Snapdown: A Text-Based Snapshot Diagram Language for Programming Education

Daniel Whatley
MIT CSAIL
Cambridge, MA
dwhatley@alum.mit.edu

Max Goldman
MIT CSAIL
Cambridge, MA
maxg@mit.edu

Robert C. Miller
MIT CSAIL
Cambridge, MA
rcm@mit.edu

***Abstract*—*Snapshot diagrams*, which visualize in-memory program state, are frequently used in programming education to demonstrate new concepts and help students develop a better understanding of program functionality. In this paper we introduce *Snapdown*, a textual language for drawing snapshot diagrams, designed for use by both students and instructors of programming courses. Snapdown is designed with an emphasis on learnability and simplicity: both to be picked up by students in a classroom setting in a matter of minutes, and to enable creation and maintenance of diagrams in instructional content with minimal overhead. We introduce several use cases of Snapdown and describe the design and features of its textual language. We also describe a deployment of Snapdown during two semesters of emergency remote teaching in 6.031 Software Construction at MIT, a software engineering course intended for sophomore- and junior-level undergraduate students.**

***Index Terms*—snapshot diagrams, textual language, programming education**

I. Introduction

In many programming courses, students complete exercises that involve writing or analyzing a program. Some exercises ask students to draw diagrams to help with mental visualization. Such diagrams, commonly termed *memory diagrams* [6, 21, 12], typically show the state of a program and its variables. These diagrams are commonly used as a teaching aid: instructors often include them in their lecture notes, digital textbooks, online exercises, and other course content. These diagrams are designed to be simple and flexible, only showing particular aspects of the program state that are relevant to the discussion at hand. We refer to these diagrams as *snapshot diagrams*, as they represent transient snapshots of memory, and depict both objects on the heap and frames on the call stack. Figure 1 is an example of a snapshot diagram presented early in 6.031 Software Construction at MIT, a software engineering course taught by the authors.

Generating many snapshot diagrams for online course content is not an easy task. Instructors must create, update, and maintain snapshot diagrams as course content changes and as students need more examples. Diagrams can be drawn by hand, constructed with an image editor like Graphviz [10] or OmniGraffle [24], or with a low-level diagramming language like DOT [8]. However, these methods are not designed to

```
cities -> (
  List<String>
  0 -> (String "Boston"),
  1 -> (String "Bogotá"),
  2 -> (String "Barcelona")
)
```

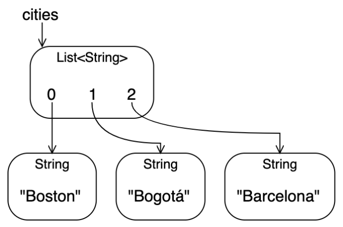


Fig. 1: A snapshot diagram that shows a list of three String objects. Left: Snapdown syntax, right: corresponding diagram.

support continuous changes in diagram content and structure. In addition, some of these methods have steep learning curves, and are not feasible for students to use in completing coursework. Programming courses typically contain tens or hundreds of code snippets, examples, and exercises, and several challenges would immediately surface if instructors and students were to use these methods to draw diagrams illustrating them.

In this paper, we address these challenges with *Snapdown*, a text-based language for drawing snapshot diagrams, intended for use by both instructors and students. Inspired by Markdown [14], Snapdown aims to represent graphical objects using text syntax that resembles the diagram as closely as possible. Snapdown also takes advantage of the fact that a primary purpose of these diagrams is to show associations between names and values: the same names that appear as labels in the diagram can be used to refer to objects in the text source, making many diagrams more straightforward to draw. Finally, Snapdown gives users the ability to draw *multi-step diagrams* that show the evolution of the state of a program. Figure 1 shows Snapdown syntax on the left for the corresponding diagram on the right, and Figure 2 contains an example four-step diagram drawn with Snapdown.

We evaluated the learnability of Snapdown by asking students and instructors of 6.031 to draw diagrams with it. Snapdown proved to be an advantage during the COVID-19 pandemic: students could complete diagram exercises online using Snapdown without a drawing tablet or other means. In addition, 6.031 incorporates pair programming exercises into its class sessions. Using Snapdown, students collaboratively edited diagrams in their web browsers. Overall, approximately 500 students used Snapdown for 10-15 exercises across two semesters, Fall 2020 and Spring 2021, of emergency remote

978-1-6654-4592-4/21/$31.00 ©2021 IEEE

```
t -> (String "a")
tarr ->
  (List<String> 0 -> t)
---
s -> t
arr -> tarr
---
#r -> (String "ac")
s -x> t
s -> #r
---
tarr add 1 -> (String "d")
```

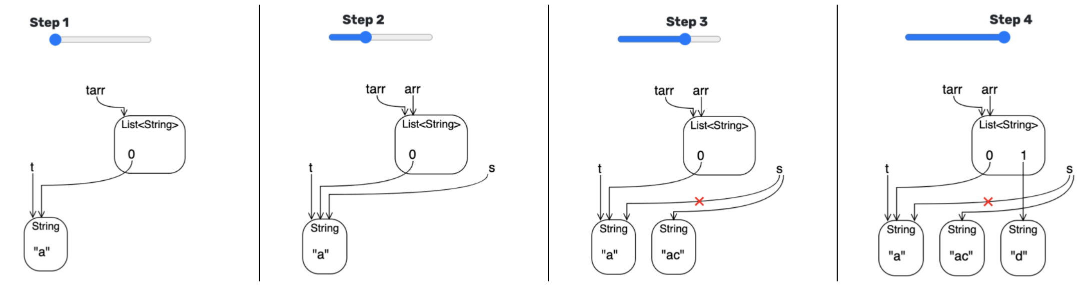


Fig. 2: Multi-step diagram demonstrating reassignment and mutation of variables. Example Java code follows. Step 1: `String t = "a"; List<String> tarr = List.of(t);` Step 2: `String s = t; List<String> arr = tarr;` Step 3: `s = "ac";` Step 4: `tarr.add("d");`

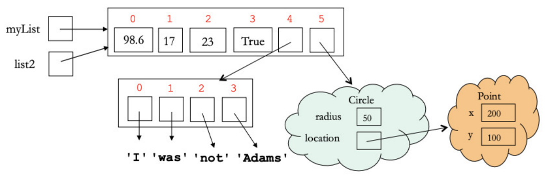


Fig. 3: Example figure from Wellesley CS 111's lecture notes on lists and memory diagrams (reproduced with permission).

```
myList -> [
  98.6, 17, 23,
  `True`, #subList, #circle]
list2 -> myList
#subList -> [
  "I", "was", "not", "Adams"]
#circle -> (
  Circle
  radius -> 50
  location -> (Point
    x -> 200
    y -> 100)
)
```

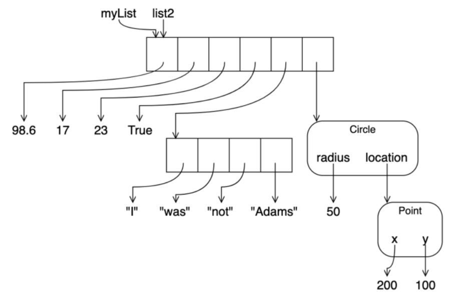


Fig. 4: Snapdown representation of diagram from Fig. 3

teaching in 6.031. Instructors of 6.031 created approximately 20 new diagrams using Snapdown for course material.

We also evaluated the generalizability of Snapdown. Using Snapdown, we recreated over 100 diagrams from 11 programming courses at a variety of institutions. For example, Figure 3 shows a snapshot diagram presented in the lecture notes of CS 111 at Wellesley College [25], and Figure 4 shows the equivalent representation in Snapdown.

This paper makes the following contributions:

- *Snapdown*, a textual language for drawing snapshot diagrams, designed to be learnable by students in a matter of minutes, with a syntax that resembles the shape of diagram elements, and that uses the same syntax for drawing names in the diagram and binding names in text.
- An implementation of Snapdown that generates SVG diagrams with an intermediate representation in JSON, which facilitates embedding Snapdown in web applications and collaborative drawing of diagrams.
- A syntax and mechanism for rendering *multi-step diagrams* that is easy to write and maintain and preserves visual consistency between steps.
- An analysis of how students and instructors in a large programming course used Snapdown over the course of two semesters.

In the next section we review related work in diagram layout engines, and diagrams and visualizations used in programming education. We then introduce Snapdown syntax and the situations in which it can be used. We detail Snapdown's layout routine, its approach to both single- and multi-step diagrams, and evaluate its learnability and effectiveness. We conclude with limitations of our approach and future work.

II. Related Work

A. Diagramming languages

Several diagramming languages are well-known and commonly used to draw diagrams for a variety of purposes, including ones related to programming education. Systems such as Penrose [18], DOT [8], Mermaid [15], and PlantUML [19] are examples. Of these, Mermaid and PlantUML both share Snapdown's approach of providing an accessible language. These systems focus heavily on drawing UML diagrams. Object diagrams in UML are similar to snapshot diagrams, but are much more focused on showing single snapshots of class and object hierarchy.

Overall, these systems solve different problems and focus on different use cases. For example, UML object diagrams are not designed to show how variables and values change over time. UML object diagrams also do not contain *compound nodes*, while most snapshot diagrams do. (In Figure 2, the `List<String>` objects are all examples of compound nodes: they have internal fields such as `0` and `1` which point to other nodes, such as the `String` objects.) Drawing a visually appealing snapshot diagram in a system designed for UML diagrams would require careful manual positioning or substantial syntax additions. This added work presents a complexity barrier which can distract the user from their primary goal of drawing a diagram.

B. Drawing diagrams in classroom settings

Research has shown that drawing diagrams is beneficial yet difficult for students in programming courses. Sajaniemi et al. [21] performed studies with students in an introductory programming course, collected many student-drawn diagrams for different types of exercises, and analyzed them to check for common mistakes and misconceptions. Based on the diagrams, a small number of key concepts, such as objects and constructors, caused most confusion. A study by Holliday and Luginbuhl [12] involved giving two different versions of a quiz to students, one with multiple-choice questions, and another asking students to draw snapshot diagrams. The quiz results led to the conclusion that drawing a correct diagram is more difficult than correctly answering the multiple-choice questions, and that a student's ability to draw a snapshot diagram is positively correlated with an assessment of their understanding.

Researchers have also performed experiments to show the effectiveness of visualizations in different settings. A study by Hendrix *et al.* [11] divided participants into two groups, one presented only with a question about a code snippet, and the second presented with the same question and code snippet but also an accompanying diagram. The group presented with the diagram performed significantly better on the question than the group without. A study by Baltes and Diehl [1] concluded that diagrams can be a helpful way to enhance out-of-date company documentation.

C. Generating diagrams from code

The problem of visualizing executable code with diagrams has been explored extensively. Python Tutor [20] is a widely-used system that generates snapshot diagrams, with both stack and heap, from executable code in several programming languages. GitUML [9] can generate UML diagrams from GitHub repositories containing object-oriented executable code. Systems built into certain IDEs such as the UML Generator for IntelliJ IDEA [13] and ObjectAid for Eclipse [16] can generate UML diagrams from code in existing projects. A paper by Dalton and Kreahling [4] took a step toward automatically generating snapshot diagrams from executable code in an instructional setting, with a focus on generating incorrect diagrams for teaching purposes. Incorrect diagrams were generated through a custom language specified by the authors that resembles executable code.

Compared to these systems, Snapdown has different requirements and makes different tradeoffs. For example, diagrams drawn directly from executable code can become very large and contain parts of the heap or stack that are beyond the scope of course material. We believe our approach better serves students and instructors who only wish to visualize parts of the heap or stack relevant to their specific needs. In addition, in scenarios where a user has not yet written working code but wishes to draw a snapshot diagram, they can do so without needing completed code that compiles and runs.

D. Generating diagrams with code

Constrain [2] is an example of a system designed to draw arbitrary figures including snapshot diagrams. Constrain is used in classes such as CS 2112 at Cornell to demonstrate evolving states of programs. To draw simple snapshot diagrams in Constrain, at a level of complexity similar to that of Figure 1, a user would need to learn approximately 10-15 Constrain library functions. A description for such a diagram in Constrain consists of approximately 20 lines of JavaScript—each line draws a single diagram element using multiple function calls. More involved diagrams require many more lines of code, and require the user to learn and use more library functions. Such code, while possible for instructors to write and use to draw a large range of diagrams, would be infeasible for students to write for their coursework. Diagramcodes [5] and Structurizr [22] are two other examples of systems that draw diagrams from a representation written in code, and each have a similar complexity overhead to that of Constrain. In comparison with these systems, Snapdown's language is much smaller and constrained and has much less flexibility. Snapdown does not require students to learn a new API or a new programming language, which greatly reduces the slope of its learning curve.

III. Language Design

Snapdown's language supports the following features. The examples presented in this section were chosen with the Java programming language in mind, but Snapdown is a language-independent drawing tool.

A. Primitives and variables

Figure 5 shows examples of Snapdown syntax for primitive data types and variables, as well as the corresponding diagram. Arrows between variable names and their values are represented with a textual arrow, `->`.

Variable names can be a single alphanumeric word, similar to those in most programming languages. For the purpose of snapshot diagrams, however, we may want to include the types of variables as well. Placing backticks around literals, such as `` `int i` ``, allows for any string literal to be interpreted as a variable name or a value.

Note how all arrows point downward, and how variables are introduced left-to-right as they are written in Snapdown. This design was based on an evaluation of diagrams drawn on paper and collected in the course of 6.031, and drawn by course staff during an initial exploratory study. Students and staff both had a strong preference to draw arrows top-to-bottom and introduce variables left-to-right.

```
i -> 12345
‘int i‘ -> 12345
str -> "I am a string!"
str2 -> "I am a
multiline
string!"
```

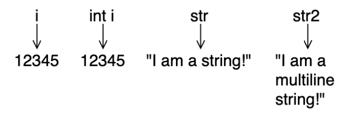


Fig. 5: Primitives and variables

B. Objects, fields, and arrays

Figure 6 shows an example of objects, fields, and arrays, and the corresponding Snapdown syntax. We use parentheses around class names and fields to resemble the round shape of an object in the diagram. The `MyFloat` object contains only its inherent value of `5.0`, and the array contains two

```
f -> (MyFloat 5.0)
arr -> [
  (String), (String)]
lst -> (
  ArrayList
  0 -> 1000
  1 -> 2000
  2 -> 3000
  length -> 3
)
```

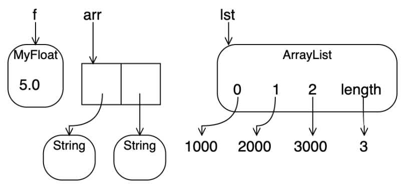


Fig. 6: Objects, fields, and arrays

```
a -x> 1
a -> 2
b -> a
```

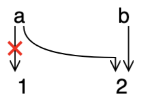


Fig. 7: Reassignments

```
coursesSet -> (Set<Course>
  _ -> cs1 _ -> cs2)
cs1 -> (Course
  `...` semester#1 -> #sem)
cs2 -> (Course
  `...` semester#2 -> #sem)
#sem -> (
  Semester
  season ->
    ((String "Fall"))
  year -> 2020
)
semester#2 -x> (Semester
  `...Spring 2020...`)
```

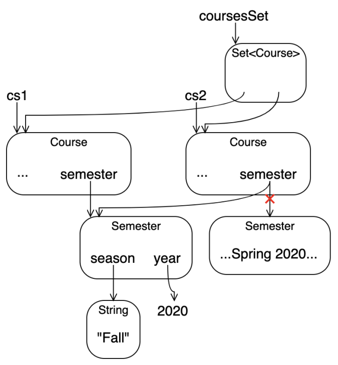


Fig. 8: Name-binding and resolution

```
x -> (MyClass)

topOfStack() {
  foo -> 5
  this -> x
}
lowerInStack() {}
main() {}
```

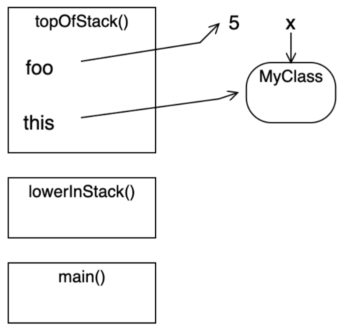


Fig. 9: Stack frames

String objects. The third object goes into more detail on the inner structure of the ArrayList. It has four fields, 0, 1, 2 and length, which may or may not actually exist in the representation of the object in a programming language. Objects can be represented in different ways depending on the desired level of detail. Adding, removing, or updating names of objects or their fields can be done by simply editing the text or using a find-and-replace operation.

C. Reassignments

Variables in a program are typically assigned new values over time. To show the evolving nature of programs, Snapdown allows users to draw both old and new values of variables using *reassignments*. Figure 7 shows a diagram illustrating a code segment such as `int a = 1; a = 2; int b = a;`. We represent crossed-out arrows with the syntax `-x>`, as it resembles what is drawn in the diagram as closely as possible.

D. Name-binding and resolution

Values in a program are typically associated with names. Snapdown takes an approach to name-binding that allows the user to easily associate names with parts of a diagram. Figure 8 shows an example of such a diagram with names for both variables (such as cs1 and cs2) and fields (such as semester, season, and year). Using underscores as field names, like in `_ -> cs1`, means that space should be reserved for a field but that no name should be drawn. Consequently, by using the names cs1 and cs2 in the definition of the Set<Course> object, the arrows from the Set<Course> object point at the Course objects.

Many programs also have duplicate names and objects with identical field names. In addition, some values may not be associated with a name. In both of these scenarios, users can include a # in variable names, both to help disambiguate between identical names and to give a reference to unnamed values. In Figure 8, the semester fields of both Course objects are named the same, so to disambiguate, we use the notation semester#1 and semester#2. #-disambiguation is helpful when we indicate that semester#2 is reassigned. In addition, the name #sem is associated with the Semester object representing Fall 2020 in the Snapdown snippet. Snapdown still binds the name #sem to this Semester object, but does not draw the name #sem itself.

The example in Figure 7 demonstrates an important feature of name-binding that applies specifically when a variable is reassigned. Here, when the user types `b -> a`, Snapdown infers that b should point to the reassigned value of 2.

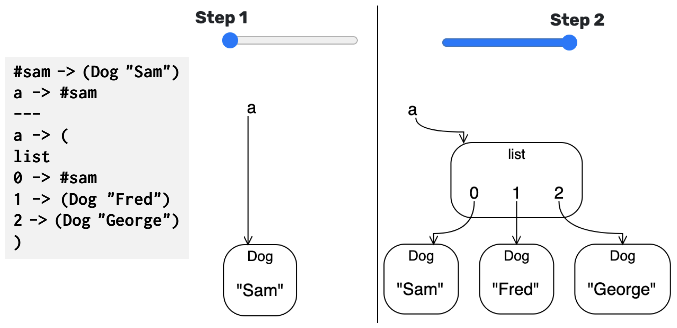
E. Stack frames

Figure 9 shows an example of Snapdown syntax for *stack frames* and the corresponding diagram. This diagram represents a call stack of three methods: topOfStack(), lowerInStack(), and main(), each of which can contain variable definitions. The method topOfStack() is likely within the class MyClass, and the this reference points at the MyClass object.

F. Multi-step diagrams

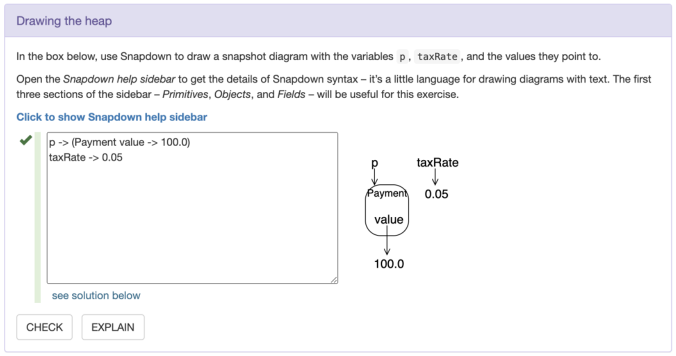
In many cases, a single diagram containing reassignments is not enough to communicate the evolving nature of a program. In Snapdown, users can draw multi-step diagrams by including dashed lines, `---`, between steps that illustrate code fragments.

One method to draw multi-step diagrams, used for example by Constrain, would be to animate transitions between snapshots. We take a different approach that does not rely on transitions to communicate the changes between each snapshot, and allows users to scroll between steps of a diagram at an arbitrary speed. Our approach requires that if the same object is present in multiple steps of a multi-step diagram,



```
#sam -> (Dog "Sam")
a -> #sam
---
a -> (
list
0 -> #sam
1 -> (Dog "Fred")
2 -> (Dog "George")
)
```

Fig. 10: Multi-step diagram demonstrating the need to keep objects in place.



Drawing the heap

In the box below, use Snapdown to draw a snapshot diagram with the variables `p`, `taxRate`, and the values they point to.

Open the *Snapdown help sidebar* to get the details of Snapdown syntax – it's a little language for drawing diagrams with text. The first three sections of the sidebar – *Primitives*, *Objects*, and *Fields* – will be useful for this exercise.

Click to show Snapdown help sidebar

```
p -> (Payment value -> 100.0)
taxRate -> 0.05
```

see solution below

Fig. 11: Completed Snapdown reading exercise.

it should not change position physically within the diagram. In the first frame of the diagram presented in Figure 10, the arrow is intentionally drawn much longer than usual to ensure a fixed position for this object.

IV. User Interface

Snapdown's user interface is its textual language, which can be embedded and integrated into many different situations and use cases. Here, we describe some scenarios in which Snapdown has been used and tested by students and instructors.

Snapdown web app: Snapdown is embedded in a simple, minimal web app that can serve as a sandbox for users to try out its features. This web app contains a textbox into which the user can type Snapdown syntax, as well as a diagram that live-updates to reflect user input. Users are also presented with a help sidebar that documents common Snapdown features. This help sidebar contains an explanation of one representative example for each core feature, the Snapdown text for that example, and the corresponding diagram. These examples can serve as templates for more complex diagrams users wish to draw: users can copy-paste the canned syntax examples and modify them as necessary. After the user has drawn a diagram, they have the option to export the diagram to an SVG file.

Pre-class readings: Snapdown is used in 6.031's required pre-class reading material. When preparing readings with embedded snapshot diagrams, instructors can type Snapdown syntax within embedded HTML `<script>` tags, and include the Snapdown library JavaScript file. Snapdown converts these HTML elements to SVG diagrams and embeds them within the final version of the reading that is released to students.

Pre-class reading exercises: Each pre-class reading also has a number of exercises, which are designed to be completed by a student in one sitting. We introduced Snapdown into some of these exercises. Students typed Snapdown in a setting similar to that of the web app: students type in a text box and a diagram is generated to the right of it. Clicking the "Check" button provides feedback on their diagram. A screenshot of a completed reading exercise is shown in Figure 11.

Collaborative in-class exercises: Snapdown was used by students to complete in-class exercises that involved drawing snapshot diagrams. Using a collaborative text editor designed for in-class pair programming, students worked in pairs to create snapshot diagrams that illustrate code segments. After each exercise was over, TAs gave feedback on diagrams drawn by each pair.

Usage by instructors: Snapdown was used by 6.031 instructors during class to explain concepts such as mutability and aliasing. Instructors also developed the in-class and reading exercises that involve Snapdown. Writing these exercises required instructors to iterate on different versions of starting and completed diagrams, and this iteration process involved writing and updating Snapdown syntax.

V. Technical Details

Snapdown is open-source[1], implemented in JavaScript and uses ELK [7] as its layout engine. ELK provides a number of layout algorithms, each with specific use cases in mind. Since we prefer arrows to be drawn downward, we chose *ELK Layered*, a layout algorithm that can support our requirement for directionality.

Snapdown uses an intermediate JSON representation to decouple parsing of Snapdown syntax from diagram layout and rendering. Snapdown syntax is first parsed, evaluated, and converted into this intermediate representation, which is then fed into the layout engine (in this case, ELK), and finally the laid-out diagram is drawn as an SVG in the browser. Because Snapdown outputs an SVG diagram, any web application with Snapdown embedded has great flexibility in post-processing or applying a custom CSS stylesheet to Snapdown diagrams.

We now describe how Snapdown visualizes the heap, how it draws stack frames, and how it handles multi-step diagrams.

A. Drawing the heap

Snapdown's implementation parses heap elements and performs name resolution on every named object, value, and field in the diagram. Through this name resolution, each object, variable name, and value is assigned a unique ID. ELK requires an input representation consisting of explicit nodes and edges, and this assignment of IDs gives ELK the information it needs to lay out the diagram. Assigning IDs is also crucial to the layout of multi-step diagrams.

[1]Implementation hosted at https://github.com/uid/snapdown, sandbox hosted at https://snapdown.csail.mit.edu/

B. *Drawing stack frames*

For diagrams with stack frames, our approach is more complicated, since these diagrams have arrows that point in multiple directions. Currently, we apply a global ELK setting that specifies a single direction across the diagram. At a high level, this setting does not allow edges to be drawn in multiple directions. Other such global settings are available, but they substantially alter the layout of the heap. To work around this limitation, we use PathFinding.js [17], an external pathfinding library, to lay out edges between the heap and the stack. We treat the stack and heap as separate diagrams and run Snapdown's full layout routine twice, with a different ELK directionality setting on each diagram. With this pathfinding library, we develop a set of constraints according to the nodes and edges that already exist in the diagram, and lay out the edges between the stack and heap. Finally, we combine the two diagrams together and draw the required edges.

C. *Multi-step diagrams*

Multi-step diagrams are the biggest challenge, since diagram elements need to have a consistent identity across steps. Whether we choose to animate them, or keep them in consistent positions as we have done, we would face the same challenges.

Each frame in a multi-step diagram represents a diff: for example, objects that are added or variables that are reassigned. In Figure 2, for instance, between Steps 1 and 2, the introduction of the two references `s` and `arr` is part of the diff between those two steps. Starting with the initial diagram (Step 1), each diff is applied to obtain a sequence of *individual diagrams*. Individual diagrams contain all the elements necessary for each step, but do not yet keep those elements in stable positions. While constructing individual diagrams, Snapdown preserves the IDs assigned to each diagram element. An object included in two distinct animation frames must have the same ID in both frames.

Next, Snapdown constructs a *combined diagram*. Every object and arrow ever drawn in the diagram will be included in the combined diagram. It is not sufficient to just make the last frame in sequence the combined diagram, since some arrows and objects may have been deleted prior to the last animation step. Finally, the individual steps of the diagram are created by deleting elements from the combined diagram that do not appear in each of the individual diagrams.

VI. Evaluation

We evaluated the learnability of Snapdown by having both students and instructors use it while taking or teaching 6.031. We also used Snapdown to replicate diagrams from courses at other institutions, and we describe our results.

A. *Presentation in readings and in class*

Across two semesters of emergency remote teaching in 6.031, instructors presented Snapdown in six pre-class readings and in four class sessions. Figure 12 shows an analysis of student responses to two exercises, named `heap` and `stack`,

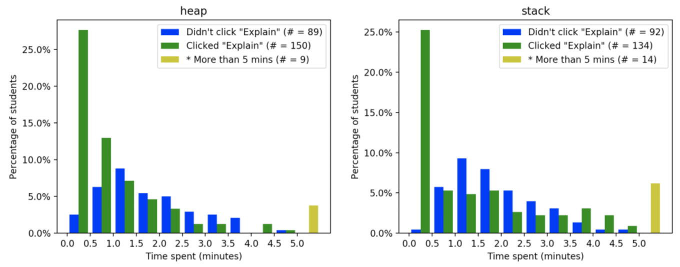


Fig. 12: Spring 2021, `heap` and `stack` Snapdown exercise analysis. * = Students who spent more than 5 minutes on the exercise; includes both students who clicked Explain and those who didn't.

```
seg -> (
Stroke
)
(Point)
(Color `BLACK`)
```

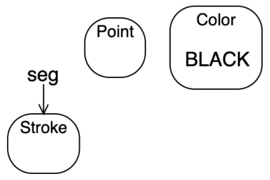


Fig. 13: Starting Stroke diagram.

presented in sequence in a reading assignment. During these two exercises, students were seeing Snapdown for the first time. Approximately 240 students were actively participating in the course at the time, and 226 students completed at least one of these two exercises. Each exercise has an *Explain* button that students can click if they wish to give up and see the answer without figuring it out on their own. We consider an exercise completed if either a student clicks the Explain button or they get the exercise correct. Student submissions were checked for correctness using a regular expression on the Snapdown source.

For each exercise, students who did not click the Explain button took an average of about 2 minutes to get the exercise correct (`heap`: 127.8 seconds, `stack`: 145.2 seconds). For both exercises, approximately 40 students clicked the Explain button almost immediately after opening the exercise. For students who spent more than 120 seconds on either exercise after opening it, we analyzed their sequences of answers. For both exercises, we believe these students were deliberately taking much more time to learn Snapdown by playing around with many of its features. The help sidebar also presented a large number of features, which may have been daunting to some students, especially to those who clicked the Explain button. In the text of the exercise, we specified which features were relevant, but it may have been better to only present the features we said would be useful.

One issue that came up frequently, and that was commented on both by 6.031 students and staff, was the lack of specific error messages when Snapdown could not parse user input. When a user enters invalid syntax, or enters a name that could not be looked up, the only error message shown is "Unable to parse Snapdown input." Displaying a more detailed error message, along with the syntactic or semantic nature of the error, is an area for future work.

```
seg -> (
Stroke
start->#1
end -> #2
color -> (Color `BLACK`)
)

#1 -> (Point x->5 y->10)
#2 -> (Point x-> 20 y->15)
```

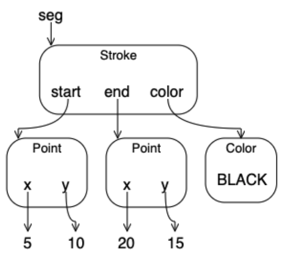


Fig. 14: Correct student-drawn Stroke diagram.

```
seg -> (
Stroke
startx -> (double 5)
starty -> (double 10)
endx -> (double 20)
endy -> (double 15)
color -> (Point)
(Color `BLACK`)
)
```

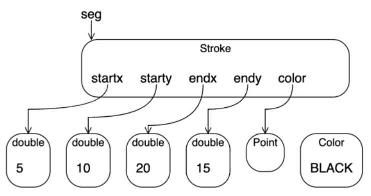


Fig. 15: Incorrect student-drawn Stroke diagram.

One of the authors of this paper presented Snapdown in a class session of 6.031 by using the main Snapdown web application. It is their opinion that compared with a tablet or whiteboard application, drawing with Snapdown resulted in much cleaner diagrams, and took approximately the same amount of time. During the Spring 2021 semester, 6.031 initiated a move from using Java to using TypeScript as its main programming language, and many snapshot diagrams had to be updated for the new language. We found that diagrams drawn with Snapdown were far easier to update, because its textual format allowed for easy global search-and-replace.

B. In-class exercises

Students in 6.031 completed collaborative exercises using Snapdown in pairs during class. One in-class exercise asked pairs of students to use Snapdown to draw a representation of a `Stroke` object in Java given its source code. Students were provided with the starting diagram shown in Figure 13. This exercise occurred after students had already seen Snapdown in reading exercises, and after they used it in class once before.

Figure 14 shows an example of a correct diagram drawn by a pair of students. A correct diagram has objects recursively nested within others: the `Stroke` object has fields pointing at `Point` objects, which themselves have fields. Such recursively nested objects were presented with corresponding Snapdown syntax in previous reading exercises, and examples were given in the help sidebar. All syntax examples, however, involved #-disambiguation—no examples were given similar to Figure 6 which directly included nested objects inside parent objects.

A total of 107 pairs of students worked on this exercise for twelve minutes. Approximately 56% of pairs drew correct diagrams. The pair shown in Figure 14 used #-disambiguation to achieve the object nesting; 6% of pairs did so in total. The other 50% of pairs put the Point objects directly inside the `Stroke` objects.

Figure 15 shows an example of an incomplete student-drawn diagram. This specific pair of students may have been unsure of how to achieve the desired nesting of objects, and directly included `startx` and `starty` as fields of `Stroke`. Approximately 14% of pairs of students had this issue. Another 4% of pairs of students drew diagrams without arrows between field names and objects, like `start` and the corresponding `Point`. Introducing students to previous examples of nested objects, either through previous exercises or in the help sidebar, may have resolved these issues.

The remaining 26% of pairs drew diagrams that demonstrated issues with conceptual understanding. Some pairs did not draw the names of fields such as `x` or `y`. In general, we consider diagrams without any field names (such as `x` or `y`) or values (such as `15` or `20`) to indicate a conceptual issue or a lack of effort.

C. Replicating diagrams from other courses

We used Snapdown to replicate diagrams presented in many courses, at a variety of institutions, that use diagrams throughout their lecture notes and lab assignments. We surveyed 11 introductory- and intermediate-level programming courses. We now describe our evaluation of Snapdown on diagrams from a representative sample of these courses.

CS 2112, Cornell [3]: We found 13 snapshot diagrams across lecture notes, 10 of which were readily expressible in Snapdown. One of the diagrams includes text written at arbitrary positions on the diagram, which Snapdown currently does not handle: the output SVG must be edited in order to achieve this effect. The other two diagrams that Snapdown could not draw contained a grid structure for a multidimensional array, and a representation of memory using hexadecimal addresses. These specific diagram layouts are not yet supported by Snapdown and are potential future additions to the system.

"OOP in Java", UCSD (Coursera) [23]: We found 14 snapshot diagrams across eight lecture videos in this course. We were able to replicate 13 of them using Snapdown. The only diagram we were unable to replicate was an invalid diagram, in which a variable name had an arrow pointing at another variable name as opposed to a value. We were also able to express two sequences of three diagrams each using Snapdown's multi-step diagram feature. We noticed that this course, among many others, prefer primitive values to be drawn directly in boxes: see Figure 16 for an example of this. Figure 17 shows our attempt to more closely replicate Figure 16 in Snapdown. To replicate this diagram, we added new syntax, `=`, to represent direct assignment of primitive values to variables. This syntax was easy to add and brought us much closer to visually replicating many diagrams not only from this course but from others as well.

CS 111, Wellesley [25]: We found 35 snapshot diagrams across two sets of lecture slides. All of them were readily expressible in Snapdown individually. CS 111 also includes a lab assignment instructing students to draw a sequence of nine snapshot diagrams that depict the state of a program at various

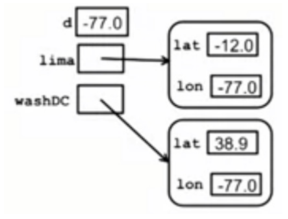


Fig. 16: Example figure from "OOP in Java" at UCSD

```
d = -77.0
lima -> (SimpleLocation
  lat = -12.0 lon = -77.0)
washDC -> (SimpleLocation
  lat = 38.9 lon = -77.0)
```

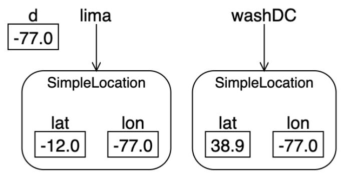


Fig. 17: Snapdown representation of diagram from Fig. 16

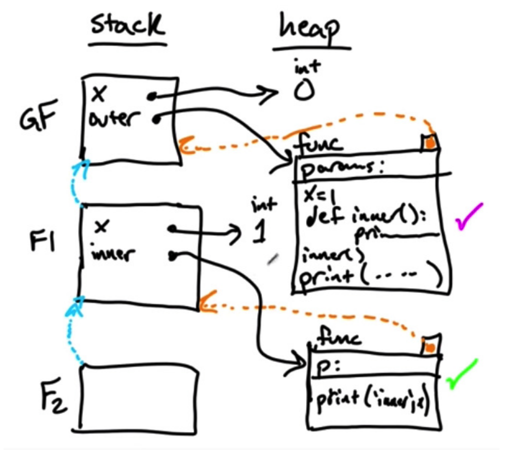


Fig. 18: Example snapshot diagram drawn in 6.009 (reproduced with permission)

points in its execution. We were able to draw all nine diagrams with Snapdown individually. We also tried expressing this sequence as a multi-step diagram in Snapdown. Snapdown has syntax for appending list elements, but does not yet have syntax for more advanced control of list elements necessary for these sequences. Future work includes adding syntax to support such mutations of lists and other objects.

6.009, MIT: Figure 18 shows an example of a snapshot diagram drawn in 6.009 Fundamentals of Programming at MIT, the prerequisite to 6.031. Contrary to 6.031 and the courses above, 6.009 does not have diagrams in online readings or a textbook—these diagrams are all drawn live by instructors during class sessions. Diagrams drawn in 6.009 contain stack frames and objects similar to those in 6.031, but with considerable visual differences. 6.009 uses Python, and being a lower-level course, its diagrams focus heavily on the inner workings of Python and how code executes. The diagram in figure 18 contains arrows pointing to stack frames, from other stack frames as well as functions on the heap. Snapdown cannot yet draw these arrows, but it can draw all other components in the diagram, modulo visual differences.

VII. Limitations and Future Work

ELK provides a number of layout algorithms—as described earlier, the ELK Layered algorithm best serves our needs. One limitation of this algorithm, however, is the lack of support for *hierarchy-crossing edges*. In other words, if we wish to have arrows point in two directions (i.e., down for arrows between heap objects, and right for stack-to-heap arrows), ELK Layered performs automatic layout only on arrows pointing in one of the two directions. We worked around this limitation by using an external pathfinding library. Our approach works because diagrams with stack frames, both in 6.031 and in other courses surveyed, are not very complex. For example, no diagram we encountered had more than four stack frames. Future work includes exploring other algorithms or solutions to this problem, especially if the need arises to draw more complex diagrams involving stack frames.

We can also add many features to our language. As described in Section VI-C, Snapdown currently does not have syntax to illustrate mutations of objects across diagram frames. In addition, some courses such as 6.009 place much more emphasis on stack frames, and have arrows pointing to stack frames to demonstrate the inner workings of a specific programming language. Snapdown is designed to be language-independent, but in the future we can add customized modes for specific programming languages. Finally, expanding our syntax for multi-step diagrams and our use of the pathfinding library are fruitful avenues for future work.

We also identified issues students had with using Snapdown. For example, some students had difficulties with drawing recursively nested diagrams, and we hypothesize that adding more such examples to the help sidebar will resolve this problem. Future work includes testing these hypotheses with targeted experiments and exercises.

VIII. Conclusion

Snapdown was developed based on our approach to teaching introductory software engineering, but we have shown that it is a general tool for drawing snapshot diagrams. The textual nature of the Snapdown language promotes usability in a wide range of situations, both in the classroom and in web applications. In addition, when used in readings or course material, updating a diagram becomes as easy as a find-and-replace operation. During the COVID-19 pandemic, drawing a diagram on paper as part of a class was no longer easily possible for students. Yet with Snapdown, students were able to complete collaborative diagram-drawing exercises, and instructors were able to use it to explain concepts during class. Finally, Snapdown's ability to handle multi-step diagrams gives it a compelling use case in course materials and in exercises given to students. We believe Snapdown makes snapshot diagrams easier to create and maintain, and will benefit courses in which drawing diagrams is a learning objective for students.

References

[1] Sebastian Baltes and Stephan Diehl. “Sketches and Diagrams in Practice”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 530–541. ISBN: 9781450330565. DOI: 10.1145/2635868.2635891. URL: https://doi.org/10.1145/2635868.2635891.

[2] *Constrain - a JS (ES6) library for responsive, animated figures, based on declarative constraint solving*. URL: https://andrewcmyers.github.io/constrain/.

[3] *Cornell University, CS 2112 Fall 2020: Object-Oriented Design and Data Structures (Honors)*. URL: https://www.cs.cornell.edu/courses/cs2112/2020fa/lectures/lecture.html?id=objects.

[4] Andrew R. Dalton and William Kreahling. “Automated Construction of Memory Diagrams for Program Comprehension”. In: *Proceedings of the 48th Annual Southeast Regional Conference*. ACM SE ’10. Oxford, Mississippi: Association for Computing Machinery, 2010. ISBN: 9781450300643. DOI: 10.1145/1900008.1900040. URL: https://doi.org/10.1145/1900008.1900040.

[5] *diagram.codes*. URL: https://www.diagram.codes/.

[6] Toby Dragon and Paul E. Dickson. “Memory Diagrams: A Consistant Approach Across Concepts and Languages”. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE ’16. Memphis, Tennessee, USA: Association for Computing Machinery, 2016, pp. 546–551. ISBN: 9781450336857. DOI: 10.1145/2839509.2844607. URL: https://doi.org/10.1145/2839509.2844607.

[7] *Eclipse Layout Kernel*. URL: https://www.eclipse.org/elk/.

[8] Emden R Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*. URL: https://www.graphviz.org/pdf/dotguide.pdf.

[9] *GitUML*. URL: https://www.gituml.com/.

[10] *Graph Visualization Software*. URL: https://www.graphviz.org/.

[11] T. Dean Hendrix et al. “Do Visualizations Improve Program Comprehensibility? Experiments with Control Structure Diagrams for Java”. In: *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’00. Austin, Texas, USA: Association for Computing Machinery, 2000, pp. 382–386. ISBN: 1581132131. DOI: 10.1145/330908.331890. URL: https://doi.org/10.1145/330908.331890.

[12] Mark Holliday and David Luginbuhl. “Using Memory Diagrams When Teaching a Java-Based CS1”. In: *Proc. of the 41st Annual ACM Southeast Conference*. ACMSE ’03. 2003, pp. 376–381. URL: https://dl.acm.org/doi/abs/10.1145/971300.971373.

[13] *IntelliJ UML Class Diagrams*. URL: https://www.jetbrains.com/help/idea/class-diagram.html.

[14] *Markdown*. URL: https://daringfireball.net/projects/markdown/.

[15] *Mermaid*. URL: https://mermaid-js.github.io/mermaid/#/.

[16] *ObjectAid for Eclipse*. URL: https://objectaid.com/.

[17] *PathFinding.js*. URL: https://qiao.github.io/PathFinding.js/visual/.

[18] *Penrose*. URL: http://penrose.ink/.

[19] *PlantUML*. URL: https://plantuml.com/.

[20] *Python Tutor*. URL: http://pythontutor.com/.

[21] Jorma Sajaniemi, Marja Kuittinen, and Taina Tikansalo. “A Study of the Development of Students’ Visualizations of Program State during an Elementary Object-Oriented Programming Course”. In: *Proceedings of the Third International Workshop on Computing Education Research*. ICER ’07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 1–16. ISBN: 9781595938411. DOI: 10.1145/1288580.1288582. URL: https://doi.org/10.1145/1288580.1288582.

[22] *Structurizr*. URL: https://structurizr.com/help/code.

[23] *UCSD, Object-Oriented Programming in Java (Coursera)*. URL: https://www.coursera.org/learn/object-oriented-java.

[24] *Visual Communication Software To Make Pro Diagrams - OmniGraffle for Mac*. URL: https://www.omnigroup.com/omnigraffle.

[25] *Wellesley College, CS111 Computer Programming*. URL: http://cs111.wellesley.edu/~cs111/archive/cs111_spring16/public_html/index.html.