Computational Thinking in *Patch*

Hasan M. Jamil

Department of Computer Science
University of Idaho, USA
jamil@uidaho.edu

Abstract. With the future likely to see even more pervasive computation, computational thinking (problem-solving skills incorporating computing knowledge) is now being recognized as a fundamental skill needed by all students. Computational thinking is conceptualizing as opposed to programming, promotes natural human thinking style than algorithmic reasoning, complements and combines mathematical and engineering thinking, and it emphasizes ideas, not artifacts. In this paper, we outline a new visual language, called *Patch*, using which students are able to express their solutions to eScience computational problems in abstract visual tools. *Patch* is closer to high level procedural languages such as C++ or Java than Scratch or Snap! but similar to them in ease of use and combines simplicity and expressive power in one single platform.

Keywords: conceptual modeling, visual programming, web interface, computational thinking, eLearning, eScience, STEM, high level languages.

1 Introduction

The differences between computing and computational thinking are significant and can be explained in a number of ways. According to Jeannette Wing [20], “computational thinking confronts the riddle of machine intelligence: What can humans do better than computers, and what can computers do better than humans? Most fundamentally, it addresses the question: What is computable? Today, we know only parts of the answers to such questions.” In particular, computational thinking is 1) conceptualizing, not programming, 2) a way that humans, not computers, think, 3) complements and combines mathematical and engineering thinking, and 4) it is ideas, not artifacts.”

In many countries, including USA, teachers, practitioners, developers and researchers are clamoring for new tools and systems to help realize these goals as part of emerging K-12 education standards such as NGSS [1] to better prepare students for new age of STEM education. Computing systems such as Alice, Scratch and Snap!, and educational systems such as Khan Academy (https://www.khanacademy.org/), ALEKS (https://www.aleks.com/) and so on have been developed in response to these pressing needs. Not only these systems and softwares are gaining popularity [5,18], they are also being accepted by formal academic institutions [17] as standards they are able to rely on. They,
However, are not above criticisms in the press (e.g., https://tinyurl.com/m8vko7v and https://tinyurl.com/kk4lrqm) and complaints (https://tinyurl.com/kzm5mny) about their delivery and conducts exists, as well as there exist significant concerns among the academicians about their effectiveness [19, 7].

NGSS calls for better integration and use of computing in STEM experiences, and in developing an effective computational thinking repertoire. Differently from computing, computational thinking demands the ability to string reasoning, analytics, knowledge, information, models, and hypotheses in a coherent manner, and often when one of the components is missing or inadequate, to develop one to solve a STEM problem at hand. If the missing component can or need to be programmed, it is prudent to expect a computational thinking approach to computing solutions as well to avoid any impedance mismatch across components within a larger system supporting NGSS or computational thinking. Approach to computational thinking for computing, however, is a lower level concept and appears to have a discipline specific bias as to how abstract should it be.

So, it is not a surprise that the computational physicists (e.g., [13]) insist on a more hands on computer programming experience than computational biologists (e.g., [6]) advocating visual or NLP based programming, or computational thinking. Fortunately, millions of K-12 students are now conversant in the graphical language Scratch [4] (developed at MIT). However, the language is incredibly clunky if one wants to go beyond what it was designed to do (make simple games), but a derivative language, Snap! [14] (developed at UC Berkeley), expands its horizons, making functions practical, enabling object oriented programming, and fixing other deficiencies of Scratch. Yet, many educators believe that it is a fantasy to expect the masses to transition from Scratch (or Snap!) to what we now think of as conventional programming languages such as C++, Python or Java. A more plausible future is for specialized languages to spring up that make use of the conventions of Scratch/Snap! but incorporate the knowledge and concepts of a field of interest to a target audience. It isn’t difficult to see a world where almost everyone speaks Scratch (like almost everyone knows algebra), and when we go into, for example, molecular biology, we naturally adopt Scratch/Genome, or if one wants to go into accounting, she adopts Scratch/Accounting. In other words, a discipline specific toolbox or plugin will tailor how the programming environment would interface with its users. Keeping these in mind, in this paper, we embark upon describing the contours of a customizable high level programming environment called Patch using a computing example. Patch is planned to be embedded within a fully autonomous tutoring and assessment system called MindReader [12] for online self-learning.

2 Patch Highlights using an Example

High school and freshman CS students learn several fundamental algorithms that they use as building blocks for more complex algorithms they will build in future by combining the basic ones in some fashion. These algorithms are language independent and can be understood at various levels of abstractions.
A bubble sorting algorithm, for example, arranges the values of objects in a collection (a list) in order of their significance – size of their values, shades of their color, or scale of the sounds. While the idea of sorting is simple, developing an algorithm or a computer program for them often is not. Yet, if given a few elements, students can usually rearrange them in the expected order by hand or visually. Decades of research helped develop numerous sorting algorithms that are vital to computing in general and all computer scientists are expected to master them. Figure 1 shows the pseudocode of the bubble sort algorithm and its actual C++ implementation. The way the algorithm works, i.e., its principal mechanism, is depicted visually in Figure 2 on a list of six values 29, -4, 2, 17, 45, and 9, as presented to the system, say, by a student.

![Algorithm 1: Bubble sort](image)

### Algorithm 1: Bubble sort

**Input:** A list of n values in random order

**Output:** Ascending order list

set sorted = false;

while not sorted do

set sorted = true;

foreach element i = 1, . . . , n - 1 do

if element i < element i + 1 then

swap elements i and i + 1;

set sorted = false;

end if

end foreach

end while

Fig. 1: Bubble sorting algorithm.

There are many online sorting algorithm visualization tools available including Toptal [2] and David Galles’ data structure visualization library [9]. These viz tools do a pretty good job at offering an intuitive understanding of the underlying working principles of the algorithms and even show the differences how they accomplish identical goals possibly with different performance overheads and under the conditions they do so. Nevertheless, these tools do not necessarily help the students develop the abstraction or the algorithm to perform the sorting. Needless to say that the distance from gleaning the sorting idea from these viz tools to developing the corresponding C++ or Python code is still significant, i.e., it is often difficult for a young student to translate a merge sort visualization to a executable C++ code that will be functionally equivalent.

### 2.1 Lowest Level of Abstraction

At the lowest level of abstraction, students write actual programs in MindReader [12] in languages such as C++, Python or even R. They have tool support, to find correct syntax, locating libraries, linking modular programs, compiling and
executing, testing and visualizing their results where appropriate, in similar ways the above viz tools allow. MindReader actually extends automated online tutoring and authentic assessment support to students differently and more powerful manner than the recently proposed Java tutor [15] that intelligently selects pre-defined templates to explain errors and how to correct them. However, at this abstraction level, students are actually required to develop codes directly in a language of choice.

2.2 Abstract Thinking

Instead, let us assume that we ask the student to write a C++ program to rearrange a list of values $a_i$ satisfying the condition that for all $i$, $1 \leq i \leq (n-1)$, $a_i \leq a_{i+1}$ (or, $a_i \geq a_{i+1}$) holds. We also offer her a graphical tool in which she is able to generate a set of $n$ values that she can also visualize in size proportional to the values. We can then contemplate three levels of abstractions - the lowest level is a C++ interface (as discussed above) in which she can directly enter a code fragment, execute it and see the results, and possibly use a visualization tool to see generated objects rearranged using her code.

The highest level of abstraction could be to let her manually and graphically rearrange her objects online. At this level, it is largely conceptual, and the actions will reflect how humans think. For example, she might choose to select the visually largest element at location $j$ to swap position with the one at the $n$th location, or simply bump all the elements up one location up to $j$ to make room at location $n$. If she continues to do so for the next largest, and then the next largest, a pattern will emerge. There are many ways this exercise can be modeled visually and some may be more sophisticated than the others. Regardless, it is possible to settle for one. For example, the visual implementation of the bubble sort algorithms introduced in Figure 1 is shown in Figure 2.

In this example, a student used iconized predefined operations on a grid to explain her procedure (numbered 1 through 8) to the computer to help the computer write the code for her in a language of choice. Icons in this figure represent conceptual actions, and further explained using forms, and they are used in a hierarchy – one to the right on the grid expands the icon immediately on
the left, giving it a tree structure. Objects can be selected, highlighted, dragged
and dropped to indicate actions. The diagram shows the module begins at step
1, which consists of the remaining steps to the right. She then created a random
list of six animated values. In step 2, she selected all the elements to signify
that the operations to the right of step 2 apply to all the elements starting
with the first one in step 3. She selected the neighbor element in the list, and
then compared in step 5, swapped them in step 6 the first being larger than its
neighbor. The process can be animated with the push of a button at any point,
and be visualized. She noticed that at the end of the animation, the largest
element was placed in the sixth position. She then repeated the process in step
2 deciding its not sorted. When she noticed at some point, the list was sorted,
she added the stop icon at step 8, completing the algorithm. As we will see in
section 4 that the level of details needed to implement the algorithm in Figure
2(a) is different, and the description in Figure 3 is more closer to the C++
implementation in Figure 1(b), while the Figure 2 is the most abstract.

2.3 Conceptual Thinking
To bridge these two levels of abstractions, we envision an intermediate or mid-
dle layer, which is still conceptual but uses constructs closer to programming
languages such as assignments, loops and decision, and is language and syntax
agnostic. Closest analogy to this abstraction layer is a computational biology
querying system called VisFlow [16]. In VisFlow, users express their compu-
tational procedures visually using high-level conceptual icons, and connecting
them in meaningful ways. While the objectives and functionalities of VisFlow
are significantly different from the Patch interface we are proposing, the look
and feel, and the approach to expressing algorithms are similar.

3 Programming in Patch
The design of Patch is based on the principle of simplicity, clarity and naturalness
of computational thinking toward programming with support for modularity and
extensibility. These choices roughly means simpler scope rules, structured con-
cept generation and incremental programming, and automatic concept matching
for module identification between modules. At the level of language, it also means
Patch supports both name independence (similar to C++, Prolog), and position
independence (similar to SQL) so that the order or the naming of parameters
of a module will have almost no relevance to its appropriateness when called
from or used in another module. All a user will have to know what a module
conceptually requires as inputs to generate a desired output. In this sense, Patch
is more declarative than procedural in design but procedural in its execution.

3.1 Basic Constructs
Patch supports an array of simple data types, programming construct, input-
output functions and user defined operations keeping simplicity in mind.
**Data Types and Variables** Patch supports basic data types integer (e.g., 14, 2, 657), real (e.g., 2.09, 77.21), Boolean (TRUE and FALSE) and string (e.g., "language", "Programming in Patch"), and native complex types lists (e.g., [20, 9, 34], [2.3, 50.77], ["Moscow", "Java"]), sets ({87.2, 2.87}, {"Patch", "Java", "C"}) and tuples (e.g., ⟨2, "Main Road", "New York", 10026⟩) of basic types. Except tuple types, all complex types are homogeneous (of the same type). Complex types, however, can have elements of complex types giving endless possibilities, e.g., lists of lists or tuples. Naming a value of basic or complex type is simple – variable names are case insensitive. Members or elements in a ordered complex type can be accessed using its position in the order using referencing operation [i]. This rule applies to both lists and tuples. Given,

\[
x=["Moscow", "Java", "Pea"]; \
y=(2, "Main Road", "New York", 10026),
\]

Patch returns "Pea" and 10026 respectively for x[3] and y[4], and yes, the base of all collection types start at 1. While individuals in lists and set types are not named (being of the same type), members of a tuple type are always named (being potentially of different types and thus conceptually different).

As we just mentioned, they can be accessed or referred by their position, they can also be accessed or referred to by their name. So, both y[4] and y.zip refer to the member 10026, if indeed it is named zip. Needless to mention that membership in a set type data can only be tested, and not accessed. Set operations allowed are membership (∈), difference (\), union (∪), intersection (∩) and cartesian product (×) with their usual mathematical meanings. In the same vein, Patch supports arithmetic operations plus (+), minus (−), multiplication (∗), division (/), and exponent (^), and comparators less than (<), greater than (>), equal to (=), less than or equal to (≤), greater than or equal to (≥), and logical connectives AND (∧), OR (∨), and NOT (¬). These operations and comparators are defined over compatible types. Both real and integers are compatible, strings are compatible with strings, and Booleans are compatible only with Booleans.

Finally, since Patch is a visual language, a corresponding keyboard syntax for all the above are not truly necessary as we will see.

**Algorithmic Constructs** In Patch, algorithms are considered logical operations constructed in steps, and by reusing previously defined computationally compatible modules or units. Patch supports five basic operations – assignment or transformation, branching, and repetition, plus input output functions such as reading and displaying. Technically, these steps are modeled as embedded trees – steps may include other steps. For example, a repeat operation may include two operations, an assignment and another repetition whereon the second repetition contains two more operations and so on. These constructs are described next.

*Assignments and Transformations* In Patch, computing an expression as a value is treated differently than assigning a value to a variable. Thus assigning constant 17, or a variable y which as a value 17, to a variable x is considered assignment.
Computing $16+1$ or $y-1$ and assigning the value to $x$ is considered a computation or transformation because transformations mutate objects, assignments do not. This distinction plays a role in understanding code segments in MindReader tutoring and assessment system of which Patch is a component. In any case, every assignment or transformation is assigned to a variable of appropriate type over which the operations are valid. Thus, $2+3.57$ producing $5.57$ or $45+3$ producing $48$ assigns $5$ to integer $x$ stores $5$ and $48.0$ into real $x$, respectively.

Branching

Branching controls flow of execution which is usually downwards, and is based on some form of Boolean decision. There are three forms of branching – by-pass, either-or, and labeled.

In a by-pass branching (traditional if-then), a Boolean decision is made if a set of steps should be followed or by-passed to the steps following them.

In contrast to a by-pass branching, exactly one set of steps is executed based on a positive or negative decision (traditional if-then-else).

But in a labeled branching, a set of steps is executed from a list of labeled steps (C++ switch). But unlike languages such as C++ or Java, in Patch, these labels can be any constant matching a variable, i.e., need not be an ordinal.

The branching decisions are based on appropriate Boolean comparisons over Boolean expressions that follow traditional logical expression constructs.

Repetitions or Loops

There are three basic types of repetitions in Patch – counter or automatic, conditional and sentinel repetition.

A counter loop or automatic loop (traditional for) executes its body a fixed number of times, and has a counter or index variable. The counter variable can be assigned a start and an end value before the loop starts, and can be used in side the loop body but cannot be assigned a new value inside the loop body.

A conditional loop (traditional while) continues to execute so long a Boolean condition, called the loop condition, remains TRUE. The loop condition is usually reversed inside the loop body.

Sentinel loops continue until an element in a complex data collection, a sentinel or a marker, is found. The collection is usually a list or a set, and its members are used in the loop body one by one until the marker is reached.

In all these loops, as well as in branching, Patch exists the body of steps it is executing if forced using a EXIT command. On exit, control returns to the step immediately after the branch or loop operation and help stop processing under exceptional circumstances.

Modules

Modules or units have different meanings in various languages and have rich history [10][3]. We break from traditional approach to modules and procedures to allow flexibility and support intuitive programming and adopt the position that the order of the parameters or their naming should not matter.
We use a schema mapping function to establish correspondence instead and allow a call only when a one-to-one mapping between formal parameters can be established. This way, we support both name independence supported in most programming languages and position independence of SQL. This choice though introduces slight uncertainty in module selection, but enhances conceptual understanding. This also means, the traditional function overloading may not work well in Patch. However, in a first computational thinking course/class, we do not see much value in supporting such complex concepts to start with.

3.2 Reading and Visualization

Patch supports basic input output operations just as any other language. But additionally to basic display of outputs, Patch also supports powerful animation and live color visualization in ways similar to Protovis [11] with the objective of supporting comprehension with execution displays similar to Toptal and Galles.

4 Patch Programming Interface

The Patch programming interface shown in Figure 3 is embedded within the MindReader [12] online tutorial and assessment system. Broadly, the interface has three panels, one icon bank to the left (blue ellipse) and one command and folder bank at the top (cyan box). The left panel (red box) is a canvas where users express their computational solutions, and right panel (green box) serves as the display and input console while the bottom panel (blue box) is reserved for diagnostics and system messages. Users draw a tree involving the icons in the icon bank, the root of which is the module icon. The solid arrows represent downward flow of control or sequence of actions, and the dashed arrows represent membership of actions in another action or step. For example, the dashed arrow from the top module icon means the remaining subtree is contained within the module, and the same way the subtree rooted in 7 is contained within node 4. The Patch tree shown in Figure 3 is the bubble sort algorithm as designed by a user and Patch is showing the state after step 3.

Technically, every Patch program is a set of named modules each with a (possibly empty) set of input data objects, and a set of data objects it returns when called. Any subset of the data objects in its I/O sets can be from/to the console, a stored repository or another module that uses it. Naturally, I/O from/to console and stored repository takes place inside the module. A module is a pair \( \langle D, S \rangle \) where \( D \) is a set of data objects it uses, and \( S \) is a sequence of logical steps it follows to manipulate the data objects. The four basic, and three complex data types introduced in section 3.1 that Patch supports can be used to declare the I/O data objects. Users are able to execute the tree on the canvas at any point and see the results on the green panel. The interface is interactive and actually shows what could be expected on the green panel as the drawing continues. The qualitative details of each node is entered using appropriate forms that can be opened with right clicks of the mouse. The drawing rules ensure all graphs are trees and each node has no more than one solid or one dashed arrow.
5 Related Research

The emergence of Scratch and Snap!, and online tools such as Khan Academy or PythonTutor ([http://pythontutor.com/cpp.html#mode=edit](http://pythontutor.com/cpp.html#mode=edit)) are relatively new. The focus of each of these systems are different and thus often are hard to compare on identical grounds. But we believe in functionality, Patch shares its execution style with PythonTutor, and it is conceptually similar to Scratch and Snap! But Patch combines the power of MindReader’s tutoring and assessment that no other system currently has. We believe the ability to build programs visually, incrementally and interactively with active tutoring support is a powerful combination. Research is, however, outstanding on the effectiveness of the approach adopted in Patch and MindReader in real life classrooms.

6 Summary

Patch is an experimental prototype with limited but emerging set of functionalities and support for multiple abstraction layers for diverse student groups and multiple STEM disciplines. Patch supports exploration and self-learning naturally. It is currently being implemented and tested to validate the hypothesis that all three abstraction layers can be seamlessly supported in a single platform and augmented with an active tutoring and assessment component. Initial experimental results are promising though more research is needed. Patch guarantees generating functionally equivalent code segments at all three of these layers if the solutions specified are indeed identical. Generated and written codes can be compared at the click of a button. Currently, a set of basic programming features for entry level programming course is being identified and abstracted, and methods are being devised to make the system incremental and hierarchical so that complex features can be designed and added using the basic features. A community based approach to incremental design is being contemplated.
References

1. Next generation science standard. [http://www.nextgenscience.org/](http://www.nextgenscience.org/) Accessed on April 19, 2017.
2. Toptal. [https://www.toptal.com/developers/sorting-algorithms](https://www.toptal.com/developers/sorting-algorithms) Accessed on April 19, 2017.
3. S. Abreu and V. Nogueira. Towards structured contexts and modules. In *ICLP*, Seattle, WA, USA, August 17-20, pages 436–438, 2006.
4. L. A. Calao, J. Moreno-León, H. E. Correa, and G. Robles. Developing mathematical thinking with scratch - an experiment with 6th grade students. In *EC-TEL*, Toledo, Spain, pages 17–27, 2015.
5. C. Chang, Y. Lin, and C. Chang. Using visual programming language for remedial instruction: Comparison of alice and scratch. In *ICWL*, Kenting, Taiwan, October 6-9, pages 224–233, 2013.
6. J. Elhai. Humans, computers, and the route to biological insights: Regaining our capacity for surprise. *Journal of Computational Biology*, 18(7):867–878, 2011.
7. W. Farag. Comparing achievement of intended learning outcomes in online programming classes with blended offerings. In *ACM SIGITE*, Calgary, AB, Canada, October 11 – 13, pages 25–30, 2012.
8. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN*, Montreal, Canada, June 17-19, 1998, pages 236–248.
9. D. Galles. Data structure visualization library. [https://www.cs.usfca.edu/galles/visualization/Algorithms.html](https://www.cs.usfca.edu/galles/visualization/Algorithms.html) Accessed on April 19, 2017.
10. J. Gutknecht. Variations on the role of module interfaces. *Structured Programming*, 10(1):40–46, 1989.
11. J. Heer and M. Bostock. Declarative language design for interactive visualization. *IEEE Trans. Vis. Comput. Graph.*, 16(6):1149–1156, 2010.
12. H. M. Jamil. Automated personalized assessment of computational thinking mooc assignments. In *IEEE ICALT*, Timisoara, Romania, July 3-7, 2017. To appear.
13. R. H. Landau, G. Mulder, R. Holmes, S. Borinskaya, N. Kang, and C. Bordeianu. INSTANCES: incorporating computational scientific thinking advances into education and science courses. *Concurrency and Computation: Practice and Experience*, 26(13):2316–2328, 2014.
14. H. Leleux, T. Holcombe, A. Strong, G. Yera, C. Smith, and T. Gonzales. Work in progress: Programming is a snap! increasing knowledge and interest in computer science. In *ACM SIGITE*, Chicago, September 30 - October 3, page 101, 2015.
15. V. J. Martin, T. Pereira, S. Sridharan, and C. R. Rivero. Automated personalized feedback in introductory java programming moocs. In *IEEE ICDE*, California, USA, April 19-22, pages 1259–1270, 2017.
16. X. Mou, H. M. Jamil, and R. Rinker. Visual orchestration and autonomous execution of distributed and heterogeneous computational biology pipelines. In *IEEE BIBM Shenzhen*, China, December 15-18, pages 752–757, 2016.
17. A. A. Reddy and M. Harper. Aleks-based placement at the university of illinois. In *Knowledge Spaces, Applications in Education*, pages 51–68. 2013.
18. J. Sáez-López, M. Román-González, and E. Vázquez-Cano. Visual programming languages integrated across the curriculum in elementary school: A two year case study using “scratch” in five schools. *Computers & Education*, 97:129–141, 2016.
19. I. Simonova. Multiple-choice testing: Knowledge, or random choice? In *IEEE EDUCON*, Istanbul, Turkey, April 3-5, pages 819–823, 2014.
20. J. M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.