Received: 05 Dec 2024 | Revised: 05 May 2025 | Accepted: 23 Jun 2025 | Published: 25 Aug 2025
Volume 3(2) Pages 1-23
DOI: https://doi.org/10.56279/jicts.v3i2.160

JICTS
Journal of ICT Systems

Teaching Mental Models in Introductory Programming

Leonard J. Mselle

[a]*Department of Computer Science, The University of Dodoma, Dodoma, Tanzania*

[1]Corresponding author
Email: mselel@yahoo.com

Funding information

This work was self-funded.

Keywords

Mental models
Memory Transfer Language (MTL)
Language-trap
Cognitive Load
Program Visualization

Abstract

In this article, an evaluation of syllabi, books, teaching materials, and examinations concerning introductory programming revealed that the subject disproportionately focuses on teaching and learning language features instead of mental models. It is demonstrated that the shift from low-level to high-level languages resulted in the "language-trap" that leads to an emphasis on language features at the expense of mental models. To mitigate the language-trap effect, a novel instructional approach called MTL three-tier that combines low-level syntax, Memory Transfer Language, and high-level syntax is proposed. Results from two experiments show that using assembly codes in combination with the MTL-three-tier approach at the beginning of the course assists instructors in avoiding the language-trap. For novices, the cognitive load is reduced, consequently, increasing the ability to form viable mental models. Results from the first experiment show that novices in the experimental group were 2.17 times more likely to form viable mental models than those in the control sample. From the second experiment, results show that novices from the experimental group were 14.50 times more likely to avoid common errors in introductory programming than were the novices in the control group.

1. Introduction

Learning computer programming is achieved by obliging the student to learn a set of simple steps expressed in statements of a certain programming language. The learning process becomes more complex as one progresses. The logical combination of the code-statements is a program which, when executed by the computer, must solve a problem according to the programmer's intention [1, 2-4]. A programming novice is required to master the syntax and semantics of a certain programming language while at the same time striving to create viable mental models. If the mental models of the novices are unviable the written program/code will be meaningless to the computer.

Mental models in introductory programming

In introductory programming, mental models entail the internal representations of a real system's structure, organization, and behavior [2, 5]. In the process of writing a code, there are three interacting entities: the programmer (who creates the code), the code (being created by programmers), and the computer that interprets and executes the code being created. The programmer thinks and writes the code using a programming language that resides in the computer. The code is interpreted by the computer and, if it is syntactically and semantically correct, the computer executes it, providing the solution to the problem as intended by the programmer. In writing a code, the programmer's thoughts, as expressed in statements, are fluid and dynamic. In this regard, they are a mixture of what is correct and what is not correct. This is what is termed as mental model. While the programmers' mental models are transient, the computer interpretation of the code is rigid and static.

Cognitive load (CL) in learning programming

Introductory programming is a subject that requires the learner to remember concepts and remember them without the slightest ambiguity. It requires the learner to understand and master the concepts together with their relationships. Programming demands one to apply the concepts and apply them according to the rules of the language being used and the nature of the problem being solved. It requires one to carry out critical analysis of the code and associate it with the complexity of the problem being solved. Programming demands one to be savvy in designing a solution to solve existing problems. All these cognitive constructs must be combined in a non-linear fashion to achieve mastery of the subject. This combination of different cognitive processes is what makes programming a subject with high cognitive load (HCL) [6-8].

Due to the HCL in introductory programming, the formation of correct/viable mental models is difficult. In addition to the necessity to form viable mental models, the learner is required to master the syntax and semantics of a programming language, which is in itself very difficult. The need to form viable mental models and the necessity to master the syntax and semantics of a programming language render introductory programming a very challenging subject for novices [1-6, 9-12].

HCL and mental models in introductory programming

Consider, for example, the two statements in C/C++/Java: *int x; x = 4;*. From the mental model perspective, these are two interrelated statements because they are dependent on one variable that has been identified by *x*. While the first statement aims at reserving a storage space identified by *x*, the other is aimed at inputting data in *x*. This relationship is an internal representation of a reality, both in the programmer's mind and in the computer. Simple and trivial as it may seem, this relationship can take different forms in the mind of a novice.

For example, when Bornat et al. [6] conducted experiments on mental models using the code segment, as depicted in Figure 1, they identified 11 different mental models based on how novices interpreted the basic expressions with the "=" operator.

```
int a, b, c;
a=5; b=3; c=7;
a=c; b=a; c=b;
```

Figure 1. A test question about the assignment (=) operator [6, Figure 2 p. 3].

Similar results were reported by Castrro-Alonso et al. [8] and Dentamaro et al. [12]. Under such a situation, if the instruction is not designed with this complexity taken into account, the novices generate unviable mental models that are different from the correct solution being pursued.

Learning Edge Momentum and failure rate in introductory programming

According to Robins [11], when CL is high, it leads to negative learning edge momentum (LEM). Negative LEM at the beginning of the course is a major cause of demotivation among novices ultimately leading to failure. Robins [11] posited that the LEM effect, negative or positive, is self-reinforcing.

The failure rate in introductory programming continues to be a preoccupation of researchers [1, 2, 13]. To address the HCL in programming, there have been numerous studies. The majority of these are directed at programming languages as opposed to mental models [2, 12, 14]. There are a number of researchers who agree that effective instructional design for introductory programming will lead to positive LEM among novices, consequently reducing the failure rate [11]. Together with the selection of language, there are numerous studies proposing program visualization (PV) as part of the solution [9, 15-19].

Language feature versus mental models

Most programming instructors and books place heavy emphasis on teaching language features, based on the assumption that once novices master these features, they will naturally develop effective mental models. Attention to mental models is either completely ignored or relegated to a secondary position.

Consider the common practice of starting introductory programming lessons with a messaging statement, such as *cout <<Hello World”;* instead of *int x;* followed by *x = 4*; The common practice of starting programming with output statements instead of declaration and input is one indicator of how programming lessons are more focused on language features at the expense of mental models.

In addition, programming examinations are an important area where the predominance of language features can be demonstrated. If in examinations language syntax and semantics are exhaustively covered, while mental models are ignored, then this is another evidence of bias towards language features. If mental models were given the importance they deserve, questions seeking to test the acquisition of mental models would have formed a significant part of the examinations. Looking at the original works of programming, such as the codes by Ada Augusta Lovelace, the presence of memory drawings/sketches is evident [5]. However, it is rare to find programming examinations with the inclusion of memory drawings/sketches.

Research on the use of low-level syntax in combination with PVs and high-level syntax in teaching programming is scarce. Analysis of PV research shows that the majority of these studies are more focused on high-level syntax with the obvious exclusion of low-level syntax. While it is widely agreed that low-level syntax is nearer to the machine, and therefore better at conveying viable mental models than high-level syntax [11, 16, 20], it remains unclear why the low-level syntax is excluded from programming instructional design.

Research on why learning to program has been subsumed into learning language features is scarce. The debate on the effective use of PV to assist in the formation of viable mental models is still ongoing [11, 15-21]. This paper expounds on the predominance of the language-trap (language-first approach) at the expense of mental models in teaching and learning introductory programming.

Further, a new approach that focuses on mental models to teaching introductory programming is designed and tested through two empirical experiments.

This research had three objectives: the first was to evaluate the language-trap in teaching and learning introductory programming and its adverse effect on the formation of mental models; the second was to propose and demonstrate a novel instructional design (the MTL three-tier Approach) based on low-level language combined with high-level languages (HLL) and MTL PV in teaching and learning introductory programming; the third was to conduct two class experiments to find out the impact of the MTL three-tier approach on novices' ability to acquire viable mental models and their cognition of introductory programming as reflected by their performance.

The rest of this article is organized as follows: Section 2 delves into the method. Section 3 delves into the evaluation of current approaches to teaching and learning introductory programming. In Section 4, the MTL three-tier approach (Assembly + MTL-PV+HLL) is demonstrated. Section 5 is about the experiments. Section 6 is about results and discussion, and Section 7 presents conclusions and recommendations.

2. Method

To attain the desired objectives, a mixed approach methodology was followed. The first part included a literature review combined with an evaluation of the current teaching and learning approaches to introductory programming. The evaluation was carried out by checking and critiquing various randomly selected introductory programming syllabi, books and online videos and notes used in universities or institutes. The second part involved the use of the design science research method (DSRM) to evolve and describe the blueprint of the MTL three-tier approach. The third part is based on class experiments, in which the effectiveness of the MTL three-tier approach in a classroom setting was tested.

The document analysis included a qualitative evaluation where written and electronic documents were checked to determine the extent to which they emphasize language features as compared to mental models. Randomly selected, syllabi, programming books, online programming content, and examinations were checked for their balanced inclusion of *worked examples*, *verbose discussion of language syntax*, and *use of visualization/diagrams.* The details of this evaluation are presented in Table 1.

The DSRM focuses on developing practical solutions through an iterative process of design, development, and evaluation. The task involved the creation of an artifact, which is the MTL three-tier approach. The design of the MTL three-tier approach is detailed in Section 4.

For the third objective, two class experiments were carried out. In the first experiment, a convenient sample of 2,322 university students was involved. A question on *selection* to test the formation of viable mental models was administered. The second experiment focused on summative examinations, where randomly selected 1,200 scripts were checked for common errors committed by novices. These errors, which imply failure to form viable mental models, were counted from the first two *code-questions* attempted by the novices for statistical analysis and discussion.

3. Evaluation of the Traditional Approach to Teaching and Learning Introductory Programming

The majority of introductory programming researchers agree that teaching and learning the subject is difficult, and therefore the failure rate among novices is very high [1, 6, 15-19]. Numerous researchers contend that HCL in programming is due to the languages used to instruct. There is, however, another group who posited that HCL in forming viable mental models is the major cause of failure [15-22]. Since the majority of the introductory programming research contends that language features are the major cause of HCL, most solutions to HCL are language-focused [19, 20-24]. This is what this article terms as the *language-trap.*

The Language-trap in introductory programming

The language-trap in teaching and learning introductory programming is so entrenched that it is discernible neither during instructional design nor during course conduct. As an example, starting to teach programming with a message output, such as *cout << "Hello world;"* instead of *mov exa 4* is almost a universally accepted standard. This partly proves how a huge emphasis is placed on compilation and language features instead of mental models or computational thinking (CL). The initial success that a novice derives from the successful compilation of *cout <<* "*hello world";* is short-lived once the reality of programming unfolds. According to the LEM theory, if the formation of viable mental models is not attained at the beginning of the study, all the effort exerted on understanding language features is futile due to the high interdependence of concepts [11].

Consider, for example, the rationale of including the coverage of all three loop constructs (i.e., *for(), while()*, and *do{..} while())* in the syllabi allocating each with equal time for lectures, tutorials and lab sessions. In fact, these three constructs represent somewhat the same mental model. This is just another indicator of more emphasis invested on language features and less on mental models.

Similarly, the majority of research on teaching and learning introductory programming has been more focused on languages and less so on mental models. There are numerous papers discussing and proposing the first language to teach introductory programming [21], without any suggestion of mental models. On the other hand, although PV research is naturally about visualization of code when running a program, most PV research and experiments have largely been influenced by high-level languages [15-24]. The tradition of giving primacy to languages might have its roots in the fact that one cannot master programming in C, C++, or Java, for example, without mastering the syntax and semantics of the language. Although this is an indisputable fact, it has led most programming instructors to trust the passive chalk-and-talk approaches and compilation to teach the language syntax using worked examples while ignoring the use of combined PVs along with worked examples in order to give required emphasis on mental models.

Apart from books, notes, and syllabi, another manifestation of the language-trap in teaching programming is the assumption by most instructors that high-level syntax is somehow an ultimate solution to the hardships of the subject in comparison to low-level syntax [4, 21].

Shifting from mental models language features

When programming was undertaken using machine/assembly languages, mental models constituted the core of the learning process [5]. It is impossible to write correct low-level code without first having the correct mental model of the effect of the statement being written. With the advent of high-level syntax and its use in teaching programming, language features became the major preoccupation. As a consequence, most solutions suggested for programming are concerned with language features instead of mental models. In most literature, among the prominent reasons given for failure are as follows: abstract nature of programming; first language used to teach;

dynamic nature of the subject; poor instructional design; poor curriculum implementation that fails to address the complexity of the subject; and inadequately prepared learning materials [17, 25].

Although not explicitly acknowledged, the language-trap problem in introductory programming can be found in studies by Kraleva et al. [21] and Mtaho and Mselle [17], who concluded that instructional design and programming examinations across universities and among instructors are vastly varied.

The undeclared and unintended language-trap could be the major reason that most solutions addressing difficulties in learning programming are language-focused instead of mental-model focused.

Transition from machine/assembly syntax to high-level languages

In the beginning, the ability to program a computer required mastery of the machine language [5]. Due to the complexity of machine syntax, machine language was replaced by assembly language, which is a machine language based on mnemonics.

The need to avoid the difficulties of mastering machine or assembly language vocabulary gave rise to high-level languages. These languages were favored over low-level languages due to their two important features. One is their syntax, which is made of English words instead of mnemonics or machine vocabulary. The second is the introduction of code blocks, which facilitate structured programming. With the introduction of high-level languages, it was expected that learning programming would be easier. However, the adoption of high-level syntax did not improve comprehension of programming [2, 6].

Among the introductory programming community, research on the revival of assembly languages, which are inherently associated with machine language, is scarce. Most researchers have ignored the possibility of proposing a partial return to machine or assembly language as a bridge toward the formation of viable mental models and eventually assisting novices to work more comfortably with high-level languages.

Among various efforts to reduce failure among novices, one has been to reduce the impact of abstraction that is caused by languages. To this end, teaching programming has traditionally been combined with some sort of visualization. There are various studies on the use of program visualization. Some of these have been compiled, analyzed, and evaluated [9, 17]. However, a proposal for the combination of PV with low-level syntax to assist novices in forming viable mental models is nonexistent. The shift from low-level syntax, which resulted in a shift from mental models to high-level language, culminated with an instructional design that is devoid of mental models.

When an evaluation to find out the balance between language features and mental models in teaching and learning materials (syllabi, books, notes/videos, and examinations) was carried out, as revealed in Table 1, none of the syllabi representing eight randomly selected institutions had mental models, visualizations or sketching as part of teaching in general and imparting mental models in particular. On the contrary, all syllabi contain detailed coverage of the language features (syntax) even at the degree of redundancy; For example, loops. The same biases were observed on a sample of six randomly selected books, six teaching videos and examinations-samples taken from twelve prominent institutions worldwide.

From these results and the evidence shown in the reviewed literature, it can be concluded that, currently, mental models in teaching and learning programming have received less attention in comparison with language features.

4. The MTL three-tier approach (Assembly + MTL-PV+HLL)

In programming, mental models are depictive representations assumed to contain a visual-spatial structure analogous to the ideas presented in the code-text. Thus, understanding a text segment of a code often requires translating the text into a coherent mental image [2]. Generally, a mental model of anything is represented by an image (drawing/sketch) or a tangible object/model. In programming, the use of drawings to represent the effects of the code has been in use for decades [5]. With the advent of high-level languages, the use of flowcharts and trace tables was intended to address the issue of mental models [2, 15-19].

While there is general use of flowcharts in teaching and learning introductory programming, flowcharting has two limitations. Firstly, flowcharting was introduced almost at the same time as high-level languages. Just like high-level languages, flowcharting is too much abstracted from the machine. As a result, flowcharting cannot represent the concrete idea of an instruction or an expression. Secondly, flowcharting uses more than 20 symbols with different rules for combining each of them. This increases the CL of the subject. For these reasons, although flowcharting is as old as high-level programming, its success and application are limited [2]. For example, consider basic code statements, such as variable declaration: *int x, y, z;* data input: *x = 2; y = 3;* and processing and outputting: *z = x + y;* Figure 2 depicts the visualization of these statements using an MTL PV in comparison with a flowchart. From Figure 2, it is evident that flow charting is more complex than the MTL PV.

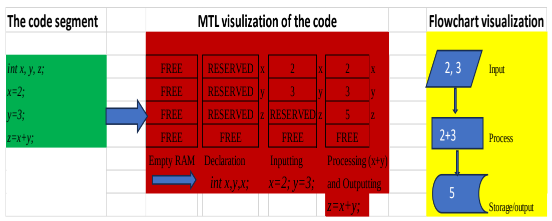


Figure 2. Visualization of the code segment (MTL left-hand side) compared with (flowcharting right-hand side).

Basic design and pillars of the MTL three-tier approach

As shown in Figure 2, MTL is a PV depicting computer RAM as code is executed. MTL PV uses one symbol (rectangles) to mimic the computer RAM to represent the learner's mental model. That is his/her mental representation of the structure of computer memory, its organization, and behavior due to actions of declaration, inputting, processing, and outputting (the code).

Together with rectangles, MTL relies on a familiar symbol (containers or physical models as depicted in Figure 4) to portray the transformation of the computer RAM as each code statement is translated and executed. This use of familiar objects for visualization reduces the CL because the diagram frees the working memory, allowing it to deal with the novel part of the content. In addition, MTL allows the novice to use any sort of small container, such as bottle caps, to mimic the computer RAM and use physical objects such as pebbles or stones as inputs or outputs (Figure 4). The same RAM diagrams can be animated using tools such as *Celiot* as depicted in Figure 8.

As depicted in Figures 4-7, 10, and 11, the MTL PV assists the instructor in using multimedia (text-visual-audio) and multiple senses (text-visual and kinesthetic/tactile) to channel more effort to the formation of mental models, reducing the CL and increasing the chance for a positive LEM. On the contrary, flowcharting employs four symbols in this case, which increases the extraneous cognitive load. In addition, flowcharting is not matched with the idea of computer RAM, hence abstracting the learner from the computer. Furthermore, flowcharting conveys different meanings to different novices. This is one reason that, although flowcharting is the most used visualization scheme in introductory programming, its use has not been effective [2].

If teaching and learning introductory programming were designed and carried out with the primary objective of building mental models, then each aspect of programming, that is, variables, data feeding, data processing, outputting, sequence, selection, loops, arrays, functions, and file handling, would be presented with drawings similar to the ones in Figures 4-7, 10, and 11 and, where every worked example is visualized using the MTL RAM diagrams. The fact that most programming materials in books and learning resources do not make visualization compulsory is yet another proof of the influence of the language-trap. Due to the language-trap, most of the proposed PV solutions are language-tuned instead of being model-focused.

The use of the MTL three-tier approach is based on three pillars, which include (1) assembly codes, (2) high-level codes (worked examples), and (3) the MTL PV. The MTL PV is used to visualize the basic programming structures and mechanisms using code segments, physical models, RAM diagrams, and animations.

Assembly code + MTL physical models

As exemplified and depicted in Figure 3, a basic assembly code is associated with a physical model in Figure 4. Registers/variables are labeled manually and inputted with pebbles to analogize input. Similarly, the process is analogized by the increment of pebbles in the destination containers. This physical model can be represented in an MTL RAM diagram as depicted in Figure 5. The MTL depiction is language independent.

```
1: MOV eax, 2
        MOV ebx, 3
        ADD      eax, ebx, ecx
```

Figure 3. An example of assembly code: a simple calculator.

Worked examples of High-level codes + MTL RAM diagrams

As is the case for the assembly codes, the basic high-level codes (in any chosen programming language) can be combined with RAM diagrams to visualize the structure, organization, and the internal behavior of the code segment (mental model). As depicted in Figure 5, an example a basic

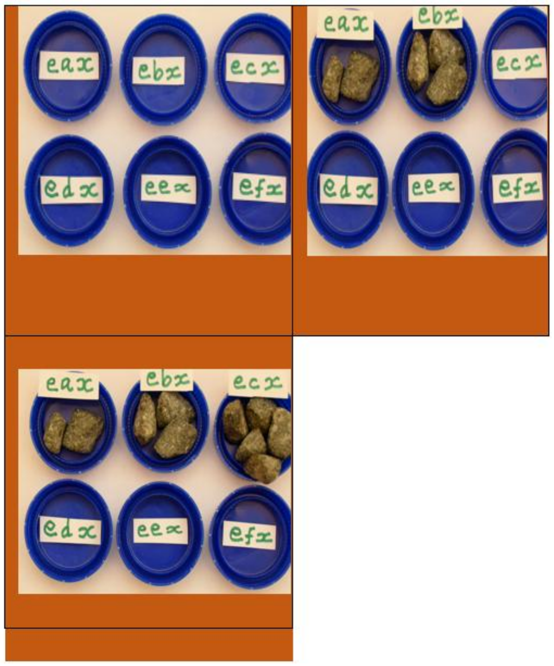
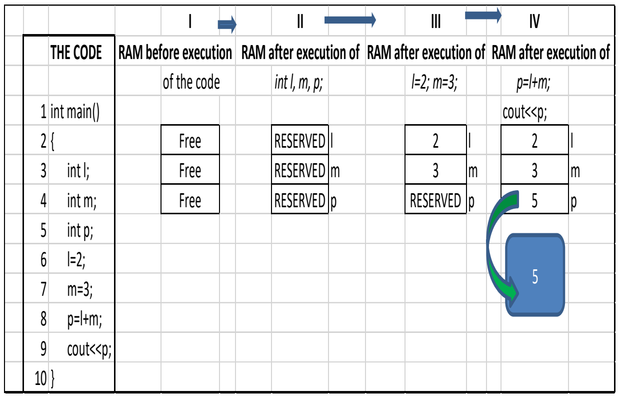


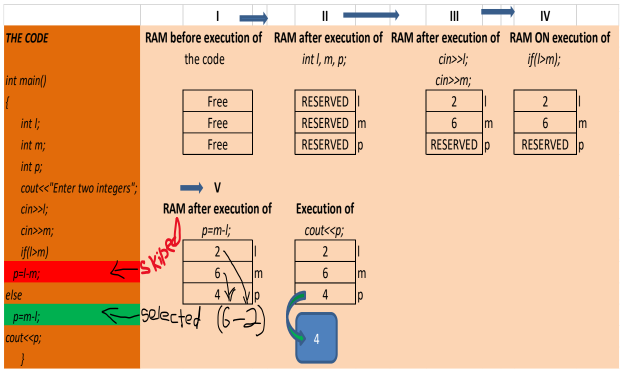
Figure 4. An example of RTL physical models mimicking computer registers on execution of the assembly code in Figure 3.

code (sequence) is depicted from the declaration, data inputting, processing, and outputting. Figure 6 is an example of a depiction of selection (*if()* construct), and Figure 7 is the visualization of loops (*while()* construct). The same can apply to the concepts of arrays, file handling, and pointers. In addition to the depiction of code-logic, MTL PV can be used by instructors to depict unviable mental models and guide novices to avoid them as depicted in Figure 11.


I
II
III
IV
THE CODE
RAM before execution of the code
RAM after execution of *int l, m, p;*
RAM after execution of *l=2; m=3;*
RAM after execution of *p=l+m; cout<<p;*
Free
Free
Free
RESERVED l
RESERVED m
RESERVED p
2 l
3 m
RESERVED p
2 l
3 m
5 p
5

```
int main()
{
    int l;
    int m;
    int p;
    l=2;
    m=3;
    p=l+m;
    cout<<p;
}
```

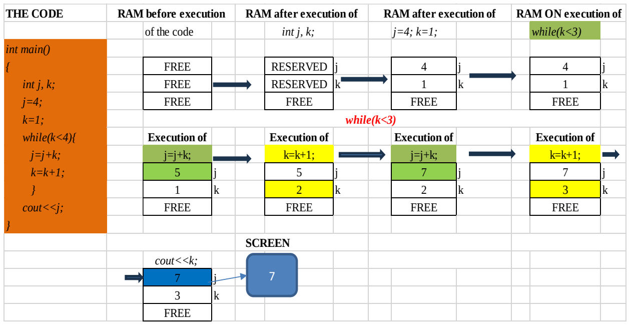
Figure 5. Example of MTL visualization of elementary variables behavior.



```
THE CODE

int main()
{
    int l;
    int m;
    int p;
    cout<<"Enter two integers";
    cin>>l;
    cin>>m;
    if(l>m)
  p=l-m;
else
  p=m-l;
cout<<p;
    }
```

I
II
III
IV
RAM before execution of the code
RAM after execution of int l, m, p;
RAM after execution of cin>>l; cin>>m;
RAM ON execution of if(l>m);
Free
Free
Free
RESERVED l
RESERVED m
RESERVED p
2 l
6 m
RESERVED p
2 l
6 m
RESERVED p
V
RAM after execution of p=m-l;
2 l
6 m
4 p
Execution of cout<<p;
2 l
6 m
4 p
4
skiped
selected
(6-2)

Figure 6. Example of MTL visualization of selection.

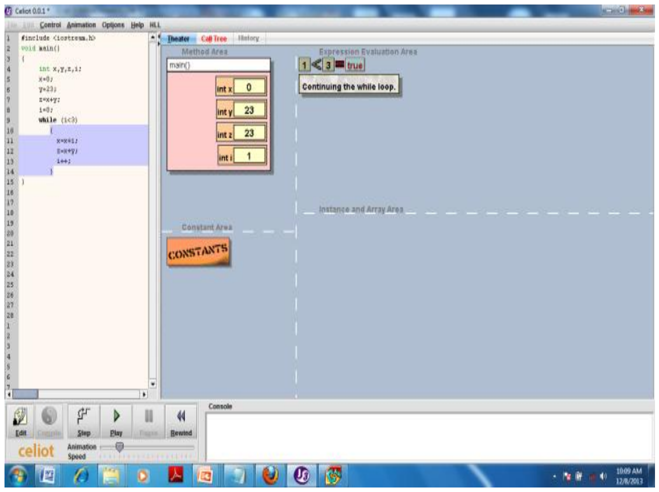


```
THE CODE
int main()
{
    int j, k;
    j=4;
    k=1;
    while(k<4){
      j=j+k;
      k=k+1;
      }
    cout<<j;
}
```

Figure 7. Example of MTL visualization of loops.

MTL high-level codes and animations

Computer animation is a powerful tool for simplification of cognition [15-17]. Using *celiot* animation tool, Masoud and Mselle [15] demonstrated that computer animations significantly reduced the cognitive load.



```
#include <iostream.h>
void main()
{
      int x,y,z,i;
      x=0;
      y=23;
      z=x+y;
      i=0;
      while (i<3)
        {
          x=x+i;
          z=x+y;
          i++;
        }
}
```

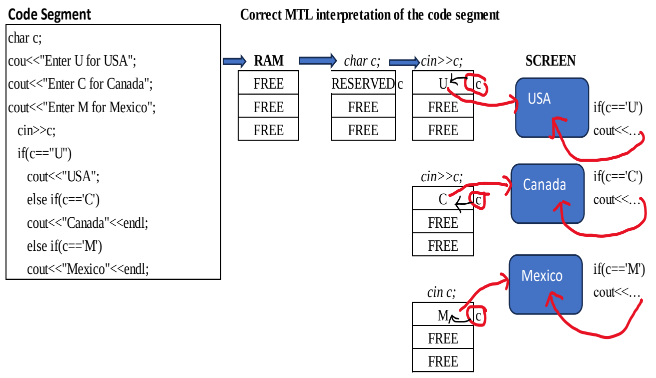
Figure 8. Example of *Celiot* animation of a *while ()* loop [16, p. 245].

High-level codes + MTL-PV and Mental models

Since mental models constitute the core of the MTL three-tier approach, instructors are encouraged to use these tools to point out the common unviable mental models that persist in novices' minds. Figure 11 is example of such predictable, unviable mental models that can be depicted, discussed, and discouraged by the instructors.

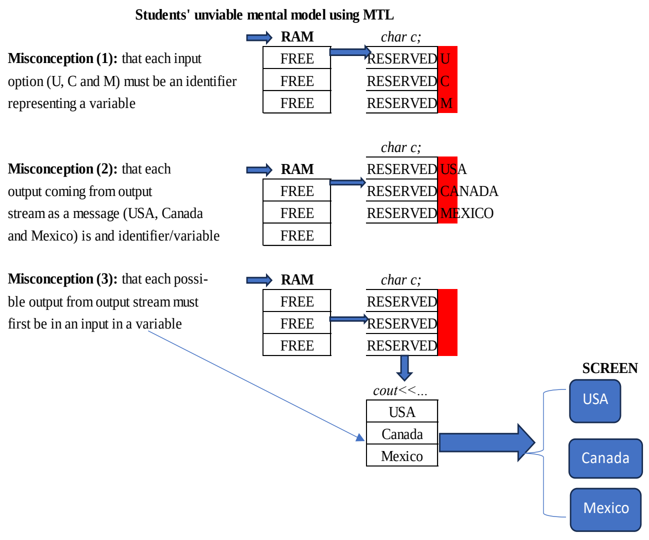
```
int main()
{
          char c;
          cout<<"The following program displays the
          NAFTA countries"<<endl;
          cout<<"Select U for USA, C for Canada and M
          for Mexico"<<endl;
          cin>>c;
          if(c=='U')
                    cout<<"USA"<<endl;
          else if (c=='C')
                    cout<<"Canada"<<endl;
          else if (c=='M')
                    cout<<"Mexico"<<endl;
          return 0;

}
```

Figure 9. The code for the first test question (selection).



```
Code Segment
char c;
cou<<"Enter U for USA";
cout<<"Enter C for Canada";
cout<<"Enter M for Mexico";
  cin>>c;
  if(c=="U")
    cout<<"USA";
    else if(c=='C')
    cout<<"Canada"<<endl;
    else if(c=='M')
    cout<<"Mexico"<<endl;
```

Figure 10. Example of the correct MTL visualization of the code and the (only) viable mental model.


Students' unviable mental model using MTL

Misconception (1): that each input option (U, C and M) must be an identifier representing a variable

Misconception (2): that each output coming from output stream as a message (USA, Canada and Mexico) is and identifier/variable

Misconception (3): that each possible output from output stream must first be in an input in a variable

Figure 11. Example of MTL visualization of unviable mental models.

5. Experiments

To test the effectiveness of MTL three-tier approach, two class experiments were conducted. The first experiment tested the viability of mental models acquired by novices. The experiment was carried out in University X (masked to avoid bias). The second experiment tested the ability of novices to avoid common errors in introductory programming. This experiment compared errors from a sample of students from University Y and University X.

Samples

With regard to the first experiment, a convenient sample of 2,322 university students was involved. Of these, 1110 constituted the treatment group, which comprised students admitted to University X in the academic year 2022/2023. The treatment group comprised 254 female students and 856 males. The control group, numbering 1,212 students, comprised the first-year students admitted to the same university in the academic year 2021/2022. This sample comprised 272 females and 730 males. Nineteen (19) students from the treatment group and eleven (11) students from the control group had prior exposure to programming in high school. This prior knowledge was ignored in this experiment since these numbers are too small to affect the outcome. All students had some prior experience with mobile phones and electronic calculators. The average age of the samples was 21 years.

With regard to the second experiment, the same experimental group of 1,110 students from University X were used. The control group comprised 600 students from University Y, where the visualization approach was not mandatory. Without regard to the difference in the questions of these two groups, a random sample of 600 examination scripts was drawn from the treatment group of University X, and a similar random sample of 600 scripts was drawn from the control group of University Y. Students' answers in the scripts were evaluated for fundamental errors as detailed in Table 2.

Materials

The materials that were used included programming textbooks, specifically those written with worked examples that were analyzed using MTL-RAM diagrams. In addition, students were encouraged to use books, such as those listed in Table 1, for self-reading and assignments. Students were encouraged to listen to lectures, such as from instructors on YouTube (Table 1).

The University X has four computer labs, each with 150 desktops, all loaded with Borland C++ compiler. These labs are accessible to all students for 14 hours, except on weekends. A random check established that at least one of the three students had a laptop. For the treatment group, in addition to the Boland C++ compiler, students were obliged to first dry-run the code segment (in assembly syntax), as depicted in Figure 3. The dry run was done using physical models (Figure 4). Every novice in the treatment sample was obliged to come up with physical models to mimic computer registers and RAM (Figure 4).

Concerning the control group, the timetable, books, learning materials, coverage, tutorials, and laboratory sessions are the same as those of the treatment group, except that the assembly code (Figure 3) and the corresponding physical models (Figure 4) were excluded. In addition, the MTL and animations were not mandatory, although these are present in some of the books and lecture videos.

Procedures

The two universities follow a 16-week semester, and the credit hours allotted for the programming course are 10. This means that lectures are allotted 2 hours, tutorials 2 hours, laboratories 2 hours, and individual study 6 hours, for 12 hours per week. Study weeks are 14, and two weeks are reserved for summative final examinations. This means that programming is studied for 168 hours. The first experiment was based on a test question about bifurcation or selection, where viable and unviable mental models were the basis of comparison.

The second experiment was based on the errors committed in the summative examination (Table 2). The errors were counted from the first two code questions attempted by the novices for statistical analysis.

For the treatment group, during the initial introductory session, which took 6 hours. In the first 40 minutes, novices were briefly introduced to computer organization, where it was shown that the computer memory is a combination of registers and RAM. As part of this experiment, memory hierarchy was avoided. Each student was tasked to perform some simple calculations (additions and subtractions) with a mobile phone.

After this introductory session, the remaining time of the initial two hours of lecture was carried out such that two worked examples using assembly language, as exemplified in Figures 3 and 4, were carried out. To improve comprehension, physical models similar to the one in Figure 4 were employed to mimic registers. Two examples were exhaustively used to mimic the RAM. One class activity where students were asked to construct their own scenarios on physical models and RAM diagrams for simple addition problems in assembly codes was conducted.

During the initial two hours of tutorials, more examples of simple calculators performing arithmetic operations, including subtraction and multiplication, were discussed with different data. These same codes were used in the initial two hours of the laboratory, where novices were allowed to write and compile the codes that were discussed and drawn during tutorials. After compilation in Borland C++ and in *Celiot* (Figure 8), these examples were explained using MTL RAM diagrams, as exemplified in Figures 5-7, 10, and 11.

For self-reading, after each topic, students were made to find a code from any book in the library and depict its behavior using RAM diagrams and physical models. For each programming aspect of sequence, selection, loop, arrays, functions, and file handling, each student was required to present one worked example (code), construct its corresponding physical model, depict the model using MTL RAM diagrams (Figures 5-7] and finally compile the code in *Celiot* and Boland C++. For each assignment, students were given one week for submission.

The control group was taught for the same length of time. Lectures, tutorials, and labs were organized such that the use of MTL and flowcharts was not mandatory, although the reference books, teaching notes, laboratory sessions, and work examples were the same. Although MTL and flowcharting visualization were used by the instructor, their use in labs, tutorials, and self-reading was not mandatory. Students were not asked to create MTL models, although they were taught and encouraged to use *Celiot* animators during lab sessions. Compulsory lectures, tutorials, and labs were finalized by the 10th week of study for both groups.

The first experiment

In the third week, after covering *selection*, the *selection* question was administered (Figure 9). Students were asked to use any means to show the dynamic characteristics (data movement) of the variables of the code and how the output was obtained. Students were allowed to use any means they found convenient (including verbal narrations) to answer the question.

The second experiment

Without regard to the difference in the examination questions, a random sample of 600 examination scripts was drawn from the treatment group of University X, and a similar random sample of 600 scripts was drawn from University Y as control sample. The treatment group was the same in the first experiment. From each script and the first two (coding) questions attempted by a novice, any error in the category listed in Table 2 (error due to an unviable mental model) was identified and counted.

6. Results and Discussion

Answers to the question in the first experiment were grouped into three categories: *MTL visualizations*, *verbal narrations*, and *other visualizations*, such as flowcharts and trace tables. Table 3 summarizes the categories of answers and the number of students who preferred such an approach.

Statistical results from the first experiment are summarized in Table 4 while results from the second experiment are summarized in Table 5.

As results from the first experiment in Table 4 show, it can be concluded that the treatment group had a significantly smaller number of unviable mental models compared with the control group. Specifically, in the control group, 687 out of 1,110 scripts, that is, 61.89% of the students, had errors due to unviable mental models, whereas in the treatment group, only 345 out of 1,212 students (28.47%) manifested unviable mental models. The relative risk is 2.17 with a *p*-value of less than 0.0001. This means that students in the treatment group were 2.17 more likely to form the correct mental model in comparison to the control group.

As results in Table 5 show, in the second experiment from the control group, 411 out of 600 scripts (68.50%) had errors committed due to unviable mental models. In contrast, from the treatment group, only 87 out of 600 scripts (14.50%) had such errors. The relative risk is 14.50, implying that students in the treatment group were 14.50 times more likely to avoid committing one or more of the errors listed in Table 2 in comparison to those in the control group. The *p*-value for the comparison between the control and treatment groups is less than 0.0001, indicating a highly significant difference in error rates.

These experimental results strongly suggest the MTL three-tier approach played a significant role in reducing unviable mental models and improving cognition among novices, and consequently reducing the failure rate of the subject.

Although *learning to program* and *learning a programming language* are interrelated and interdependent, it is obvious that learning to program is more than learning a certain programming language. As shown in the evaluation, due to the "language-trap," the majority of programming instructional design is biased towards covering the language syntaxes and semantics rather than imparting appropriate mental models.

Table 1. Evaluation of teaching and learning materials (syllabi, books, notes/videos, and examinations).

SN	The Institution	Exhaustive coverage of language syntax, (i.e. the 3 loop constructs)	Starting with output as opposed to variables, input, process	Consistent use of visualization (PV/animations) to discuss variables and their roles)
1	NCC	✓	NA	X
2	MIT	✓	NA	X
3	University of Karachi	✓	NA	X
4	MooC	✓	NA	X
5	The Open University	✓	NA	X
6	London School	✓	NA	X
7	Ahmadu Bello University	✓	NA	X
9	University of Nairobi	✓	NA	X
10	University of Cape Town	✓	NA	X
11	Delhi University	✓	NA	X
12	Dublin Institute of Technology	✓	NA	X
		Books		
1	Codding basics for beginners by Ryan Roffe, 2023	✓	✓	X
2	Computer programming for beginners by C. Konnors, 2023	✓	✓	X
3	Phyton programming for beginners by Kevin Wilson, 2024	✓	✓	X
4	Codding for beginners, Mike Mcgrath, 2015	✓	✓	X
5	Computer programming for beginners, Murali Chemuturi, 2018	✓	✓	X
6	C++ Programming: From Problem Analysis to Program Design, by Malik DS, 2017	✓	✓	X
		Videos/Notes/Lectures		
		Exhaustive coverage of language syntax,	Starting with output as opposed to variables, input, process	Use of visualization (PV animations)

1	https://programming-24.mooc.fi/	✓	✓	X
2	https://www.youtube.com/watch?v=EjavYOFoJJ0	✓	✓	X
3	https://www.youtube.com/watch?v=EjavYOFoJJ0	✓	✓	X
4	https://www.youtube.com/watch?v=K5KVEU3aaeQ	✓	✓	X
5	https://www.youtube.com/watch?v=4JzDttgdILQ	✓	✓	X
6	https://www.khanacademy.org/computing/computer-programming/programming/intro-to-programming/v/programming-intro	✓	✓	X

Examinations

		Exhaustive coverage of language syntax, (i.e. the 3 loop constructs)	Questions starting with the verb show/analyze/evaluate	Question demanding explicit depiction of memory (RAM)
1	NCC	✓	X	X
2	MIT	✓	X	X
3	University of Karachi	✓	X	X
4	MooC	✓	✓	X
5	The Open University	✓	✓	X
6	London School	✓	✓	X
7	Ahmadu Bello University	✓	X	X
9	University of Nairobi	✓	X	X
10	University of Cape Town	✓	✓	X
11	Delhi University	✓	✓	X
12	Dublin Institute of Technology	✓	✓	X

Table 2. Errors due to unviable mental models.

SN	**Topic areas**	**Question**	**Average time spent on the topic (minutes)**	**Number of examples**	**Examples of related errors**	**Number of errors counted from students**		**Description of the unviable mental model**
						Treatment (600)	Control (600)	

1	Variables, declaration, data inputting, outputting	Declare an integer variable called x and input data in it using assignment or cin>>	15-60	3-5	int x;
					x=4; followed by cin>>x; or
					int x; followed by i=0;
						24	119	1) Some novices fail to discern that cin>> and assignment (=) perform the same task of imputing.
								2) Others think that x can contain more than one value simultaneously
								3) Others fail to reference variables for their intended purpose

2	Outputting from variables when mixed with messaging	Consider the code segment: *char c; cin>>c;* *If (c==”U”)* *cout<<”USA”;* *else if(c==”K”)* *cout<<”UK”;* What is the output of this code?	15-60	2-4	U or K or U and K	46	134	Some novices fail to discern the source of a message when this is embedded in the code as an output resulting from a variable evaluation

3	Incrementing and decrementing a variable	*int i=0;*
		i=i+1; or ++i;
		What is the value of i?
			4-5	1-3	i=i+1; or or ++i;	23	150	Since this expression involves variable overwriting and incrementing/decrementing, some novices fail to assimilate this computing reality

4	Variable overwriting and copy retention	*int x, y; x=5; y=4;*
		x=y; cout<<x;
		cout<<y;
		What is the output from x?
			1-5	1-3	1. 5 is co-existing with 4 in x (5, 4)
					2. 5 is still alone in x
					3. x contains both 5 and the letter y
						45	160	Similar to 3

5	Variable-to-variable assignment	*int x, y; x=4; y=x;*
		cout<<y;
		what is the output from y?
			1-6 0-15	0-2	4. x or y or xy	56	166	Similar to 3

Some researchers have pointed out that the instructional design in introductory programming does not sufficiently emphasize on teaching such that correct mental models are the primary objective [12, 20, 21, 15-19, 26]. Syllabi for introductory programming, teaching materials, and techniques that are geared toward the formation of mental models are treated as secondary. The programming books, teaching materials, and notes show that beginning programming classes with *output* statements such as *cout<< "Hallo World*"; is a standard. As shown in this article, starting to teach programming with *declaration* (i.e., *int x),* followed by *input* (i.e., *cin >> x;/x = 4)*, and later followed by *output* (i.e., *cout << x)* is more productive since it covers both language features and mental models equally.

Due to the language-trap, even when some instructors use visualization tools, such as flowcharts, these are both sparingly used and wrongly employed as an end to themselves rather than a means for the formation of viable mental models [2, 7].

From the first experiment, statistical results are summarized in Table 3. From second experiment, results are summarized in Table 4. The central claim of this paper is that teaching and learning introductory programming has disproportionately been more about teaching and learning language features and less about mental models/computational thinking. It is further claimed that this language-trap is the cause of high CL and negative LEM and, subsequently, a high failure rate in programming. This language-trap is not easily detectable due to the fact that it is hidden in the historical emergence and convenient use of high-level languages in programming. Due to the language-trap, even most of the solutions that have been prescribed, including PVs, have been insufficient [20].

Table 3. Categories of answers and the number of students by group.

Category of answer	Total (N)	MTL visualization	Verbal narrations	Other visualizations	Right	Wrong
Control	1110	529	511	70	423	687
Treatment	1212	1187	25	00	867	345

Table 4. Proportion of students with errors counted from the first experiment by group.

Experiment group	Total (N)	Counted errors N (%)	Relative Risk, RR (95% CI)	P-Value
Control	1110	687 (61.89)	2.17 (1.97, 2.40)	<0.0001
Treatment	1212	345 (28.47)		

Table 5. Proportion of students with error counted from examination scripts by experimental group.

Experiment group	Total (N)	Counted errors N (%)	Relative Risk, RR (95% CI)	P-Value
Control	600	411 (68.50)	14.50 (3.86, 5.78)	<0.0001
Treatment	600	87 (14.50)		

The notion that learning to program is synonymous with learning a programming language is prevalent among programming instructors, partly due to the oversized role of high-level languages in introductory programming.

MTL three-tier approach

Instead of the traditional approach, this study proposes an instructional design that combines initial codes in low-level language, visualized by both MTL physical models (Figure 3) and MTL PV, followed by worked examples in high-level language consistently visualized by MTL PV.

This novel instructional design allows for a multisensory approach to teaching and learning programming. As the results from the two experiments show, the performance of the treatment group was significantly better compared to the performance of the two control groups. Some PV skeptics contend that PVs in programming have not been widely accepted due to the need for drawing and re-drawing. The fact that some researchers and instructors are complaining about redrawing implies that to them, PVs are not used as a means to create correct mental models but rather as vehicles to understand a programming language instead. If mental models were to be treated as a mandatory aspect of teaching and learning programming, then drawing would be a mandatory aspect of all programming syllabi.

As demonstrated in both experiments, in order for the effort to be directed at mental models, the MTL PV uses familiar examples of low-level syntax combined with MTL visualization, which is tied to mental models. As exemplified in Figures 5-8, MTL PV can be employed throughout lectures, tutorials, labs, and self-reading for every aspect of programming. MTL diagrams and physical models are just used as instruments to enforce the mental model-building process. Constant and mandatory use of these visualizations do not demand more time or any alteration of programming syllabi because they are just part of the general discussion [9, 16]. With an effective focus on mental models using a combination of low-level syntax and MTL PV, a positive LEM is guaranteed and, therefore, the possibility for more novices to comprehend the subject.

As revealed in the first experiment, unviable mental models, such as the ones depicted in Figure 11, indicate that even after spending numerous hours teaching novices, the majority of them fail to form viable mental models concerning the combination of variable declaration (int/char), data input, data overwriting, and message output. Only if a novice is able to portray the correct flow of events, as depicted in Figure 10, can it be concluded that the student has acquired viable mental models concerning a scenario such as the one represented by the question represented in Figure 9. If a student has failed to form these correct mental models, then his or her LEM has come to a dead end, and it is futile for such a student to continue classes concerning loops and array functions, among others, because all these depend on the viable mental models. Normally, students who have not acquired the required mental models for basic concepts are allowed to continue their studies as the focus is *to cover the language syntax* instead of consolidating mental models. A similar argument can be made in the case of the second experiment.

Machine or assembly syntax and mental models

There exists a cause-and-effect relationship between unviable mental models and errors that a novice commits [12, 25]. As may be found from literature and experiments, writing correct code in machine or assembly language goes hand in hand with having viable mental models [2, 5, 10]. This is because the language used to communicate with the computer is the language of the computer itself. It is therefore inferred that anyone learning programming using machine or assembly language is obliged to first direct effort on mental models and later on the language features [5]. However, when one is using a high-level language, the reverse is true. The instructors emphasize more on syntax and less on mental models.

High-level language introduces abstraction and more cognitive load through variables and their different types, which increases the possibility for a novice to form alternative mental models that are unviable [4]. Although low-level languages have tedious syntaxes, their use at the initial level of learning programming is easier and more straightforward than in high-level languages (Figures 3 and 4). Using low-level syntax during the initial stage of programming (*data storage space*, *data input, data processing, and outputting of results from inside the machine registers*) requires fewer concepts and fewer associations than it does with high-level languages. In addition, low-level syntax bears a closer analogy with calculators (which are more familiar) than high-level languages and their *hello world* examples. This focus on mental models using concrete, familiar concepts (calculators and rectangles) reduce CL and, therefore, increases the chance of positive LEM. However, as the lessons progress to more complex aspects, such as selection and loops, low-level syntax must be replaced with high-level syntax because this is much easier to use and construct. Since at this stage the novice will have formed initial viable mental models, the LEM is maintained, and learning is not negatively affected.

Take, for example, the assembly code depicted in Figure 3. The number (1) is a label that instructs the computer where the execution should begin. This is a detail that can be ignored in the visualization because it will be part of any code. The code segment is visualized using an MTL physical model (Figure 4). *MOV exa, 4* is an instruction that stores integer 4 in the register exa; *MOV exb, 7* instructs the computer to store integer 7 in the register exb. *ADD exa, exb, exc* instructs the machine to add the content of *exa* and that of *exb*. Finally, the output is stored in exc, which is again a fundamental concept that must be understood without ambiguity. Actually, telling a novice that the computer has in-built storage space (registers), as depicted in Figure 4, is more familiar than telling a novice about the declaration of variables, types of variables, and accessing them, or *cout* << Hello *world.”*

Not only are these concepts responsible for increased CL and consequently the formation of unviable mental models, but due to the language-trap, variables attract instructors and learners to engage in data types prematurely, further increasing the CL while reducing resources from the early formation of viable mental models.

Consider the elementary assembly code in Figure 3. The visualization of this code using MTL to portray the basic programming mental models is more familiar to novices compared to its version in high-level languages, as demonstrated in Figures 2 and 4.

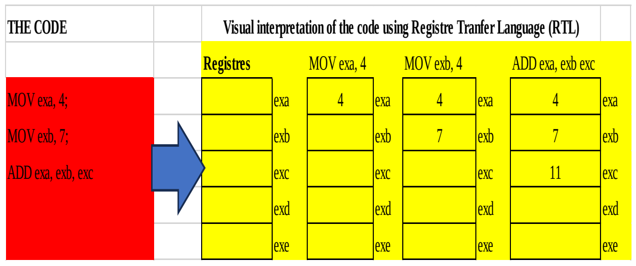


Figure 12. Visualization of the assembly code using RTL.

The concept of variables and variable declarations, with their associated data types, may seem simple to an expert. Nevertheless, as demonstrated, in the experiments, variables and their associated roles constitute the source of much confusion that most novices have when attempting to form mental models.

```
int x;

int y;

int z;

x=4;

y=7;

z=x+y;
```

Figure 13. A high-level version of the code in Figure 12.

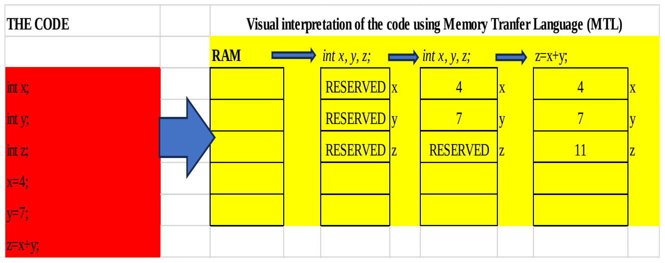


Figure 14. MTL visualization of the code segment.

Some authors conclude that PVs, if massively applied in classrooms, can reduce the HCL and positively impact learning and teaching programming [5, 12-16]. As argued in this paper, the use of PVs, such as MTL, combined with low-level syntax and high-level language codes, is part of the solution. Numerous PV proponents for learning and teaching programming have reported positive results [16, 18]. Nevertheless, encouraging as these results may be, the language-trap is one reason the use of PVs and mental models has not consolidated itself in mainstream teaching and learning programming. A PV diagram is just a means and not an end in itself. An effective PV scheme must be simple both in vocabulary and syntax, as demonstrated throughout this study. It does not require elegance. In fact, if PVs are effectively used in the opening of the topics, their use for complex aspects, such as functions and file handling, may not be necessary.

Program visualization (PV) and its relative success in assisting comprehension in programming

The PV approach in programming is as old as programming itself. The oldest and among the widely used PV is the flowchart. Together with reducing CL, flowcharts were meant to aid communication between the analyst and the programmer. Flowcharts were introduced to teaching and learning programming at the same time as high-level languages were introduced. As a result, the use of flowcharts in programming was influenced by the language-trap. Flowcharts, by their nature, are more useful at conveying language features than concrete mental models [9]. As argued by Esmenger [2], flowcharts have been ineffective both in the reduction of CL and in professional programming. On the contrary, modern PV schemes, such as *Celiot*, *Jeliot 3*, *Plan Ani*, and *MTL*, have been reported to reduce CL and, by implication, the failure rates [17, 22].

Mtaho and Mselle [18] and Saha et al. [25] argued that the adoption of PVs and mental models in programming has had limited success due to their failure to allow for students' engagement and the time required to incorporate them into instruction. However, as demonstrated in this paper, the tacit or covert influence of high-level syntax in programming (language-trap) remains dominant in most PV applications, rendering them ineffective as means for building mental models.

Mselle and Ishengoma [16] reported that students who found programming challenging but manageable were the most positive about using visualizations, while the strongest and weakest students were less impressed. In fact, any student who is using PV as a means to validate mental models will be positive about the use of PV. Any student who perceives programming as a study of a language will find PVs to be an added burden and, therefore, a waste of time. The same findings were reported by Pelánek and Effenberger [18] and Sun and Zhou [23], who showed that the most successful novices were those who preferred to visualize their codes, while those who preferred alternative means had unviable mental models. This conforms with the results from this study and the correlation between the desire to use visualization and success in forming viable mental models.

Using results from this study, it is suggested to incorporate a mandatory use of PVs and mental models in the instructional design for introductory programming. Since mental models are pivotal in introductory programming, it is important to allow a novice to advance to a subsequent stage only after first proving that he or she has acquired appropriate mental models in each prior stage. Once appropriate mental models have been depicted at each stage with the assistance of low-level code combined with MTL PVs, the same can be used as a means to assist a novice in the process of learning other concepts, such as data types and other language features.

As demonstrated in this study, novices can be introduced to the unfamiliar aspect of variable overwriting, first by de-learning "co-existence" in the computer register and later in RAM through PVs and mental models (Figures 4-6). Similarly, left-to-right evaluation is familiar to learners due to their experience in mathematics. To de-learn this and adopt right-to-left evaluation requires a lot of time and extensive use of PVs. It is hard to find an instructional design that takes these details seriously. Issues such as the de-learning of co-existence, incrementing, and the right-to-left operation property of the = operator must be given more emphasis in the beginning than they are currently done, where coverage of the language syntax instead of mental models seems to be the main concern.

7. Conclusion and Recommendations

Introductory programming is a subject with unusual HCL [1, 10, 12]. The subject requires mastery of mental models and language features both of which carry HCL [22, 23]. From documents evaluation in this research, it was revealed that instruction effort and research on how to reduce HCL in programming has been disproportionally concentrated on language features and less so on mental models.

Although the PV approach, which emphasizes both mental models and language features has been advocated for over five decades now [13, 14, 15-20], this approach has yet to receive wide acceptance among programming instructors. This research has shown that the language-trap in programming instructional design is responsible for the low uptake of PVs in instructional design among programming instructors. The language-trap has a self-reinforcing effect in books, teaching notes, examinations and even research on how to reduce HCL in introductory programming.

In this research, using DSRM, a novel approach based on PV in combination with MTL and low-level syntax and high-level syntax, was designed, implemented, and tested. Results from the first experiment show that students who were instructed using the MTL-PV three-tier approach were more successful in showing the exact mental models for the *selection* question than was the case for students who were taught in the traditional approach. Furthermore, results from the second experiment showed that students who were taught using the MTL-PV three-tier approach were less likely to commit the common programming errors (Table 2).

According to some authors [7, 12, 20, 26], by strategizing on an instructional approach that combines both audio and visual channels, the CL is reduced. As demonstrated in this study, the use of low-level syntax in combination with MTL PV at the initial stage of a program was made possible and less demanding cognitively.

This study has shown that using a limited assembly syntax in combination with MTL PV to emphasize mental models right at the beginning of the course is more effective than exclusively relying on high-level languages.

Worldwide, governments and educational institutions are taking measures to introduce programming at lower educational levels. This implies that more students will be required to study the subject. Results from this study may be useful to instructional designers. Before making a decision, a few questions are worth asking: how much effort should be directed towards mental models? Is it worth for an instructor to continue teaching higher-level topics, such as selection and loops, before ensuring that novices have correct mental models concerning the concept of variables, data inputting, processing, storage, and retrieval?

This study was carried out in one country using two groups of students admitted to just two universities. For generalization, studies of this type must normally be carried out across different countries. Despite this limitation, there is a reason to do more experiments on the viability of partially using low-level programming in combination with PVs and animations at the beginning of programming for the purpose of building viable mental models and later transitioning to high-level syntax without abandoning PV.

This study does not advocate the replacement of high-level languages with assembly code or mental models in introductory programming. As shown in all worked examples, high-level languages remain at the core of learning and teaching programming. This article calls upon the programming community, instructors, curricula developers, book writers, students, and researchers to embrace mental models and liberate themselves from the language-trap by exploring the possibility of a partial return to low-level languages combined with memory PVs along with high-level languages for programming instructional design.

It has to be pointed out that C++ is used just as one of the high-level programming languages. Examples in any other high-level language, such as C, Java and C#, would neither change the implementation of the MTL three-tier approach nor the findings of this research.

CONTRIBUTIONS OF CO-AUTHORS

Leonard Mselle	ORCID: 0000-0001-6326-6437	Conceived the idea, conducted experiments, and wrote the paper

REFERENCES

[1] G. Ashman, S. Kalyuga and J. Sweller, *Problem-solving or explicit instruction: Which should go first when element interactivity is high?* Educational Psychology Review, **32**(1), p. 229–247, 2022. [Online]. Available: https://doi.org/10. 1007/s10648-019-09500-5.

[2] N. Esmenger, *Multiple meanings of a flowchart.* Information & Culture: A Journal of History. **3**(51), 2016.

[3] S. Fincher et al. *Notional Machines in Computing Education: The Education of Attention,* ITiCSE-WGR '20, June 17–18, 2020. Trondheim, Norway.

[4] S. Garces et al. *Engaging students in active exploration of programming worked examples*. Education and Information Technologies, **8**(3), 2022.

[5] D. Morais da Silva et al. *Ada Lovelace's Legacy for Computation History.* Science, Technology & Public Policy. **3**(2), p. 8-13. 2019, doi: 10.11648/j.stpp.20190302.11.

[6] R. Bornat, S. Dehnadi, and Simon, *Mental Models, Consistency and Programming Aptitude*. ACE '08: Proceedings of the tenth conference on Australasian computing education, **78**, 2008.

[7] S. Cammeraat, G. Rop and B. B. de Koning, *The influence of spatial distance and signaling on the split-attention effect.* Computers in Human Behavior, **105**(106203), 2020. [Online]. Available: https://doi.org/10.1016/j.chb.2019.106203.

[8] J. C. Castro-Alonso, P. Ayres and J. Sweller, *Instructional visualizations, cognitive load theory, and visuospatial processing.* In J. C. Castro-Alonso (Ed.), Visuospatial processing for education in health and natural sciences, Springer, pp. 111–143, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-20969-8_5.

[9] J. Sorva, V. Karavirta and L. Malmi, *A review of generic program visualization systems for introductory programming education.* ACM Trans. Comput. Educ. **13(**4), 2013. [Online]. Available: http://dx.doi.org/10.1145/2490822.

[10] A. Korbach et al. *Should learners use their hands for learning? Results from an eye-tracking study.* Journal of Computer Assisted Learning, **36**(1), 2020. [Online]. Available: https://doi.org/10. 1111/jcal.12396.

[11] A. Robins, *Learning edge momentum: a new account of outcomes in CS1*, Computer Science Education, Taylor and Francis, 2010.

[12] V. Dentamaro et al. *Human activity recognition with smartphone-integrated sensors:* A survey. *Expert Systems with Applications*, p. 123–143, 2024.

[13] P. E. Dickson, N. C. Brown and B. A. Becker, *Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices*, ITiCSE '20, June p. 15–19, 2020, Trondheim, Norway.

[14] J. C. Castro-Alonso et al. *Five Strategies for Optimizing Instructional Materials: Instructor- and Learner-Managed Cognitive Load*. Educational Psychology Review, **33**(1) 2021. [Online]. Available: https://doi.org/10.1007/s10648-021-09606-9.

[15] M. Masoud and L. Mselle, *Using Spiral Approach in teaching programming to Novices to improve problem composition and algorithmatization*, The 3rd International Virtual Conference on Advanced Scientific Results, p. 25– 29, 2015. [Online]. Available: www.scieconf.com

[16] L. Mselle and F. Ishengoma, *Memory Transfer Language as a Tool for Visualization-Based-Pedagogy*. Education and Information Technologies, Springer Nature, **27**(9), 2022.

[17] A. B. Mtaho and L. Mselle, *Difficulties in Learning Data Structures Course: Literature Review* Journal of Informatics, **4**(1), p. 26-55, 2024, [Online]. Available: https://doi.org/10.59645/tji.v4i1.136.

[18] R. Pelánek and T. Effenberger, *The Landscape of Computational Thinking Problems for Practice and Assessment.* ACM Transactions on Computing Education **23(**2), 2023.

[19] S. Phlix, *How can self-learners learn programming in the most efficient way? A pragmatic approach.* Submitted in partial fulfillment of the requirements for the degree of Master of Science in Management / Grande Ecole Diploma at HEC Paris. Business Studies, University of Mannheim, 2016.

[20] E. F. King, *Bit by Bit: A Graphic Introduction to Computer Science*, Edition Stanford University Press, NY, 2024.

[21] R. Kraleva, V. Kralev and D. Kostadinova, *Investigating some programming languages for children to 8 years.* International Scientific and Practical Conference "WORLD SCIENCE" South-West University "Neofit Rilski" Bulgaria, Blagoevgrad, 2016.

[22] N. Sundararajan and O. Adesope, *Keep it coherent: A meta-analysis of the seductive details effect.* Educational Psychology Review, **32**(3), 2020, [Online]. Available: https://doi.org/10.1007/s10648-020-09522-4.

[23] L. Sun and L. Zhou, *Does text-based programming improve K-12 students' CT skills? Evidence from a meta-analysis and synthesis of qualitative data in educational contexts*. *Thinking Skills and Creativity*, **49**(1), 2013.

[24] W. Leahy and J. Sweller, *The centrality of element interactivity to cognitive load theory*. In S. TindallFord, S. Agostinho, & J. Sweller (Eds.), Advances in cognitive load theory: Rethinking teaching. Routledge. p. 221– 232, 2020. [Online]. Available: https://doi.org/10.4324/9780429283895-18.

[25] A. Saha et al. *A survey of machine learning and meta-heuristics approaches for sensor-based human activity recognition systems.* Journal of Ambient Intelligence and Humanized Computing **15**(1) p. 29–56, 2024.

[26] T. van Gog, V. Hoogerheide and M. van Harsel, *The role of mental effort in fostering self-regulated learning with problem-solving tasks.* Educational Psychology Review, p. 1055–1072, **32**(4), 2020. [Online]. Available: https://doi.org/10. 1007/s10648-020-09544-y