# LMC + Scratch: A recipe to construct a mental model of program execution

NOMAN JAVED, London School of Economics and Political Science, United Kingdom
FAISAL ZEESHAN, Namal Institute Mianwali, Pakistan

Understanding how programs execute is one of the critical activities in the learning journey of a programmer. A novice constructs a mental model of program execution while learning programming. Any misconceptions at this stage lead to the development of a discrepant mental model. If left untreated, learning in advanced subjects like data structures and compiler construction may suffer. One of the ways to prevent the situation is carefully and explicitly unveiling the details of program execution. We employed Little Man Computer (LMC) for this purpose. Its interactive visual interface helped them internalise how software interacted with the hardware to achieve the programmer's objective. After spending a few sessions on the programming of LMC, we moved to Scratch. Scratch is a much higher-level language than the LMC assembly. So, while introducing Scratch programming constructs, we mapped the LMC equivalents of these instructions. The strategy helped evade several misconceptions by developing a deep understanding of the program execution model. It also served as a building block for introducing other concepts like state, abstraction, the need for higher-level languages and the role of compilers etc. We tried this approach in an Introduction to Computer Science module where most students had zero or very minimal exposure to programming. We received positive feedback from students and other fellow teachers teaching in the subsequent semesters.

Additional Key Words and Phrases: Little man computer, Scratch, CS1, Novice Programming

## 1 INTRODUCTION

Programming is a complex phenomenon that requires the learner to understand the problem, discover an algorithmic solution, and then express it in programming language constructs. The boundary between the algorithmic solution and its expression in a programming language is obscure. Therefore, the choice of programming language can impact the algorithmic thinking of programmers. Higher-level languages provide sufficient abstraction and allow the programmers to operate at the application level. On the other hand, lower-level languages unveil the underlying computational architecture. It is much easier for a novice to focus on the application side while ignoring the underlying complexities. That is why most of the introductory programming modules expose students to some high-level programming language [7].

Working at higher levels of abstraction is good for focusing on the application logic. However, problems arise when teachers and students ignore the computational aspects of program execution. This ignorance may sometimes result in grave errors, especially those related to the runtime behaviour of the program [5]. The situation is even more critical when these students have no clue why their programs generate such errors. One can avoid the problem by revealing the other side of the picture, the execution model of a program. Some of the teachers do so by starting with assembly language programming [4]. It makes the learning of the program execution model implicit, as one can not write an assembly program without knowing the architectural details.

Both the approaches are not problem-free. Starting with a high-level language puts too much emphasis on the problem and algorithmic levels. A beginner may develop a tendency of ignoring the lower-level details. On the other hand, beginning with the assembly level may distract the student from the logic building and shift their focus on understanding the architectural concerns. So, what is the right approach to teaching students to keep in mind both sides of programming without overwhelming them? A linked question is what is the right time for doing this. In this paper, we propose a strategy to answer these questions.

Authors' addresses: Noman Javed, London School of Economics and Political Science, Houghton St, London, United Kingdom, WC2A 2AE, n.javed3@lse.ac.uk; Faisal Zeeshan, Namal Institute Mianwali, Mianwali, Pakistan, faisal.zeeshan@namal.edu.pk.

The traditional way to deal with this issue is to start with a high-level language and focus on the algorithmic side in the beginning. For the execution side, students will have to wait for a few semesters. They will learn it in subsequent modules like computer organisation and architecture module. They will study the concepts of the fetch-decode instruction cycle, the memory hierarchy and the other relevant concepts. Moreover, Compiler construction will help them fill the gaps by explaining the nitty-gritty of the translation process. There are two problems with this approach. The first one is that the students develop a habit of looking at one side of the picture. They start ignoring how computer perceives their instructions which also relates to the issue of not comprehending the bigger picture of how software executes over the hardware to accomplish the prescribed task. The second issue is the poor understanding of the concept of abstraction. Why a programmer must think about it, and what is the cost of providing it?

We propose a bottom-up strategy to deal with this issue. We start by discussing the architecture of a computer using the Little man computer (LMC). We then use the LMC assembly language to construct simple programs and learn how these programs execute over LMC. These interactive animations aid the construction of a mental model of program execution. A mental model means the internal representation of a concept that promotes logical reasoning [1]. So, by a mental model of program execution we mean, how a student thinks a machine will execute a program. We then switch to Scratch, a very high-level visual programming language. This transition from LMC assembly to Scratch allows us to discuss the role of abstraction. In addition to that, while introducing the constructs of Scratch, we regularly refer back to the equivalent code in the LMC assembly code. This approximate translation strengthens the mental model of the program execution. It also helps us complete the whole programming picture by discussing the role of compilers, the ease provided by the abstraction and why it comes up with a cost.

## 2 LITTLE MAN COMPUTER

The Little Man Computer is a simplistic model of the computer architecture created by Dr Stuart Madnick in 1965 [2]. Many teachers use it to introduce the concepts of Von Neumann computer architecture. It models computer architecture by placing a little man inside a room containing 100 mailboxes labelled from 0 to 99. These mailboxes are analogous to the concept of computer memory. The little man can interact with the outside world through two boxes labelled Input and Output. He can perform simple addition and subtraction with the help of a calculator, called an Accumulator, placed on a table in the centre of the room. In addition, a resettable program counter is also at his disposal. The program counter points to a mailbox location and is usually incremented by one. However, in some situations, the little man can set it to point towards a different memory location. The complete instruction set of the LMC is presented in table 1.

Little man can do nothing other than the things mentioned in the instruction set. He starts execution by opening the mailbox zero, labelled as 000, and proceeds as per the instruction contained in it. Since, in Von Neumann architecture, data and instructions both take place in memory, the mailboxes of LMC do the same. So, it is the programmer's responsibility to design the sequence of instructions cautiously. The little man himself is not intelligent enough to differentiate between data and instructions. He may bog down if the mailbox contains data in place of instruction or vice versa. If everything goes fine, he fetches instructions and data from the mailboxes, one by one, and does whatever is prescribed.

Many people have developed the simulators of LMC over the years. Usually, there is a program input area for the programmer to write the assembly code. This assembly code is loaded to the mailboxes of LMC instruction by instruction. Programmers can create labels for readability purposes. However, little man remains oblivious to these labels. To him, these labels are just the number of mailboxes. Using the instruction set of the LMC, programmers can write code to solve various problems. Since both data and instructions are present in the mailboxes, programmers

| Code | Name | Description |
|---|---|---|
| 0 | HLT | Stop execution |
| 1xx | ADD | Add the contents of the memory address to the Accumulator |
| 2xx | SUB | Subtract the contents of the memory address from the Accumulator |
| 3xx | STA or STO | Store the value in the Accumulator in the memory address given. |
| 4 | | Not in use |
| 5xx | LDA | Load the Accumulator with the contents of the memory address given |
| 6xx | BRA | Branch - use the address given as the address of the next instruction |
| 7xx | BRZ | Branch to the address given if the Accumulator is zero |
| 8xx | BRP | Branch to the address given if the Accumulator is zero or positive |
| 9xx | INP or OUT | Input or Output. Take from Input if address is 1, copy to Output if address is 2. |
| | DAT | Used to indicate a location that contains data. |

Table 1. LMC - Instruction Set

can overwrite the content of any mailbox. Thus, in this way, one can effectively write a self-modifying program by replacing an instruction with another. Writing and executing programs for LMC promotes the construction of the mental execution model of a program. It does so by animating the fetch-execute instruction cycle. In this way, it helps students construct a better conceptual image of the interaction of the software with hardware to achieve the prescribed task.

## 3 SCRATCH PROGRAMMING

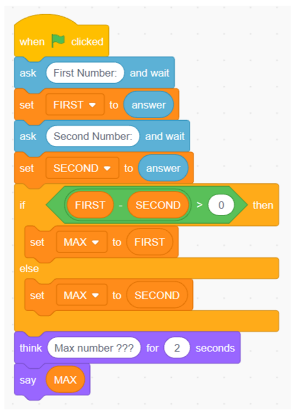
Scratch is a visual programming language developed at MIT Media Lab [6]. It promotes the teaching and learning of programming to young kids by replacing the textual syntax with a block-based environment. A program in Scratch is a composition of these blocks. Programmer can drag and drop these blocks on the stage area. After arranging blocks in the desired order, one can hit the green flag to start program execution. The impact of program execution is usually apparent on the Sprite, a visual character, which acts as per instructions. So, you can create all sorts of animations by issuing commands to a single or multiple sprites.

Scratch offers two prominent advantages over the other languages. The first one is liberating programmers from the learning of syntax by virtue of its block-based nature. The visual appearance of blocks provides clues to programmers about their connection with the other blocks. The second key advantage is the availability of instant feedback. Scratch offers this opportunity through the animation of Sprite using block instructions. It thus enables rapid prototyping by identifying and fixing the errors immediately. Rapid prototyping is especially useful for novice programmers, as it provides immediate feedback rather than waiting for the complete execution to finish.

## 4 METHODOLOGY

Before jumping on the methodological details, we first establish the context in which we were operating. We applied this strategy in a module titled Introduction to Computer Science offered in parallel to an introductory programming module. These modules were part of a computer science undergraduate program in a Pakistani institute where most students had little or no computing background. One of the objectives of this module was to introduce students to the basic concepts that are critical in computer science. These will act as the foundation stones on which the building of computer science will stand. So, the role of this module was to complement the learning in the programming module.

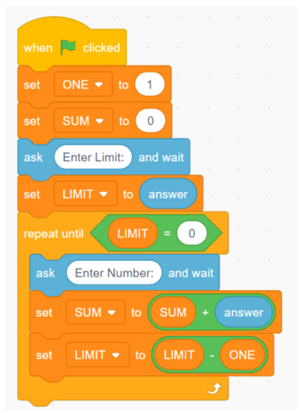
```
        INP
        STA FIRST
        INP
        STA SECOND
        LDA FIRST
        SUB SECOND
        BRP MAX
        LDA SECOND
        BRA DONE
MAX     LDA FIRST
DONE    OUT
        HLT
FIRST   DAT
SECOND  DAT
```



```
when [flag] clicked
ask (First Number:) and wait
set [FIRST] to (answer)
ask (Second Number:) and wait
set [SECOND] to (answer)
if <((FIRST) - (SECOND)) > (0)> then
    set [MAX] to (FIRST)
else
    set [MAX] to (SECOND)
think (Max number ???) for (2) seconds
say (MAX)
```

(a)

```
        INP
        STA LIMIT
LOOP    INP
        ADD SUM
        STA SUM
        LDA LIMIT
        SUB ONE
        STA LIMIT
        BRZ TOTAL
        BRA LOOP
TOTAL   LDA SUM
        OUT
        HLT
NUM     DAT
LIMIT   DAT
ONE     DAT 001
SUM     DAT 000
```



```
when [flag] clicked
set [ONE] to (1)
set [SUM] to (0)
ask (Enter Limit:) and wait
set [LIMIT] to (answer)
repeat until <(LIMIT) = (0)>
    ask (Enter Number:) and wait
    set [SUM] to ((SUM) + (answer))
    set [LIMIT] to ((LIMIT) - (ONE))
```

(b)

Fig. 1. Mapping of LMC and Scratch programs. Left hand side presents the mapping of conditional statements and the right hand side shows the mapping of repetition.These mappings are the not the exact translations from Scratch to LMC assembly. However, they are approximate enough to point out the relevant programming constructs on both sides.

One of the authors has several years of teaching experience in computer science. He has observed several misconceptions in students during these years. Most of these misconceptions stem from the absence of a clear understanding of how software interacts with the hardware to fulfil programmers wish. He traced back the problems in the construction of mental models to the introductory programming modules. Since there were two modules in the first semester of the undergraduate program, we decided to use the Introduction to Computer Science module. Another reason for this choice was the increased focus of the programming fundamentals module on data programming, and we did not want to disturb that.

We started with the introduction of computer architecture. We used a web-based simulator of LMC [3] for this purpose. After explaining and demonstrating the role of each part and their interactions, we focused on exploring assembly language instructions. Through these instructions, students learned to issue commands to the little man to perform their tasks. This marked their first step in the programming world. They understood the meanings of individual instructions and learned to combine them to form a complete program. LMC brought several advantages at this stage because of its simplicity, small instruction set and fetch-execute cycle's animation. It permitted the students to interactively explore the interactions between different parts of a computer system to execute an instruction. We spent a couple of weeks constructing simple assembly language programs. Along the way, we introduced the concept of a variable, the idea of a conditional statement using branches, and the repetitive execution of a set of instructions. We considered these points as anchor points that we will refer to later while covering the same constructs in high-level languages.

After LMC, we introduced the Scratch programming language. It was significantly different from the LMC's assembly. After learning simple programs in Scratch, they knew that they could interact with the computer system at multiple levels as low as the assembly and as high as the Scratch. Scratch was relatively easy and attractive than the LMC to them. They started developing a sense of how distancing from hardware details made programming relatively easy and allowed them to focus more on the problem. So to introduce the idea of translation, while introducing the constructs of

Scratch we started referring back to the equivalent LMC code as shown in figure 1. These mappings refined their mental models and linked the two ends of the programming: the application end and the computational end. The outline of our strategy is presented below:

1. Little man computer
   - Introduction, Fetch and execute model
   - Labels, Variables
   - Conditionals
   - Repetition
2. Scratch
   - Introduction, Animating sprite
   - Basic operations, using variables
   - Conditional blocks
   - Repetition

## 5 RESULTS AND FEEDBACK

We tried this strategy only twice in a computing fundamentals module offered in the first semester of a computer science undergraduate program. We had not gathered any formal conclusive evidence. However, in light of the informal feedback received from the students and the fellow teachers, we want to share it with the larger community. The quality enhancement cell gathers feedback from students twice a semester: once around the midterm and once at the end of the semester. We received very positive feedback from the students in all these evaluations. There was no significant difference between this feedback and the feedback of the previous years. So, we can safely conclude that our strategy did not create any negative ripples in terms of students satisfaction. A more formal way to test the validity of our methodology is to compare the results in subsequent programming modules. Unfortunately, we could not do so because the earlier and the later batches were on different curricula. However, the informal feedback from the colleagues teaching Object-oriented programming and data structures was encouraging. They praised the performance of these students in both these modules.

## 6 DISCUSSION

The primary goal of introducing LMC followed by Scratch and creating their equivalency mapping is to facilitate the construction of a mental model of program execution without overwhelming them. We noticed certain added advantages. One of them is seeing abstraction in action. They experience the complexity and limitation of programming at the assembly level. The transition towards Scratch feels empowering and highlights the benefits of hiding complexity. The equivalency mapping makes sure to establish the connection between the two extremes. They are now better placed to understand the penalty incurred by abstraction because of the translation process. This also links to the understanding of the performance of code. As they have witnessed the computational steps involved in the execution of an instruction, they develop a sense of code performance in terms of the number of steps required. So, in our opinion, this strategy adequately prepares them for learning in subsequent modules.

At this point, we would like to issue a word of caution for the practitioners. We deliberately made it part of the Introduction to computer science module rather than the Programming Fundamentals module. As both these modules were offered concurrently, we thought to complement the programming module using this strategy. One can try to embed it in a programming module by keeping an eye on the time constraints. It took us four weeks, containing eight lectures and four lab sessions, to properly execute this module. And some of the students were complaining about the fast pace at which we were going. So, you can embed it in a programming fundamentals module if you think you can reserve that much time at the start.

There is an alternative to this strategy as well. Rather than starting with LMC, start with Scratch and then move down to the LMC level. We have not tested this alternative, but we think it is worth investigating and comparing the two will give further insights into the students' preferences. We opted for the other approach as we structured the whole module in a bottom-up fashion.

## 7 CONCLUSION AND FUTURE WORK

We presented a strategy of starting with LMC followed by Scratch to facilitate mental model construction of program execution. We observed that by learning LMC programming, students developed a good mental model of program execution. Introducing Scratch after LMC landed them in the real world in which they operate most of the time. The mapping of Scratch and LMC constructs connected the two worlds and exposed the benefits of abstraction. To evaluate our strategy, we gathered informal feedback from the students and colleagues that was positive. So, we decided to share it with the larger community. However, we wish to conduct formal experimentation to validate it further. One of the ways to test the validity is through purposefully designed assessment. Another possibility is to measure the performance of these students in subsequent modules like data structures and algorithms.

We would also like to validate the strategy using semantic waves theory. Semantic waves theory suggests modelling the conceptual journey using a wave structure. First, the teacher introduces the topic at an abstract level. The next level is simplifying the problem, in terms understandable by a novice, to facilitate the construction of analogical relationships for deep understanding. In the latter half, the teacher repacks and returns to the abstract level. The wave-like movement alternating in the complex and comfortable zones promotes the learning of attacking complex problems. Novices do so by relating the complex problem to the familiar ones, developing a solution, and applying it to solve the original problem. So, we plan to validate our strategy using this semantic wave theory in future.

## REFERENCES

[1] Mordechai Ben-Ari. 2001. Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching* 20, 1 (2001), 45–73. https://www.learntechlib.org/p/8505

[2] Irv Englander. 1996. *The Architecture of Computer Hardware and Systems Software: An Information Technology Approach.* John Wiley and Sons, Inc., USA.

[3] Peter Higginson. [n. d.]. Little man computer - CPU simulator. https://www.peterhigginson.co.uk/LMC/

[4] Chien-fei Chen Joseph Foy and Erin James Wills. 2014. Integrating Assembly Language Programming into High School STEM Education (work in progress). In *2014 ASEE Annual Conference & Exposition.* ASEE Conferences, Indianapolis, Indiana. https://peer.asee.org/20660.

[5] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying Student Misconceptions of Programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10).* Association for Computing Machinery, New York, NY, USA, 107–111. https://doi.org/10.1145/1734263.1734299

[6] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[7] Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. 2018. Language Choice in Introductory Programming Courses at Australasian and UK Universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18).* Association for Computing Machinery, New York, NY, USA, 852–857. https://doi.org/10.1145/3159450.3159547