COMS 3998: Project Proposal

Veronica Woldehanna | vtw2108 | Fall 2018

SAGE Gameful Direct Instruction: Parson's puzzles

1. Abstract

This paper explains the goals for development on the Gameful Direct Instruction epic for the SAGE (Social Addictive Gameful Engineering) project. It aims to improve parson's coaching capability and to give feedback that better reinforces learning.

2. Introduction

SAGE [1] extends scratch, a drag and drop based programming language that allows users to develop their own games, animations and puzzles, and adds the ability of instructors to author puzzles and students to gamefully solve them as they learn computational concepts.

Parson's programming puzzles [2] in particular is a type of game that allow students to reorder blocks of code to get the desired solution. Parson's puzzle was created to avoid the problem of having syntactical issues block a student from learning concepts, to make rote learning of the syntax fun, to constrain the logic of a potential solution so students are guided to the solution and are kept from being side tracked, and to make it engaging for students to learn.

Parson's puzzles in SAGE [3] allows the instructor to author the puzzle, incorporates "Self-Explanations" that help reinforce learning, animates sprites to indicate success or failure, has a scoring system that takes in to account not just the final solution but how students get there and will allow the teacher to differentiate between student performances and assess mastery of a concept.

However, when Dale Parson first wrote the paper on parson's programming puzzles, he notes that the majority of students wanted a better feedback system. They wanted specific explanations discussing their errors. He also notes that this is very difficult to do because that would mean preparing text comments for potential error (or class of errors) and map them to each possible incorrect arrangement. This is assuming we have understood the student's confusion, which in itself is difficult to do especially when you have only a misplaced block of code to go off of. This difficultly still persists.

Instead, this paper specifically aims to help students understand where they went wrong as opposed to trying to identify it for ourselves and only then giving them our hints.

3. Proposal

3.1 Visual of code execution

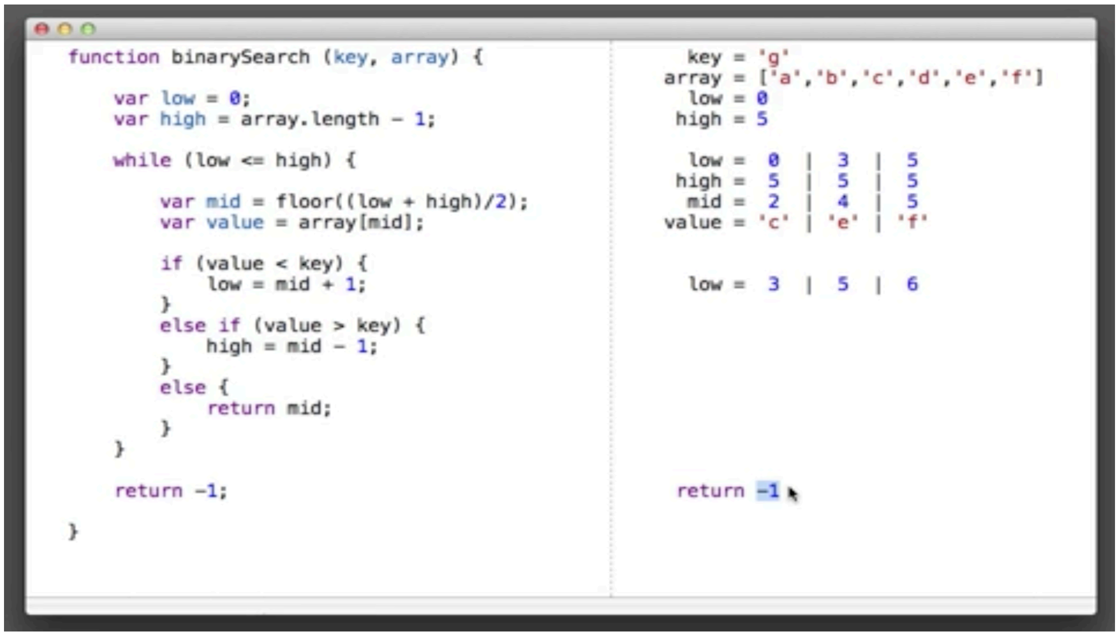
In most university courses instructors usually use visual aids to explain a machine's inner working during the execution of a piece of code. Similar drawings are also used when students talk amongst each other or when talking to teaching assistants.

In 2012, Bret Victor, in an article he wrote called "Learnable Programming" [4], explains that a coding environment should be able to let the learner easily understand what the syntax means, understand what happens and when, what state the program is currently in, get to a solution by correcting mistakes and working towards it, as well as get comfortable enough with a piece of code so a learner can abstract it.

The visual aids mentioned above are very powerful because they manage to let the learner understand the code execution and what state it is in, modify the code to get to the goal state using logical reasoning, and trust the code enough that learners can comfortably abstract it and move on to greater problems. Using visualizations, students can understand concepts better because they are learning them in context. So, it helps them conceptualize computer science concepts that foster computational thinking like recursion, iteration, and abstraction

There are currently live programming environments that change the output instantaneously as you modify your code. However, this isn't very helpful and as Brett voices in the article, you can't just give students cooking ingredients expecting them to understand how the ingredients were combined. A better way to show them would be step by step.

Applying this to programming, we could show them how variables change as each line of code is written. Furthermore, we could also show them the history of the variables along with their current values to make it even better.



```
function binarySearch (key, array) {

    var low = 0;
    var high = array.length - 1;

    while (low <= high) {

        var mid = floor((low + high)/2);
        var value = array[mid];

        if (value < key) {
            low = mid + 1;
        }
        else if (value > key) {
            high = mid - 1;
        }
        else {
            return mid;
        }
    }

    return -1;

}
```

```
  key = 'g'
array = ['a','b','c','d','e','f']
  low = 0
 high = 5

  low =  0  |  3  |  5
 high =  5  |  5  |  5
  mid =  2  |  4  |  5
value = 'c' | 'e' | 'f'


  low =  3  |  5  |  6









 return -1
```

Figure 1: Bret's example of an optimal visualization of code execution

This specific approach was studied [5] on novice programmers and the study yielded very promising results. It was found to help students form proper mental models and more clearly discuss a program's behavior.

Applying this to Parson's puzzles, a similar system could be designed that can act as a feedback mechanism such that as every block is positioned in the script pane, the variables update. Currently, should the students place the wrong block next, they are just told they made a wrong move and a point is deducted from their score but they aren't told why their move was wrong. If they don't understand what went wrong and what the problem is, how can we expect them to come up with a solution. If Parson's supports visualization, however, by looking at the progression of states, they can see that the code is heading in an undesired direction. They'll understand what their code is currently doing and have a better idea of what their next step will be because they now know what the problem they have to overcome is.

As students are first starting to program, it can be kind of hard to isolate the variables they think they have to keep track of and the ones they don't have to care about for the moment as they visualize the execution of their code. However, with the visualization laid out for them, they'll easily see what their code affects and what it doesn't, they'll start to recognize patterns and eventually once they feel comfortable they'll start experimenting on different approaches they can take. This will also let them focus on higher level concepts instead of focusing on the details. They could even compare different approaches and have an idea as to which is better.

Since we are further breaking the problem down into mini-problems they run into and solve, it can be Gameful and engaging. Furthermore, we can add animations to the code visualization to keep them entertained and make it fun.

3.2 Feedback for errors

Visualization of code works given that the computer can execute it, meaning that students' solutions can't have syntax problems. A solution to get rid of this limiting factor can be to map compiler errors (or groups of them) into a naturalistic and conversational report of what the problem is. This report can be voiced by a sprite so that it can be interactive.

For example, if a student drops a block that makes use of a certain variable "x" before inserting the block that declares "x", the student can immediately hear from a sprite "Uh-oh, looks like you never told me what 'x' was." This will show the student that the steps he has to take are very logical and not just there as rules.

3.3 Hover for meaning/syntax

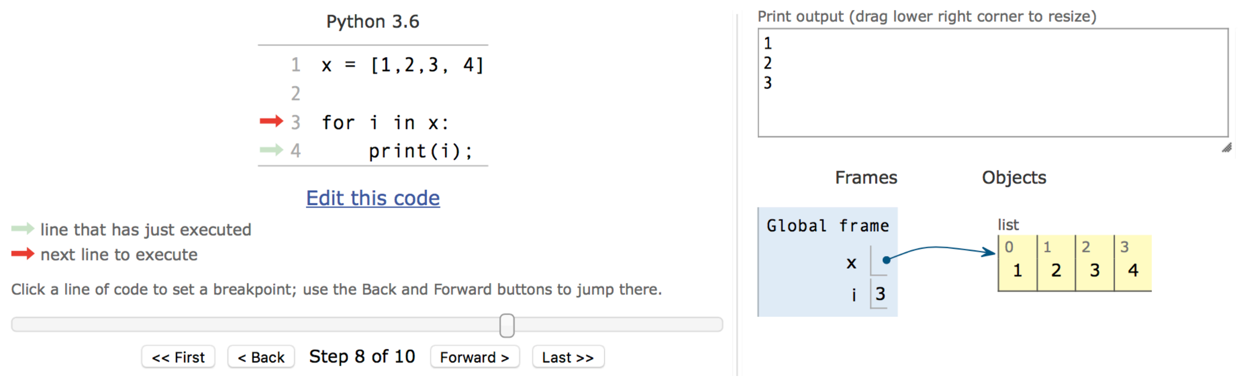
Syntax can be a huge road block to students and even though Parson's somewhat avoid syntax errors by giving them partly constructed code, it can be frustrating for students to see a block of code they don't really understand. If we allow instructors to enter quick definitions for a new construct they want to introduce students to (they could do this incrementally), then should students find a block confusing and they're not really sure what it does, they can potentially hover over it (it being the entire block, a phrase, or a word), to find out what it means.

This will allow instructors to introduce them to new syntax incrementally alongside the concepts they're learning while also avoiding syntax errors and student frustration.

4. Related works

4.1 Python Tutor

Python tutor is an online tool that visualizes code.



```
x = [1,2,3, 4]

for i in x:
    print(i);
```

line that has just executed
next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

Print output (drag lower right corner to resize)

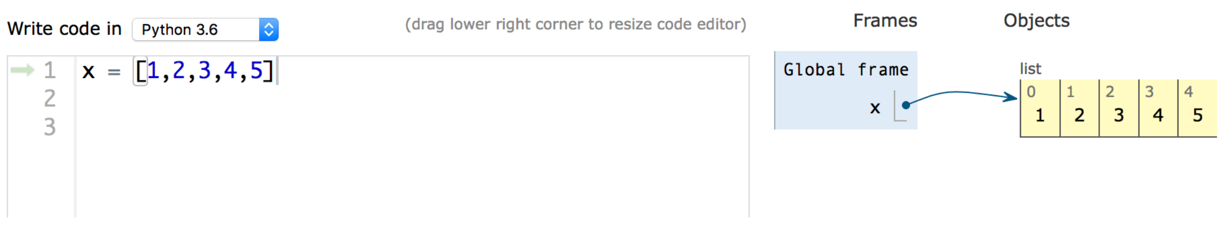
```
1
2
3
```

Figure 2: Python Tutor

Python Tutor will list the variables on the right changing their values accordingly as each line of code executes. This however could be improved to show instant change as each line of code is written.

4.2 Python Live Programming

Python live programming is better in that it shows instant change as code is written.



```
x = [1,2,3,4,5]
```

Figure 3: Python Live programming

This, however, could be made even better if we saw not just the updated values of variables but their entire updated history as each line is written.

4.3 OmniCode

OmniCode, partly inspired by Bret Victor, doesn't just give you updated variables as each line of code is written, it gives you an updated history of the variable. This has proven to be successful in field study experiments mentioned earlier [5].

5 Future Work

The hinting system could look at patterns in the visualization of the code and try to find specifically wrong ones in the context of the problem, and offer it to students as a hint in case they might've missed it.

6 References

Bender, J. (2015). Developing a collaborative Game-Based Learning System to infuse computational thinking within Grade 6-8 Curricula.

Parsons, D. and Haden, P. (2006). Parson's Programming Puzzles: A fun and effective learning tool for first programming courses.

Mohan. (2017). Gameful Direct Instruction (Parson's Puzzle)

Victor, B. (2012). Learnable programming: Designing a programming system for understanding programs.

Kang, H. and Guo, P. (2017). OmniCode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations

Comment: Modifications could be made if some suggested functionalities are already available, either suggesting new ideas or improving existing functionality. I will also be looking in to "sub-goal labels"