The impact of Memory Transfer Language (MTL) on reducing misconceptions in teaching programming to novices

Authors:
Leonard J. Mselle[1]
Hashim Twaakyondo[2]

Affiliations:
[1]Department of Computer Science, University of Dodoma, Tanzania

[2]Department of Computer Science, University of Dar es Salaam, Tanzania

Correspondence:
Leonard Mselle

Email:
mselel@yahoo.com

Postal address:
PO Box 490, Dodoma, Tanzania

Dates:
Received: 2011
Accepted: 23 Mar. 2011
Published: 28 June 2012

How to cite this article:
Mselle LM, Twaakyondo H. The impact of Memory Transfer Language (MTL) on reducing misconceptions in teaching programming to novices. Int J Machine Learn Appl. 2012;1(1), Art. #3, 6 pages. http://dx.doi.org/10.4102/ijmla.v1i1.3

Despite the fact that programming is at the heart of computer science, it is argued that even at its simplest level it is a difficult subject to teach and learn. For any new learner programming concepts are abstract and confusing. As teaching programming continues to be a daunting task, this article revisits common challenges inherent in teaching computer programming to novices. Further, Memory Transfer Language (MTL) as used to teach programming is introduced and demonstrated. Different kinds of misconceptions in programming and their associated bugs are analysed. An experiment using MTL to teach programming was carried out, using error-counts in examination scripts from two groups of students, one instructed using MTL and the other through the conventional approach. Results indicated a highly significant statistical difference ($p = 0$) between the two groups, showing that MTL can help novices avoid common programming misconceptions and reduce the errors they make. This shows that if programming is taught using MTL, comprehension is enhanced.

Introduction

A novice programmer is a person who is learning programming for the first time. The basic characteristics of novice programmers include inability to design proper algorithms to solve given tasks, and inability to master the syntax and semantics of programming languages. A distinctive characteristic of novice programmers is that they see each line of code in isolation from others. Novices find the learning environment (editors and debuggers) to be unfriendly. This mixture of challenges makes programming intrinsically a difficult subject to learn.[1]

Since the 1970s computer science researchers have tried to investigate and suggest various solutions to this problem. Samurcay[2] contends that programming is a complex domain of knowledge and practice, corresponding both to the scientific field and professional practice field. He argues that learning programming means acquisition of specific programming concepts mediated by a technological tool, which necessitates construction of higher-level representations and conceptual invariants. He concludes that a definite solution to this challenge is not yet available. A similar point of view is expressed by other authors.[3,4,5,6,7,8] Although there are other researchers, like Wilson and Moffat[9], who conclude that programming is not difficult, all researchers propose further investigation of how to teach programming more effectively.

Employment of models in teaching programming

Employment of models to teach computer programming has been advocated by numerous researchers.[1,4,10] Du Boulay, O'Shea and Monk[4] posit that concrete models can have a strong effect on the encoding and use of new technical information by novices. Allowing novices to 'see the works' allows them to encode information in a more coherent and useful way. Testing their model called LOGO, Du Boulay et al.[4] found that the model enabled the user(s) to develop intuition about what goes on inside the computer for each line of code.

Mayer[10] contends that models not only help learning programming but also aid in acquisition of other aspects of technological knowledge. He argues that when appropriate models are used, the learner seems able to assimilate each new code statement to their model of the computer system. Some models, such as Discover by Ramadhan[1], have been reported to provide good results in this direction.

The recent trend has been to combine models with animation tools to teach programming. Along with concrete models, code animators have been reported to be effective tools in teaching programming.[8,10,11,12] Closely related to animation, automated flowcharts (which combine traditional flowcharts with the power of animation) have been reported to produce positive results in enhancing programming comprehension.[13] However, all current program animators are machine-driven. Apart from Memory Transfer Language (MTL), there is no evidence of an absolutely learner-driven animator.[14]

Problem statement

Although the use of models and animation has yielded positive results, so far there is no one universally accepted approach that has entirely solved the problem of learning and teaching programming. Current program animators, such as Jeliot 3, BlueJ and Raptor, are machine-driven. This deprives the learner of the sense of 'I am working the machine', which is important for a novice programmer to build confidence.[4] A learner-driven visualiser could be a solution to this problem. Teaching and learning programming continues to be a daunting task for both learners and instructors.[3] There is still much room for improvement of current tools, methods and approaches to evolve a learner-driven visualisation tool.

Objective of the study

MTL is a learner-driven visualisation tool. An experiment was carried out to evaluate the impact of MTL in reducing common misconceptions in programming. We were interested in determining whether using MTL in teaching would result in a reduction in common misconceptions and other common errors in programming. This article revisits the misconceptions and errors in programming as presented by Du Boulay[5], Perkins et al.[15] and Alain.[16] We used these works to evolve a protocol to count the errors committed by novices. MTL was used to teach programming, and in order to determine its impact the numbers of errors committed by novices instructed using MTL were compared with those made by novices instructed using the conventional approach.

A brief description of Memory Transfer Language

Memory Transfer Language (MTL) is a language or device used by programmers to describe the impact of code-line on computer memory (RAM). MTL is a modified version of Register Transfer Language (RTL) and trace tables.[17] Whilst RTL describes the program behaviour at machine-language level, MTL is used to describe program behaviour in high-level language.

The MTL framework is based on the assumption that any source code written in high-level language can be humanly interpreted and/or compiled by reflecting and/or drawing the RAM status for each line of code. The semantics of each RAM diagram produced in MTL is both human- and machine-interpretable. MTL provides a common interpretation of the source code between the machine and the programmer, reducing the ambiguity and misconceptions which abound in programming. MTL is a programming language for interpreting the source code machine-wise. It is a learner-driven visualiser comprising macro-steps (statements) as non-terminals and RAM status as terminals.[14,17] The general framework for MTL is illustrated in Figure 1.

Employment of MTL for program interpretation is fully covered by Mselle.[14] The application of MTL is demonstrated in Figure 2, Figure 3 and Figure 4, where MTL is used to describe variable declaration, data inputting, data operation, loops and functions respectively.

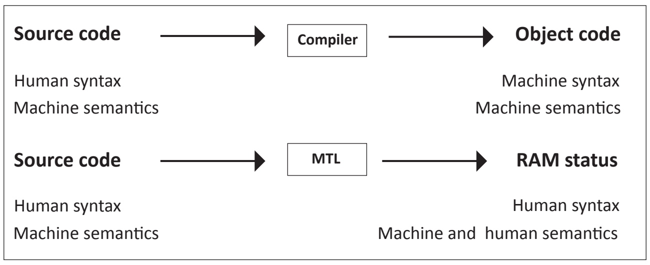
As illustrated in these figures, high-level language statements such as *int sum; sum=0; and i=i+1;* are interpreted into respective RAM status, pictorially reflecting what happens inside the machine RAM as a statement is executed. Equally, the concepts of variable initialisation, looping, function calls, *return (),* and parameter passing are visibly demonstrated by MTL. Mselle[14] has shown that MTL can be used to cover all basic programming concepts, including file handling, pointers and stacks.

The experiment

An experiment was carried out amongst students of Kigali Institute of Science and Technology who were studying programming for the first time, in order to test the effectiveness of MTL in facilitating close-tracking, fixing bugs, providing feedback and reducing misconceptions.

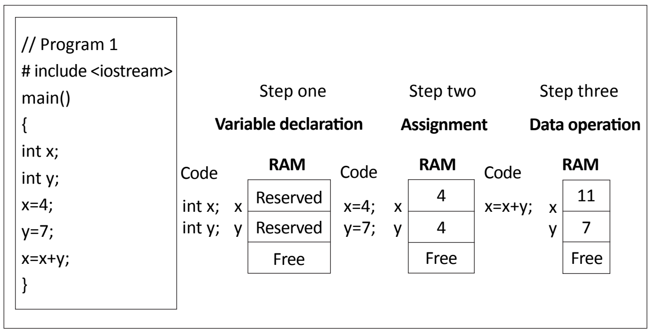
Method

A sample consisting of 34 examination scripts, drawn from a population of 543 past examinations collected for five years, constituted the control group. Twenty eight scripts drawn from an examination completed by a group of students who volunteered to pursue the course using MTL constituted the experiment sample. The experiment group comprised first-year undergraduates majoring in Architecture and Environmental Sciences. All students involved in the



Source: Authors' own data
MTL; Memory Transfer Language; RAM, computer memory.

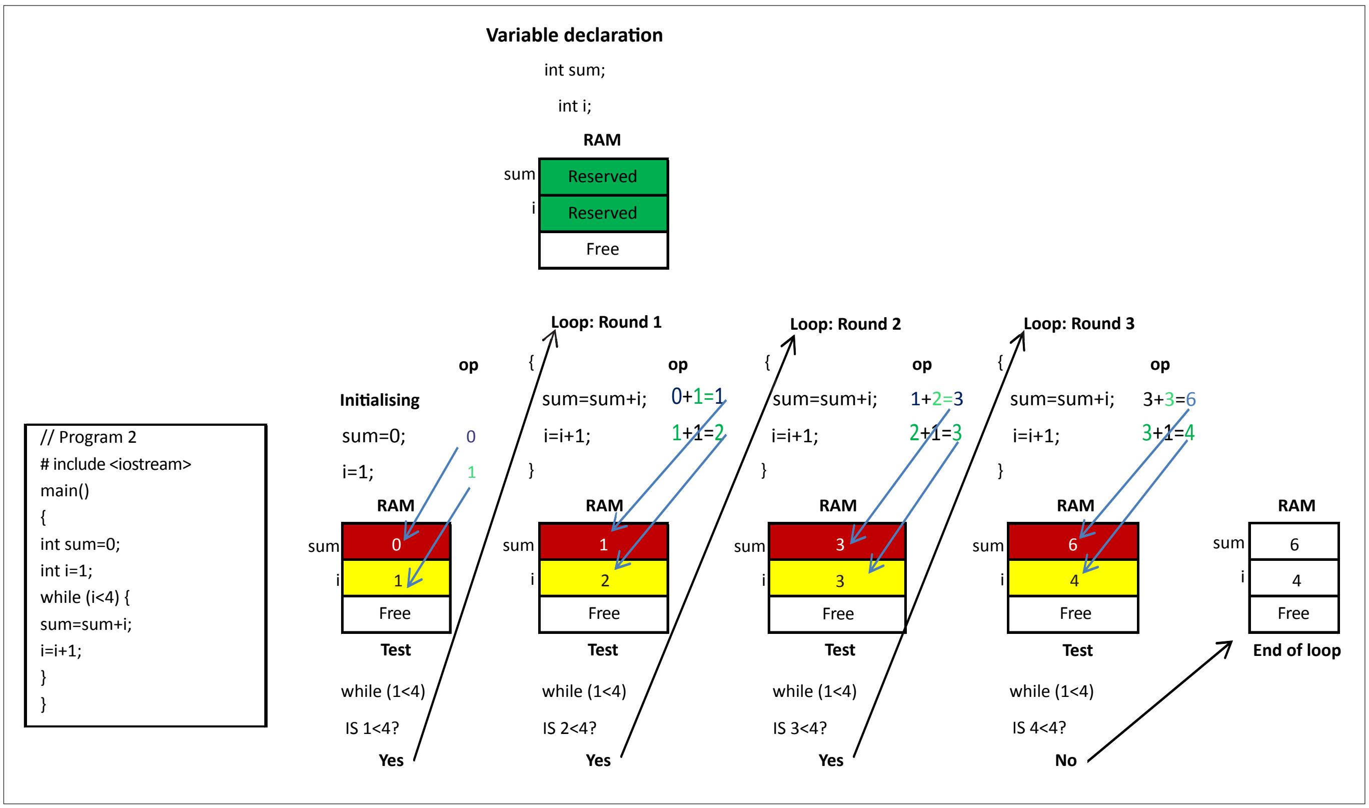
FIGURE 1: The general framework of Memory Transfer Language.



```
// Program 1
# include <iostream>
main()
{
int x;
int y;
x=4;
y=7;
x=x+y;
}
```

Source: Authors' own data
RAM, computer memory.

FIGURE 2: Demonstration of variable declaration, data input and operation (Program 1).



```
// Program 2
# include <iostream>
main()
{
int sum=0;
int i=1;
while (i<4) {
sum=sum+i;
i=i+1;
}
}
```

Source: Authors' own data
RAM, computer memory.

FIGURE 3: Loop interpretation (Program 2).

experiment were novices studying programming for the first time. The course title was 'Introduction to Programming in C' and the course syllabus covered variable declaration and types of variables, constants, data inputting, data manipulation, data outputting, flow of control, functions, arrays, strings, pointers and file handling. The duration of the course was 52 hours, with 26 hours used for lectures and 26 for laboratory sessions. The lead lecturer for the experiment group had five years' experience in teaching programming in C.

Materials used

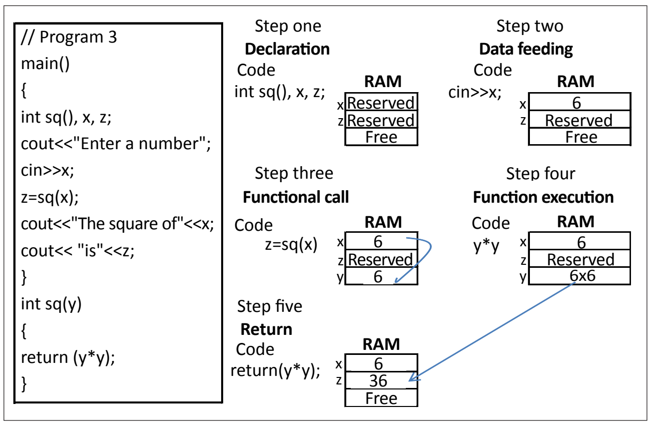
The materials used included examination scripts, an errors protocol and a programming manual in C. The programming manual covers introductory programming, which includes variables and variable declaration, data inputting, processing and outputting. Other topics are covered individually, including flow of control (bifurcation and looping), arrays, strings, functions, files handling and pointers.

Examinations and examination scripts

Programming examinations were set by a panel of lecturers involved in teaching the subject. Questions and solutions prepared by the panel were handed over to the Examinations Section. Examinations together with the marking schemes were reviewed by external examiners to ensure adequacy and conformity to the syllabus. Examination scripts were marked by the same panel of examiners. After the examination and publication of results, the researchers performed a random selection of 34 scripts from past papers, which were compared with 28 scripts from the volunteer group.

Errors protocol and programming manual

The errors protocol was constructed by combining types of programming errors and misconceptions as discussed by Du Boulay[5] Ramadhan[1] and Alain.[18] This protocol was constructed and employed to analyse errors committed by students when answering examinations. The list of errors included undeclared variables, uninitialised variables, setting variables to uninitialised value, using = instead of = = to check for equality, confusion due to repetition of a variable in a statement (i.e. x = x + 2), and algebraic noise (i.e. treating *a* = 4 as 4 = *a*;) and *a* = *b* taken as '*a* is linked to *b*'. The protocol



```
// Program 3
main()
{
int sq(), x, z;
cout<<"Enter a number";
cin>>x;
z=sq(x);
cout<<"The square of"<<x;
cout<< "is"<<z;
}
int sq(y)
{
return (y*y);
}
```

Source: Authors' own data
RAM, computer memory.

FIGURE 4: Functions interpretation (Program 3).

focuses only on fundamental errors. In order to guarantee simplicity, more complex errors such as those pertinent to function arguments, confusion of types and violation of array boundaries were excluded. A complete programming manual in C, derived from Mselle[14] written in MTL and describing all programming aspects was used. Employing the protocol, errors committed by students were analysed, recorded and counted.

Procedure

The lecturer of the experiment group was requested to use the manual, which was written based on MTL, to explain all aspects of introductory programming in C. The lecturer agreed to introduce the manual to students and advised them to use it for their studies and laboratories along with other books. Students were informed of the experiment and the specific approach of the manual. Initially students were reluctant to use the manual because they had already been given class notes, from which they were told examination and tests would be set. The lecturer continued to employ the manual for lecture and laboratory sessions due to its illustrative approach and the simplicity with which the material is presented. Students built up the courage to volunteer for the experiment after attending four hours of classes. Twenty eight volunteers were instructed using the MTL approach. In the end examination scripts were used to analyse the amount of errors committed, and this amount was compared with errors counted in 34 scripts randomly selected from the population of past papers.

Results

Error-count results from the control and experiment groups are summarised in Table 1.

Results from using error-counts in examination scripts from two groups of students, where one group was instructed using MTL and the other through the conventional approach, suggested a highly significant statistical difference (Mann-Whitney test $U = 58$; $N1 = 34$; $N2 = 28$; $p = 0$). This indicates that students who were instructed through MTL committed less errors and therefore had a better understanding of the subject compared to those who were taught through the conventional approach.

Given this result, we tend to agree with Wilson and Moffat[9] that programming is not difficult; rather, it is the way it is presented to the learners which breeds confusion, leaving the learners with various misconceptions which frustrate the effort to learn. With MTL, aspects such as variable declaration, assignment, variable overwriting and data operations are made obvious at the very beginning. With MTL, as demonstrated in Figure 2, Figure 3 and Figure 4, the meaning of variable declaration, data feeding and data operation are made obvious by illustrations. Why variables are declared is made obvious, and what happens to each variable during initialisation is clearly distinguished. What and how various operations are carried out and the meaning of assignment and roles of variables are clearly demonstrated at every turn of the code. With these aspects made clear, confusion and ambiguities are potentially mitigated, together with associated errors.

TABLE 1: Error-count summary.

Groups	N	Total error counts	Mean	SD
Control	34	392	10.315	4.32049
Experiment	28	208	7.428	4.98463

N, number; SD, standard deviation.

Discussion

Since MTL permits the programmer to illustrate execution of the code from the machine's point of view, it cultivates the sense of 'I am working the machine'. Samurcay[2] observes that novices tend to feel that the machine is 'reasonably human', and they therefore expect it to understand the code 'as it was intended' rather than 'as it means'. Since the machine does not turn out to be 'reasonably human', novices are discouraged by bugs reported by the compilers, and their desire to code is undermined.

Lack of a tool to illustrate the effect of each line of code on the machine is a major source of misconceptions. Du Boulay[4,5] proposes a tool representing a notional machine, advising that such a machine should observe simplicity, be small and have few constructs. He argues in favour of implementing a language in such a way that either pictorial or written traces can be displayed. MTL, as illustrated in Figure 2, Figure 3 and Figure 4 and as presented in Mselle[14] is a programmer-driven visualisation device which bears most of these characteristics. Being learner-driven, MTL has the capability to transfer programming authority to the programmer, whilst creating the sense that the machine is not responsible for mistakes committed by the novice.

Flowcharts, animation tools and Memory Transfer Language

Flowcharts are amongst the traditional tools employed in illustrating the logic of the code. Flowcharts are powerful tools for algorithm planning.[13,19] They are, however, unsuitable for precise close-tracking and debugging. Code simulation and animation tools, such as BlueJ, Jeliot and Plan Ani, have been introduced recently to illustrate the logic of the code to the learner.[20] Animations are suitable for precise close-tracking. In effect, they are a plausible breakthrough. However, since animations are entirely machine-driven, their exclusive use may accentuate the role of the machine, which is already made enormous by the compiler and the editor.

Code animators used alone may reinforce the notion that the machine is 'reasonably human' and totally in control of the programming process. Mselle[14] has shown that there can be a one-on-one relationship between language statements and the machine semantics, as may be reflected by memory status. MTL allows the novice to play back the code from the machine's point of view. MTL provides the novice with the necessary freedom from compiler bullying and authority over the machine. Since MTL is a learner-driven device, it is unconstrained by the initial design of the code. To its additional credit, MTL can be used in conjunction with other animators and flowcharts.

Use of Memory Transfer Language for close-tracking

Perkins et al.[15] posit that a vital skill for any programmer is 'close-tracking', which means reading the written code to determine precisely what it does. Close-tracking can be useful for filtering out bugs before testing a program. It is also important for diagnosing bugs that appear when the program is compiled, and sometimes gives clues as to how bugs should be repaired. Accurate close-tracking is a mentally demanding activity which requires understanding of the primitives of the language and the rules for flow of control. Perkins et al.[15] conclude that although in principle close-tracking is a mechanical procedure, in practice it often proves a source of difficulties. Students commonly neglect to do it, but need the information that close-tracking provides in order to untangle a problem. However, when the novices attempt to track what their programs are doing, they often fail.[15]

Close-tracking is related to a methodical approach to programming. Kagan and Kogan, cited in Perkins et al. [15], confirm that those students who naturally approach problems methodically and reflexively may be better trackers than those who approach their work in a more trial-and-error or impulsive fashion. However, close-tracking can only be attractive to novices if there is an effective tool to track, diagnose errors and precisely correct them.[4,15] As demonstrated in Figure 2, Figure 3 and Figure 4, MTL stands out as being such a tool. If the novice is not sure of the semantics of the code, then close-tracking is not useful. Mselle[14,17] has shown that most elementary programming concepts can be entirely conveyed to novices through MTL. In this approach novices are trained to learn programming by an amalgamation of lectures, laboratory classes and discussion through close-tracking using MTL.

Memory Transfer Language and Feedback factor

Learning feedback is an ingredient which may determine whether a learner keeps on with a subject or drops out. Feedback has been demonstrated as playing an important role in instruction. Many learning theorists posit that feedback is essential to students' learning. In general, instructional feedback provides students with information that either confirms what they already know or changes their existing knowledge and beliefs. Meaningful and timely feedback that is of high quality helps students become cognitively engaged in the content under study, as well as in the learning environment in which they are studying. Feedback serves as a type of formative assessment, designed to improve and accelerate learning. Specifically, feedback is described by Ertmer et al.[21] as 'anything that might strengthen the students' capacity to self-regulate their own performances'.

When writing code using a conventional approach, immediate feedback comes from the compilers. Compilers are unforgiving, bullish, and sometimes misleading. This unfortunate situation may be one of reasons why many novices are disheartened by their early experimentation with programming, and hence develop a dislike for the subject.[3] The need to escape from the vagaries of compilers has prompted some experts to propose specialised languages for novices.[4,5,9]

MTL provides the novice with an instrument to play the role of compiler outside of the machine environment, putting the novice on a par with the machine with regard to verification of the correctness or incorrectness of the program. As demonstrated in Figure 2, Figure 3 and Figure 4, instead of bullish and terminate feedback provided by the compiler, MTL provides feedback in an exploratory manner. It constitutes a temporary refuge from the machine compiler, and hence a stimulus to persevere with learning rather than giving up.

Conclusion

The objective of this study was to introduce MTL and evaluate its impact in reducing common misconceptions in programming which culminate in programming errors by novices. We were specifically interested in determining whether using MTL in teaching programming would reduce common misconceptions and consequently avoid common errors in programming and improvement of comprehension. Results from the experiment confirm that misconceptions and errors can be significantly reduced if students are instructed using an instrument that can give them the power to illustrate the different aspects of programming. Specifically, a tool such as MTL has proved to be handy for novices to use to close-track, debug and provide feedback to them. It was found that using MTL in learning programming improves comprehension of the subject. There are, however, some shortcomings with this study. The sample size is too small to justify generalisation, and the population is taken from one university. The lecturer of the experiment group could have been the better teacher with or without MTL, which could have contributed to good results on the part of the experiment group.

Recommendations

More experiments in different settings should be carried out with a much larger and diverse population, to confirm the effectiveness of MTL. More areas of research on the effectiveness of MTL in distance learning and in different age groups are open for future investigation.

Acknowledgements

The author would like to thank Paula Saphir for her willingness to carry out the experiment. David Moffat is thanked for his invaluable time in reviewing this paper and for advice he gave. Jorma Sajaniemi and Moti Ben Ari are thanked for supporting the MTL idea from its inception (in the form of RAM diagrams) to the present stage.

Competing interests

This study was motivated by academic curiosity. It was not initiated by any third party. The authors declare that they have no financial or personal relationships which may have inappropriately influenced them in writing this paper.

Authors' contributions

L.M. (University of Dodoma) was the lead researcher, and H.T. (University of Dar es Salaam) performed the statistical analysis.

References

1. Ramadhan H. Discover. Cybernet Syst. 1992;(31):87–114. http://dx.doi.org/10.2190/FGN9- DJ2F-86V8-3FAU
2. Samurcay R. The Concept of Variable in Programming: Its Meaning and Use in Problem-Solving by Novice Programmers. Education Stud Math. 1985;16:143–161.
3. Dehnadi S, Bornat R. The Camel has Two Humps (working title). 2006 [cited 2009 Nov 20]. Available from: http://eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf
4. Du Boulay B, O'Shea T, Monk J. The Black Box inside the White Box: Presenting Computing Concepts to Novices. Int J Man Machine Stud. 1981;14:237–249. http://dx.doi.org/10.1016/S0020-7373(81)80056-9
5. Du Boulay B. Some Difficulties of Learning to Program. J Educational Computing Research. 1986;2:459–472.
6. Mayer RE. Comprehension as Affected by Structure of Problem Representation. Mem Cognition. 1976;4:249–255. http://dx.doi.org/10.3758/BF03213171
7. Mayer RE. The Psychology of how Novices Learn Computer Programming. Computer Surveys. 1981;13:121–141.
8. Yousoof M., Sapiyan M., Kamaluddin K. Measuring Cognitive Load - A Solution to Ease Learning of Programming. 2007 [cited 2007 Aug 22]. Available from: http://waset.org/journals/waset/v26/v26-41.pdf
9. Wilson. A, Moffat D. Evaluating Scratch to Introduce Younger Schoolchildren to Programming. In: Lawrance J, Bellamy R (eds.). Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group; 2010 Sep 19–21; University Carlos III of Madrid, Leganés. Madrid: Maria Paloma Díaz Pérez & Mary Beth Rosson.
10. Mayer RE. Different Problem Solving Competencies Established in Learning Computer Programming with and Without Meaningful Models. J Edu Psych. 1975;67:725–734.
11. Ben Ari M, Sajaniemi J. Roles of Variables from the Perspective of Computer Science Educators. [cited 2004 Jun 28–30]. Available from: http://cs.joensuu.fi/pub/Reports/A-2003-6.pdf
12. Ben Bassat LR, Ben Ari M, Uronen P. An Extended Experiment with Jeliot 2000. Proceedings of the 1st International Program Visualization Workshop. Pavoo, Finland: University of Joensuu Press; 2001. p. 131–140.
13. Ziegla U, Crews T. An Integrated Program Development Tool for Teaching and Learning how to Program. Paper presented at: SIGCSE 1999. Proceedings of the 30th SIGCSE Symposium on Computer Science education.1999 Mar 24–28; New York: ACM; 1999. p. 276–280. http://dx.doi.org/10.1145/299649.299786
14. Mselle L. *C++ for Novice Programmers*. Berlin: Lap Lambert Academic Publishing; 2010.
15. Perkins DN, Hobbs HR, Martin F, Simmons R. Conditions of Learning in Novice Programmers. J Educational Computing Research. 1986;2:37-55. http://dx.doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL
16. Alain A. The Common Programming Errors [cited 2009 Mar 14]. Available from: http://cprogramming.com/how_to_learn_anything.html.
17. Mselle L. Enhancing Comprehension by Using Random Access Memory (RAM) Diagrams in Teaching Programming: Class Experiment. In: Lawrance J, Bellamy R (eds.) Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group; 2010 Sep 19–21; University Carlos III of Madrid, Leganés. Madrid: Maria Paloma Díaz Pérez & Mary Beth Rosson. http://www.ppig.org/papers/22nd-Teach-1.pdf
18. Alain A. What's Wrong With My Program - Common Programming Mistakes [cited 2009 Mar 20]. Available from: http://cprogramming.com/tutorial/common.html
19. Scott A, Watkins M. Duncan M. A Step back from Coding – An Online Environment and Pedagogy for Novice Programmers [cited 2005 Aug 22] Available from: http://ics.heacademy.ac.uk/events/jicc11/scott.pdf
20. Ala Mutka K. Codewitz Needs Analysis [cited 2003 Feb 22]. Available from: http://cs.tut.fi/~edge/literature_study.pdf
21. Ertmer PA, Richardson JC, Belland B, Camin D, Connolly P, Coulthard G. Using Peer Feedback to Enhance the Quality of Student. J Computer-Mediated Comm. 2007;*12*(2):412–233. http://dx.doi.org/10.1111/j.1083-6101.2007.00331.x