’s (1993) usability heuristics are presented below. The heuristic is in boldface, Nielsen’s commentary is in italics while our own comments are in roman. These heuristics deal with user interfaces, but they apply to programming when programmers interact with a computer, and provide considerable insight into the difficulties. Their advantage is that they have been experimentally verified to be the major causes of problems with user interfaces which raises the discussion above that of “my opinion versus your opinion”.

1. **Visibility of system status** *The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.*

Programming languages fail this most basic usability principle because variables, which are not visible, hold the state of the computation and evolve over time. Although debuggers help, they also present a new tool and thus a new complexity, and in view of this it is not surprising that we have had limited success with their use. The solution is not to use programming languages to teach concepts that can better be taught by other techniques where the system status *is* visible. In a spreadsheet the whole computation is displayed, and there is no time evolution, and therefore the computation will be much easier to comprehend.

Spreadsheets are used to introduce expressions, function use, data types and recurrence relations. Our experiences are that these are major stumbling blocks for beginners. Recurrence relations are the very basis of the understanding of the loop and without understanding them it is doubtful if the looping construct can be understood at all. These issues need to be explored in an environment where the system state *is* visible.

2. **Match between system and the real world** *The system should speak the users’ language, rather than system-oriented terms.*

This issue is central to minimalism and is a tenet of the constructivist educational principles. The learner brings an understanding of the world and the tasks to the learning experience. The system should support that view. The novice understands that the integers are a subset of the reals; it is only programming languages that treat them as disjoint. Novices have never even thought about the finite size of numbers in a computer. It is quite likely that their calculators delivered sufficient precision for their needs. Double precision numbers based in the IEEE standard are sufficiently precise not to bother the novice about finite word sizes. Similarly, sequences are infinite in mathematics and there is no reason to know about the finite size of arrays. There is no need to understand that arrays are really mappings to memory; in the real world sequences are just an ordered collection of objects, any objects. Most controversially the concept of a type is not a natural (Taivalsaari 1999) one nor is object orientation (Martin 1998, Taivalsaari 1999). The concept of the contents determining the type is much more natural and objects, which are an advanced, rather than different, construct can wait. Some regard C as a simple language, but it fails the match with the real world in many ways.

3. **User control and freedom** The original heuristic relates to the ability of users to navigate

freely in the system. The analogy with programming languages would be a dynamically typed system which allows programmers to store any type of 'object' in any variable and where every object is a first class object. It also requires a 'safe' environment where the student can explore the language without fear of crashing the whole computer system or running into lots of fine print caveats. JavaScript provides this, C certainly does not while Java itself is somewhere in the middle.

4. **Consistency and standards** *Consistency is one of the most basic usability principles. Users should not have to wonder whether different words, situations, or actions mean the same thing.*

There are two forms of consistency, the system should be consistent with the external world and internally consistent. The external world of the novice is one of mathematics and the C-family. The arguments here are the same as the arguments for "training wheels" and will not be repeated. It also means that the language should be internally consistent. C++ certainly is not. Java and C# are much better but no better than JavaScript.

5. **Error prevention** *Even better than good error messages is a design which prevents a problem from occurring in the first place.*

The more error conditions that flow from system or historical considerations the harder the system will be to learn. Unfortunately the C-family is full of such constructs. Conversely, JavaScript removes a number of errors; division by zero is not an error, undefined values are not errors and arrays are dynamic so there are no subscript bounds. Conversion between types is sensible and automatic. For small programs this works extremely well.

6. **Recognition rather than recall** Programming languages in general do not follow this principle which is one of the reasons they are hard to learn. I am not going to offer help on this point.
7. **Flexibility and efficiency of use** The objective is to automate activities not germane to the algorithms; memory management is a typical case. It is very easy to write generic functions in JavaScript.
8. **Aesthetic and minimalist design** *Every extra unit of information competes with the relevant units of information and diminishes their relative visibility.*

This applies to the environment and the language. The languages and environments of C++, Java, C# and such like are not minimalist, containing many features that are of no interest to the beginner. These features will only confuse.

The JavaScript programmer has only one kind of number, has only one kind of string, has only one kind of array, has only dynamic binding, does not have to manage memory, has a single mechanism to return function results. Most algorithms do not need more than this; the algorithms that novices write never require more than this. It is so that JavaScript inherits many constructs from C that are less than ideal but there has to be a balance with the consistency and standards issue above. There are also many things that could be left out, like the switch statement and the while,

without degrading the language in any way; it is a pity they were not.

It is possible to find free editors which provide a simple IDE for JavaScript, our own favourite is Arachnophilia, (Lutus 2000), which has an integrated browser capability which provides a one click compile and go. It is conceded that some excellent free tools also exist for C++ and Java. Commercial systems like Visual C++ are not minimalist and contain too many features which must confuse the novice.

9. **Help users recognize, diagnose, and recover from errors** We agree, but the solutions advocated do not achieve this. The JavaScript error messages are just as perplexing as those in C++ and Java and the runtime diagnostics just as hard to interpret. More is the pity, and the more difficult it is for students to learn to program.

10. **Help and documentation** If the instructor cannot understand the help system it is highly unlikely that the student will be able to. Unfortunately the help systems of complex languages are too complex for even instructors to understand. This contrasts quite strongly with JavaScript where the core language is small (Netscape 1998) and the documentation is better understood by a novice. Microsoft have produced a set of documentation for JScript which is downloadable from their site. There is also an excellent summary of the Core JavaScript from Netscape (1998).

When it comes to academic material there is alas nothing that avoids the same pitfalls that most of the C++ and Java material already falls into. The situation is no worse than these languages as material of the same type is available.

## 6 Discussion and Conclusions

The languages like C++, C#, Java, Delphi and VB embedded in their typical environments are large complex systems that fail just about every usability principle. In this paper it has been argued that teaching programming by first using spreadsheets and later JavaScript along with an HTML editor with integrated browser capability matches the principles much more closely. The principles follow from Minimalist concepts and Nielsen's usability heuristics. These also predict that languages that are not essentially a subset of C++ / Java are less likely to be successful if the eventual aim is that learners master one on the C-family of languages. In this context it is easy to understand why flow charts are not effective although this has been appreciated for some time (Ramsey et al. 1983). The approach has been tested with an introductory programming module for business students as well as introductory programming section for largely science students. The results show that instructors can get to deeper algorithms faster than with the traditional route. It has also shown that the transition to C++ is greatly eased. The informal experience is that the types cause the biggest difficulty.

However, we are not advocating the use of a scripting language for teaching advanced programming courses. We are not asserting that they are useful for constructing large programmes. Programmers need to understand the advanced techniques; we are just asserting that they are harmful at the outset. This point of view is fully supported by HCI principles (Nielsen 1993) and minimalist educational

principles (Carroll 1990). It is difficult to understand why in informal discussion and current practise the issue is so widely ignored.

## References

- ACM (2000), Computing curricula 2001, Technical report, Association for Computing Machinery.
- Barron, D. (1999), *The World of Scripting Languages*, Wiley, Chichester, Sussex.
- Bell, D. & Parr, M. (1998), *Java for Students*, P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA.
- Brady, A., McDowell, R. C. & Schultz, K. (2002), 'JavaScript programming basics: a laboratory series for beginning programmers', *Journal on Educational Resources in Computing (JERIC)* **2**(2), 1–1.
- Carroll, J. M. (1990), *The Nurnberg funnel : designing minimalist instruction for practical computer skill*, M.I.T. Press, Cambridge, Mass.
- Carroll, J. M. & Mack, R. L. (1984), Learning to use word processors: By doing, by thinking, and by knowing, in J. C. Thomas & M. L. Schneider, eds, 'Human Factors in Computer Systems', Ablex Pub. Corp., Norwood, New Jersey, chapter 2, pp. 13–51.
- Deitel, H. M., Deitel, P. J. & Nieto, T. R. (2002), *Internet and World Wide Web: how to program*, second edn, Prentice-Hall, Upper Saddle River, NJ 07458, USA.
- Flanagan, D. (1998), *JavaScript : The Definitive Guide*, O Reilly and Associates, California.
- Gurwitz, C. (1998), The internet as a motivating theme in a math/computer core course for nonmajors, in 'Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education', ACM Press, pp. 68–72.
- Hoc, J. M., Green, T. R. G., Samurcay, R. & Gilmore, D. J. (1990), *Psychology of Programming*, Computers and People Series, Academic Press, San Diego, Calif.
- Hong, J. (1998), 'The use of Java as an introductory programming language', *ACM Crossroads* **4**(4).
- Laurel, B. (1993), *Computers as Theatre*, Addison-Wesley Publishing Company, Reading, Massachusetts.
- Lutus, P. (2000), Arachnophilia. Port Hadlock, WA. **URL:** [www.arachnoid.com](http://www.arachnoid.com)
- Martin, P. (1998), 'Java, the good, the bad and the ugly', *ACM SIGPLAN Notices* **33**(4), 34–39.
- Mercuri, R., Herrmann, N. & Popyack, J. (1998), Using HTML and JavaScript in introductory programming courses, in 'Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education', ACM Press, pp. 176–180.
- Netscape (1998), *Core JavaScript Guide*, Netscape Communications Corporation, Mountain View, CA.
- Nielsen, J. (1993), *Usability Engineering*, Academic Press, Boston, MA.
- Ousterhout, J. K. (1998), 'Scripting: Higher level programming for the 21st century', *IEEE Computer* **31**(3), 23–30.
- Pane, J. (2002), A Programming System for Children that is Designed for Usability, PhD, Carnegie Mellon University, Computer Science Department, Pittsburgh.
- Pane, J. F., Myers, B. A. & Miller, L. B. (2002), Using HCI techniques to design a more usable programming system, in 'IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)', Arlington, Virginia, USA, p. 198.
- Popyack, J. L., Shokoufandeh, A. & Zoski, P. (2000), 'Software design and implementation in the introductory CS course: JavaScript and virtual pests', *The Journal of Computing in Small Colleges* **15**(5), 166–177.
- Ramsey, H. R., Atwood, M. E. & Doren, J. R. V. (1983), 'Flowcharts versus program design languages: an experimental comparison', *Communications of the ACM* **26**(6), 445–449.
- Reed, D. (2001), Rethinking CS0 with JavaScript, in 'Proceeding of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE-01)', Vol. 33.1 of *ACM Sigcse Bulletin*, ACM Press, New York, pp. 100–104.
- Sanders, I. & Mueller, C. (2000), A fundamentals-based curriculum for first year computer science, in 'Proceedings of the 31st SIGCSE technical symposium Austin, TX, USA', pp. 227–231.
- Schneider, G. M. & Gersting, J. (1999), *Invitation to Computer Science*, Brooks/Cole, Pacific Grove, CA, USA. Contributing author: Sara Baase.
- Scholtz, J. & Wiedenbeck, S. (1989), Learning a new programming language, in 'Proceedings of the Third International Conference on Human-Computer Interaction', Vol. 2 of *Designing and Using Human-Computer Interfaces and Knowledge Based Systems; Learning*, pp. 152–159.
- Taivalsaari, A. (1999), Classes and prototypes. some philosophical and historical observations, in 'Prototype based programming', Springer-Verlag, Singapore.
- Waddel, K. C. & Cross, J. H. (1988), Survey of empirical studies of graphical representations for algorithms, in 'Proceedings of the 1988 ACM sixteenth annual conference on Computer science', ACM Press, p. 696.
- Ward, R. & Smith, M. (1998), JavaScript as a first programming language for multimedia students, in 'Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education', ACM Press, pp. 249–253.
- Warren, P. (2001), 'Teaching programming using JavaScript', *The Journal of Computing in Small Colleges* **17**(2), 192.
- Weinberg, G. M. (1971), *The Psychology of Computer Programming*, Van Nostrand-Reinhold, New York, NY. ISBN 0-442-20764-6 (PPBK).
- Wilkens, L. (2002), 'Objects with prototype-based mechanisms', *The Journal of Computing in Small Colleges* **17**(3), 131–140.

Winslow (1996), 'Programming pedagogy – a psychological overview', *SIGCSE* **28**(3), 17–25.

Zachary, J. L. & Jensen, P. A. (2003), Exploiting value-added content in an online course: introducing programming concepts via HTML and JavaScript, *in* 'Proceedings of the 34th SIGCSE technical symposium on Computer Science education', ACM Press, pp. 396–400.

Zelle, J. (2003), Python as a first language. [http://mcsp.wartburg.edu / zelle /python /python-first.html](http://mcsp.wartburg.edu/~zelle/python/python-first.html), accessed August 2003.