INTRODUCTION
Recent years have witnessed an upsurge in the popularity of design thinking, a phenomenon which has given rise to a vast number of projects that span traditionally separate disciplines. Prominent among these are applications that combine programming and design principles to create new approaches to problems. However, it is relatively rare to see these principles applied to the process of coding itself.

Traditionally, programming is an exhausting process: Type some code into an editor, hit compile, search for syntax errors, hit compile again, search for conceptual errors, and repeat. This disconnect between the code and its execution is significant both in terms of the resources it wastes (time and energy) and its implications for programmers' approach toward code. Rather than developing a conceptual understanding of the problem, all too often, developers simply “hack together” a working system with little attention to its design. This tendency is especially problematic for novice programmers, for whom access to common resources such as abstract data types may inspire more confusion than clarity. Though some attempts have been made to visualize common algorithms such as MergeSort and Dijkstra’s algorithm, programming has remained a highly text-based, non-visual process. To those unfamiliar with the process, the difficulty of translating a user’s mental model for a solution into code can be a major deterrent to entering the field.

In this paper we present Conjure, a tool for visualizing the results of arbitrary code during its execution. Inspired by canonical programming interview problems, our editor provides insight into the structure of the user’s own solution to a problem and offers a visual means for debugging. Conjure combines previous work in visualizing set algorithms with research into reduced development environments to create an application that allows even novice programmers to compose and visualize solutions to problems suitable for their skill level. This focus was motivated by a desire to decrease the obscurity of code for incoming students of Computer Science, encourage them to fully understand each line of their own code, and provide a means to visually explore various routes to a solution.

Though our application has parallels to both existing data structure visualizations and code evaluators, the final result is novel in its user-centric approach to data structure and algorithm visualization. Our design leverages the flexibility of a custom API to show the effects of each individual statement a user writes and highlight potential issues with a particular approach to the problem. To demonstrate the suitability of the application for this particular domain, we have used Conjure to visualize two common algorithms, Reverse a Linked List and Merge Two Sorted Arrays. The latter purposefully contains two bugs, the source of which Conjure can help identify via step-by-step visualization.

We first review related research in data structure visualization and Computer Science education and follow with a detailed discussion of Conjure’s visualization and implementation. We then describe Conjure’s results for the two algorithms described above and conclude with an overview of potential future work.

RELATED WORK
Visualizations of algorithms and data structures have a relatively rich history as well as modern, active research communities. Numerous studies show that visualizations with an emphasis on user interaction enable students to better grasp concepts [11] [12]. Understanding the entire field of educational software, as well as its strengths and pitfalls, helps guide our development. Gomez-Albarran categorizes software for Computer Science education into four types: [3]

- Reduced Development Environment
  Tools that reduce the complexity of programming environments in order to be more friendly toward novice students.

- Example-based Environments
  Tools that use examples that students can build off of and learn from.

- Visualization Tools
  Tools that visualize code behavior and properties.

- Simulation Environments
  Tools that help the user understand code execution via a virtual world.
While Gomez-Albarran classifies a given tool into one of these four categories, tools often span more than one category. Take for example, Karel the Robot. Karel the Robot, developed by Richard E. Pattis, according to Gomez-Albarran belongs to the Simulation Environment category, as it constructs Karel’s universe. However, modern-day implementations of Karel the Robot tend to also include a visual manifestation of the code [10]. Thus, it spans both visualization tools and simulation environments.

The University of Macdeonia, Greece employed a particularly interesting reduced development environment known as AnimPascal [13]. One of the most interesting features of AnimPascal is its ability to record user input and results. This enables instructors to gather data about how students code, what bugs they encountered, and how they approached debugging.

Visualization software includes Animal, JAWAA, Data Structure Visualizations, and Online Python Tutor among others.

- Animal: a tool that enables designers to create visualizations in three ways: using AnimalScript (a custom scripting language for developing animations), a graphical editor, and a Java API.

- JAWAA: a tool that allows designers to either use a graphical editor or the JAWAA scripting language to create visualizations using primitives such as circles, lines, text, and rectangles [5]. The resulting outputs are animations that can be embedded in Java web applets. JAWAA is geared toward instructors who wish to create visualizations to supplement their course materials.

- Data Structure Visualizations is a project by David Galles of the University of San Francisco that takes common algorithms and data structures and provides visualizations with the option of specifying inputs and parameters [8]. He provides an API for generating such visualizations, but notes that the algorithms become convoluted when viewed in the animation source code.

- Online Python Tutor is a web-based Python visualization tool developed by Philip J. Guo [4]. Online Python Tutor allows users to input fully-featured Python code directly into a web browser to construct a visualization. Specifically, it draws the stack and variables and allows the user to step backward and forward in the program’s execution. In addition, it allows users to easily embed these visualizations within their own websites.

METHODOLOGY

In the development of our visualization software, we drew from the advice of Naps, Cooper, et al. as well as from the taxonomy of Gomez-Albarran and Ihantola et al [1] [2] [3].

Design Criterion

Scope

In dealing with the scope of what the project could accomplish, we decided to make our visualization tool straddle the line between course-specific and domain-specific scope.

Course-specific scoping refers to tools that would be suited towards producing material for a single type of course, whereas domain-specific tools target a broader audience [2]. We do so by focusing on the visualization of algorithms and data structures, rather than lower-level alternatives such as stack frames, as in Online Python Tutor.

Integrability

Often, the difficulty of integrating software impedes its adoption. That is, the more complex software is to find, download, install, learn, and develop, the less likely a potential consumer would be to use it [1]. In order to minimize this overhead, we decided to build our software as a web application using a rough approximation of JavaScript syntax. The choice of web application allows for platform independence and reduces user setup time. In addition, we believed that the choice of JavaScript syntax was practical in its familiarity to a large audience.

Interaction

Typically, the more interactive the visualization, the more students will learn from it [12]. To this end, we wished to produce a tool that was immediately available to the student, rather than a tool used by a professor or teacher to generate visualizations for students. Additionally, we wanted to ensure that the user would have a direct influence on the visualizations. Unlike the USF Data Structure Visualizations, rather than having the students simply perform the operations by click buttons, we decided to take an approach similar to Online Python Tutor. This approach focuses on interaction, in particular enabling the user to visualize the operations in his or her own code.

IMPLEMENTATION

Conjure’s implementation can be split into two high-level parts: data structure visualization and code interpretation. For the visualization, we created our own API for common data structures. The API was developed in JavaScript using the toolkit D3.js [15], leveraging the toolkit’s support for data-driven document manipulation and interactive capabilities. We implemented the remainder of the frontend using JavaScript, HTML, CSS and the DOM selection functionality of the jQuery toolkit. To support the visualization of arbitrary code, we wrote a custom JavaScript interpreter adapted to recognize the Conjure API. This interpreter was also implemented in JavaScript.

Data Structure Visualization

To support our goal of creating a tool useful for students and educators alike, we chose to focus on commonly-used data structures with intuitive abstractions. To keep the project within a reasonable scale, we limited this iteration of the project to list-based data structures containing number (decimal or integer) and string values. Foremost among these is the array, likely the most widely used data structure in programming, which we define for our API to be a sequentially-indexed, variable-length list of objects. Natural extensions of this are the queue, a first-in-first-out (FIFO) data structure, and its counterpart the stack, a last-in-first-out (LIFO)
data structure. Both of these data structures have applications in many common algorithms. Last is the linked list, an ordered list of nodes consisting of a variable value and a reference to the next node in the list. Linked lists often prove tricky for novice programmers due to the abstract nature of object references. However, that abstraction makes the data type useful for a wide variety of applications, which makes it a worthwhile choice for visualization.

In visualizing these data structures, we had to consider limitations of the medium, the most important of which was limited space. To display values that overflow the boundaries of their containing elements, we determined the highest number of characters that could fit within the element, truncated values with lengths exceeding that number, and appended an ellipsis to indicate the truncation. To view the full value for a truncated element, users may hover over the element. At this point, the hovered element will take on a slight glow, and all other elements’ opacity will be reduced so that the full value is easily legible.

A related consideration was the positioning of data structures within the canvas. To ensure multiple structures can be displayed within the fixed-size canvas at any given time, the interpreter assigns each structure a 500x130-pixel space, with margins to accommodate minor variations in height. As these data types can in theory contain an infinitely large number of elements, this presented an issue of scale. To address this, we implemented a collapsed version of each data structure that would activate when the representation would otherwise cross the margin of the canvas. In this, we had to ensure that we did not violate the common conventions of the visualized structure. Thus, for data structures for which all values are accessible at any given time, we implement an interactive hover mechanism to display the full value. Details of this implementation per data type can be found in the subsections below.

**Arrays**

In line with the standard representation of an Array in reference material, the Conjure VArray, or “Visual Array,” is as a horizontal list of connected elements. Each element is represented by a pale blue square containing its value and identifiable by its index, which is drawn just below the square in brackets. Starting from the left-hand size of the canvas, each element is displayed in increasing order of index, from which the total length of the array can be inferred. When users add, remove, or change the value of elements, only the necessary elements are redrawn, and have the appearance of fading in or out as appropriate.

If the number of elements in the array exceeds nine, the maximum number of full squares that can be displayed on the canvas at once, the left-most elements are collapsed into rectangles of the same height, but only one-tenth the width. One downside of this approach is that it hides the value of the element from the user. To mitigate this, the hover mechanism for arrays restores hovered elements to normal size, revealing the value. This allows the data type to display as many as one hundred elements at once without rendering their values inaccessible to the user.

**Queues**

Though similar to the VArray in its use of horizontally ordered rectangles to represent elements, the VQueue has several key differences. A canonical queue can only be modified in one of two ways: by pushing an element onto the end of the queue, or by popping an element off the head. To reflect this abstraction, we collapse the width of all elements in the middle of the queue, regardless of the length of the data structure, and do not implement a reveal-upon-hover mechanism.

In comparison to the array, this reduces the total length of the visualization for any number of elements greater than two. However, without an extra measure, enough elements would still cause the visualization to cross the end of the canvas. In addition, the number of collapsed elements may not be immediately obvious for a large number. To handle this, a bracket and label is included below the reduced-width elements to indicate their number, and the number of collapsed-width elements displayed is capped at fifty. If additional elements are pushed onto the queue, the bracket label is changed from "x elements" to "x collapsed elements" and the number x is incremented, but the number of collapsed elements displayed stays the same.

**Stacks**

Given their complementary nature, the representation of the VStack is similar to that of the VQueue. Differences between the two stem from the contrast between a FIFO and a LIFO data structure. While both the head and tail of a queue are relevant to the programmer, only the top element of a stack is accessible at any given time. Therefore, the VStack uses a similar collapsing mechanism to the VQueue, but with the bottom element collapsed along with the rest of the middle elements. Also notice the difference in semantics between the two types: a stack has a "top" and "bottom," while a queue has a "head" and "tail": terminology that stems from their physical-object parallels, and explains the common representation of stacks as vertical and queues as horizontal, despite the related nature of their abstractions. In keeping with this convention, the VStack is displayed as a literal stack of collapsed elements. The vertical nature of the type introduces some difficulty due to the significantly more limited space along the y-axis as opposed to the x. However, this is a problem easily solved by reducing the number of elements before the representation of collapsed elements falls to the bracket label to five.

**Linked Lists**

Perhaps the trickiest of our chosen data structures, singly-linked lists are also the most visually complex. Fortunately, a conventional representation does exist: a list of rectangular elements, as in the earlier data structures, but connected by arrows representing pointers. Our VLinked class adheres to these conventions, and in doing so demonstrates a key advantage of creating visualizations with SVGs rather than `<div>`s or other traditional DOM elements. Although the
number of CSS classes applicable to SVGs is extremely limited compared to those applicable to `<div>`s, the range of possibilities available with SVGs is simply higher, as evidenced by the pointer arrows in each VLinked element. Each element, defined as a Node in our pseudo-JavaScript syntax, consists of a SVG rect containing the value, a thinner SVG rect, and a pointer from this thinner rect to the next element in the list.

As in the other data structures, excess elements are collapsed into the central element. However, as each Node is significantly wider than an array element, values of collapsed elements are displayed as a list below the center element upon hover rather than revealed via expansion. In addition, each VLinked is preceded by a labeled arrow to the head element and ends with an arrow to a smaller rectangle with an X through the center, which is a common representation for a null value. In the code, users manipulate the linked list via the traditional method of setting the `value` and `next` attributes of nodes, which is reflected in the visualization by redrawing the list whenever it is modified.

**User Interface**

Although the novelty of the application lies in the parsing and resulting visualization, an effective editor and canvas are key in ensuring that functionality is readily available to the user. To that end, the majority of screen real-estate in the application is dedicated to these two elements, with the editor on the left and canvas on the right. We aimed to create a visual distinction between the elements of the application associated with the code and those associated with visualization. As such, code/data structure-related text is displayed in Monaco, the default monospace font on OS X and a familiar typeface for many programmers, while the remainder is displayed in Helvetica Neue, a common “neutral” font of choice for web applications. This dichotomy is particularly noticeable in the editor, which uses purely Monaco and resembles a traditional programming-oriented text-editor. Meanwhile, the canvas features brighter colors in the form of the data structures themselves and responds instantaneously to hover, which draws attention to the canvas.

Aside from these two key elements, for the overall look and feel of the application, we aimed to balance minimalism with approachability. The sans-serif fonts contribute to the feeling of approachability, as does the subtle texture in the background and rounded edges of the editor. In addition, we ensured that only the necessary content is on the main page of the application; finer details are relegated to the User Manual and API Reference pages accessible via the links to the upper-right. To avoid having to load a separate window whenever the user wants to consult these pages, both these sections are implemented as `<div>`s that fade in and out on top of the canvas and editor. Finally, our two code examples are instantly accessible via two tabs on the left-hand side of the editor, which slide out slightly upon hover to indicate that they are clickable objects.

**Backend**

**Interpreter**

The interpretation is performed in a series of steps.
• Preprocessing and Parsing: The user’s code is stored and modified to remove comments as well as to regularize spacing. Additionally, it is recursively broken down into logical statements, based on the end of logical statements, designated by the semicolon and existence of programming constructs such as for loops, while loops, if-else blocks, etc. Effectively, these steps construct a parse tree.

• Evaluation: Each individual statement was syntactically analyzed and the corresponding action was performed.

Code Evaluation
There were several difficulties in dealing with the code evaluation.

• Dealing with Asynchronous D3 Commands: There was some initial difficulty in being able to serialize the actions and perform the visualizations while allowing for the proper animation. The first iteration of the program would either simultaneously visualize the actions or would end up having artifacts similar to race conditions. We decided to deal with this by effectually constructing a queue whose next step is only evaluated once the previous animation is finished.

• Stepping functionality: After constructing the queue, implementing stepping became much simpler. Instead of continuously popping statements from the queue, we simply pop one statement at a time.

• Parsing All Features: Given the set of all features that could have been included, the space of potential functionality was extremely vast. To handle this, we decided to limit the scope of allowable syntax. For example, our interpreter does not support the increment operator (i.e. i++) or switch statements.

EXAMPLES
To demonstrate potential applications of Conjure, we visualized two common algorithms: reversing a linked list and merging two sorted arrays. The first used a slightly unconventional approach to the problem that separates the problem into two distinct lists: the original list and the new, reversed list. We chose this approach for its more intuitive nature when compared to the original.

In visualizing the second algorithm, we purposefully left two errors in the code to demonstrate the application’s potential in identifying and correctness bugs in program behavior. Specifically, the error was the omission of two increment statements (i = i + 1 and j = j + 1), which caused the program to enter an infinite loop. In the visualization, the user can physically see the results of the line that is being executed repeatedly, as a particular element is repeatedly appended to the end of the solution array. In addition, the console indicates which line is looping infinitely. In our experience presenting the demo to potential users, we found these cues effective in helping identify the source of the errors.

DISCUSSION
User Reaction/Feedback
Following our first presentation of the project concept, we were received with both great enthusiasm for its potential and reservations about its feasibility. Many commenters suggested the potential of the project for teaching Computer Science, while others expressed concerns that executing the project successfully would require writing a compiler, which was a lofty goal for a project of only a few weeks. As we sketched out and implemented early prototypes, this feedback was central to the growth of the application, both in terms of its high-level direction and in choosing a reasonable scope. We quickly moved away from our initial plan of providing a database of interview questions and solutions and instead focused on the essential data structures for demonstrating the value of the concept for education. In this, we drew inspiration from past teaching experience section leading for Stanford’s CS198: Teaching Computer Science program, to choose data structures that either had compelling physical parallels (arrays, stacks, and queues), or posed a particularly challenging task that visualization could help mitigate (linked lists).

Our second presentation of the project involved a functional prototype and demo of the two programming problems described in the prior section: Reverse a Linked List and Merge Two Sorted Lists. This was again met with great enthusiasm and encouragement to polish the project for use in the CS198 program. Perhaps more notably, it also sparked a significant number of attendees to mention how it reminded them of designer Bret Victor’s talk Inventing on Principle[14]. After watching the video, we realized just how great the potential for code visualization was, but also of the many limits of our prototype.

FUTURE WORK
Other Data Types

Primitives and Pointers
Although Conjure supports data structures that include primitives and pointers, it does not support their independent visualization. In future work, space in the canvas could be reserved for values of local variables, which would allow the user a more complete picture of the code when data structures are not explicitly used. In addition, pointers are a particularly challenging topic for programmers encountering them for the first time. Visual encodings for pointers could include arrows drawn from a “pointer” cell to its pointee variable, or an interactive a highlight-on-hover mechanism. However, implementing this would require a strict definition of what constitutes a pointer; whereas languages such as C and C++ have a clear definition of a pointer, others, such as Java and JavaScript, use implicit object references instead, which may not be as intuitive to visualize using arrows.

Graphs, Maps, and Strings
Support for graphs, maps, and strings would greatly enhance the flexibility of the API and its potential to support truly arbitrary code. While graphs and strings have relatively straightforward conventional representations, however, maps can vary widely in their level of abstraction. Possible visualizations could reflect the backend implementation of the map, for in-
stance via hashing or a tree structure, or adhere to a higher-level, more abstract visualization of keys and values.

**Arbitrary Data Types**

Particularly ambitious future work could tackle the visualization of arbitrary data types or classes. This extension would be particularly challenging in that these data types may have a strong physical parallel (for instance, a Motorcycle class), or lack any sort of natural physical representation (an abstract, user-defined collection of data). As the user’s desired representation would be difficult to infer from the code, one solution would be to give the user input in the final visualization – for instance, a choice of three high-level types. Another interpretation of this topic would be to remove the need to specifically reference our API. While this would be challenging using JavaScript’s variable typing scheme, as the only clearly defined data type is the array, it could work very well for languages with strict typing, such as Java.

**Nested Structures**

As Conjure currently only supports structures of numbers and strings, another valuable extension of the project could support nested structures – for instance, arrays of linked lists or sets of sets. High-level, low-detail versions of nested data types are relatively simple, but as the granularity increases, so too does the complexity. One possible approach would be to add support for zooming, such that “thumbnail” versions of data structures are displayed when fully zoomed out and higher-resolution versions appear when zoomed in.

**General Functionality**

Our project lends itself to expansion into a more highly featured development environment while maintaining the reduced environment appearance. To this end, there could be several improvements.

- **Line-highlighting**: As a debugging feature, highlighting the line currently executing would be more helpful in following the flow of program execution than the current console that is in place.

- **Syntax Checking**: Adding support to let the user know when their input code is formatted incorrectly would help users reduce the errors from miswritten code to focus more on algorithmic and structural misunderstandings.

- **Breakpoints**: Allowing the user to add breakpoints would enable users to replay their steps without worrying about missing key portions of the visualization.

- **Adjustable Visualization Speed**: The current visualization provides little control over the animation delay. Adding functionality to adjust the speed of the animation would allow users to speed up portions of the visualization that they understand and slow down portions that they do not, saving valuable time.

- **Example Database**: Providing a more extensive database of examples would help students draw from others’ previous experience.

- **Multi-Language Support**: Adapting the program to handle support for other languages, for instance Java or Python, would increase the audience size and reduce the learning period for users unfamiliar with JavaScript.

**Ideal Functionality**

- **Complete Interpreter**: Our current interpreter is limited in its functionality and supported syntax. From a fundamental level, the program could be tackled in a more effective and comprehensive approach. An ideal method might be to lexically categorize each portion of the code, generate a parse tree, and then evaluate the statements in that tree. Additionally, support for a fully object-oriented approach would greatly increase the number of potential use cases for the system.

- **Live Display**: One major improvement could feature a live view of the algorithm as a user is coding. For example, a corresponding visualization would be constantly updated as the user types code into the editor, similar to Bret Victor’s demonstrations in *Inventing on Principle* [14].

**REFERENCES**


