Bridging the Syntax-Semantics Gap of Programming

Theo B. Weidmann*
ETH Zurich
Switzerland
tweidmann@ethz.ch

Sverrir Thorgeirsson*
ETH Zurich
Switzerland
sverrir.thorgeirsson@inf.ethz.ch

Zhendong Su
ETH Zurich
Switzerland
zhendong.su@inf.ethz.ch

Abstract

Computer programming typically requires people to describe operations in a formally specified textual language. Unfortunately, working with syntax is a significant cognitive load, making programming difficult for beginners and time-consuming for professional developers. In response to this, contemporary research often focuses on abstracting or improving the process of composing code. We believe, however, that one fundamental reason why programming is difficult is the disconnect between the symbols and metaphors used in code and the mechanics they represent. Programming languages use abstractions whose superficial similarities to natural language neither effectively help users understand programs nor enable them to work creatively. To tackle this fundamental limitation, this paper introduces a new language based on a novel programming-by-demonstration paradigm that (i) enables users to experiment and test their programs, (ii) allows describing complex operations without the need to learn any syntax, and (iii) always displays an approximation of the program state while programming a new operation. We explain the rationales behind our new approach and present our design and implementation using illustrative examples and a supplemental video recording.

CCS Concepts: • **Human-centered computing** → *Visualization systems and tools*; • **Applied computing** → *Interactive learning environments*; • **Software and its engineering** → **General programming languages**.

Keywords: programming-by-demonstration, visual programming, non-textual programming, end-user programming, learnable programming

1 Introduction

Most computer programs today are created using formally specified programming languages. To manipulate the state of the computer, programmers need to compose a textual representation of a program that strictly adheres to a formal grammar. However, the words of the textual representation have little to no inherent connection to the state change they provoke [17]. Consequently, programmers have to map abstract syntax to concrete state changes in their mind, while rarely seeing the program's state. This creates obstacles not only for people learning to program but also for professional software developers.

In the computer science (CS) education community, syntax has been considered a cognitive load [18], a barrier to learning [7, 29], and a source of negative emotions such as frustration and boredom [4]. In industry, developers often have a clear idea of what they want to achieve, but syntax makes it time-consuming to convert high-level ideas into executable code; studies have found that as much as 35% of a developer's worktime is spent searching for code examples online [12, 37]. Research such as Heyman et al. [12] proposes solutions for enhancing code auto-completion to also take the developer's intent into consideration. Other studies have advocated for new auxiliary software tools, such as debuggers or visualizations.

In our view, many of the recently proposed innovations, while useful, are too incremental and do not address the root cause that explains why programming is difficult, namely what has been called "the tension between human comprehension and computer interpretation" [16]. We believe that common programming languages use abstractions that obfuscate the meaning of code, and that those abstractions are concealed by code's superficial similarity to natural language. As a remedy, a radical new approach to programming is needed.

In this paper, we present a solution to this problem by introducing a new version of the programming language *Algot* [30] that uses programming-by-demonstration and a visual representation of the state. With this system, we make three key contributions:

- We present a system for constructing complex operations that does not require learning textual syntax.
- We show how a programming-by-demonstration environment might display an approximation of the program state and allow it to be manipulated.
- We introduce a new programming paradigm based on recursion that decreases control flow complexity in three simple stages.

In the remainder of this paper, we first offer the reader a glimpse at Algot (Section 2). Then we describe important background concepts (Section 3) and explain the motivation and design rationales behind our work (Section 4). Following this, we explain in detail how our web-based implementation of Algot works (Section 5) and support our explanation with examples (Section 6) and a short, silent video recording[1],

*Theo B. Weidmann and Sverrir Thorgeirsson are co-primary authors.

[1]Algot is a highly interactive system and any textual description of it is limited. For the reader's comprehension, we highly recommend reviewing this video before reading the system description.

```python
def bubblesort(a):
    swapped = False
    for n in range(len(a)-1, 0, -1):
        for i in range(n):
            if a[i] > a[i+1]:
                swapped = True
                a[i], a[i+1] = a[i+1], a[i]
        if not swapped:
            return
```

Figure 1. An implementation of bubble sort in Python.

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def insert(item, tree):
    if (item < tree.value):
        if (tree.left != None):
            insert(item, tree.left)
        else:
            tree.left = Node(item)
    else:
        if (tree.right != None):
            insert(item, tree.right)
        else:
            tree.right = Node(item)
```

Figure 2. An implementation of binary tree insertion in Python.

available at https://vimeo.com/729159376. Finally, we briefly discuss future developments (Section 7) and our vision for its use in education and industry (Section 8) before offering concluding thoughts (Section 9).

2 Motivating Examples

We motivate our work by providing a brief high-level comparison between a popular textual language (Python) and Algot. This is merely meant to give the reader a taste of Algot — we will give an exact description of the system and examples later in the paper.

Fig. 1 shows an example implementation of the sorting algorithm *bubble sort* in Python. While the algorithm is conceptually simple, the code is not straightforward; the programmer needs to work with and understand multiple constructs that are typical for implementations of bubble sort in textual programming languages, such as iteration structures, control structures, and assignment statements [15]. For beginners, this is far from ideal, as research indicates that they struggle with understanding "the iteration pattern used in the comparison procedure, how nesting loops work and the meaning of both the variables and the counters used" [15, 33]. The same research indicates that beginners also seemingly

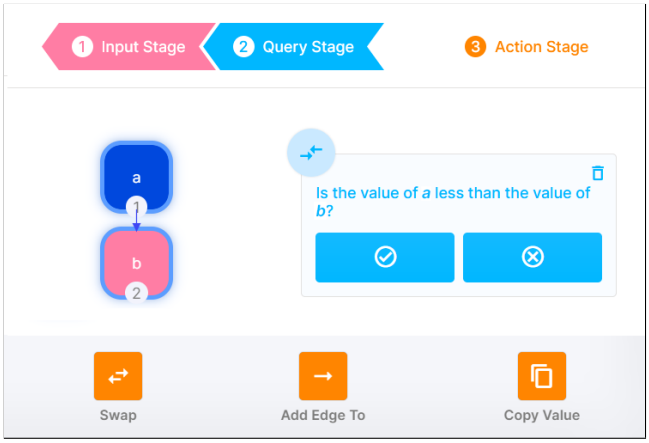

Figure 3. A partial screenshot of the implementation of bubble sort on a linked list in Algot. Using pattern matching, the programmer has selected the root (1) and then its child (2) of a two-node tree and then defines a query (right) to compare their values. The three stage input-query-action panel is visible on the top. The action phase is active.

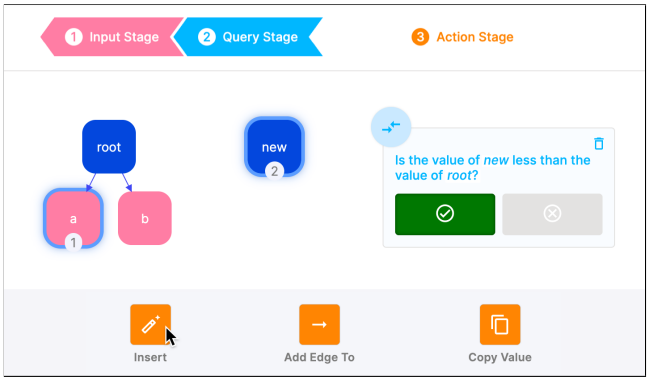

Figure 4. A partial screenshot of implementing binary tree insertion. The new value is inserted into either the left or the right subtree after comparison.

lack the ability to initialize the counters used and pay no attention to "initializing or terminating the whole sorting procedure."

In contrast, we believe that Algot brings the implementation of algorithms like bubble sort closer to the actual cognitive mechanisms that humans use to understand and define algorithms, namely via simple-to-understand queries, visual pattern matching, and a three-stage input-action demonstration process that is congruent with how people explain algorithms to each other. Fig. 3 shows a snapshot of how these constructs look when the language is used to implement bubble sort.

As a visual language, Algot can offer additional benefits. Many computer science students and professional developers are accustomed to visual graph representations from classes

and textbooks. This representation of data structures is more convenient for human reasoning than textual source code and is thereby more accessible. Algot tries to automatically layout graphs in a customary shape to support the programmer in their work. Fig. 2 shows a Python implementation of inserting a value into a binary search tree. For comparison, Fig. 4 shows part of the implementation of the same algorithm in Algot.

Later in this paper, we will describe in detail how Algot works and offer comprehensive illustrative examples that show some of its capabilities.

3 Background

In this section, we introduce background concepts that inspired the new version of Algot and which we refer back to later in the paper.

3.1 Algot

We expand on our past work Algot [30], a visual programming language aimed at beginners to explore foundational programming concepts. AlgotâĂŹs "language syntax, both in terms of composition and outcome, imitates its visual semantics." That is, data structures are presented visually, and the user manipulates the program state using the mouse only — by clicking and using drag and drop. The result of any action is immediately visible.

A *history bar* displays all the commands the user has executed. The entirety of the executed commands can be understood as a program. Using a rule table, the system detects patterns and offers to automatically repeat similar operations, eliminating the need for control loops.

We proposed two main applications. First, the environment can be used to test students' understanding of an algorithm by asking them to execute the algorithm on a provided sample input. Second, the tool can serve for exploration as part of problem-solving.

Designed as a minimal language, Algot's expressiveness is limited to manipulating arrays and trees. To overcome this, our new version builds on our original principles and insights to build a computationally more powerful platform.

3.2 Learnable Programming

Bret Victor, software engineering innovator and interaction designer, argued in an influential 2011 talk that creators should have an "immediate connection" with their creations [31]. To illustrate his claim, Victor showed two design tools with a graphics/code split screen and compared the difficulty of implementing new functionality by writing code, for example, by updating a variable in a platform game to change a character's speed, and using direct manipulation on the graphical component to achieve the same effect. Victor demonstrated not only how the latter option was easier and more accurate, but also how it could facilitate exploration and creativity. While the immediacy of the change is important in itself — development environments with a long edit-compile-run cycle, even to the tune of just several seconds, have a negative impact on some developers [36] — it also relieves the programmer of continuous task switching between code and effect.

3.3 Block-Based Programming

Block-based programming languages such as *Scratch* have been instrumental in making programming more accessible. *Scratch* was originally aimed at a younger audience, featuring illustrations and sounds [19] that appeal to children and adolescents. Since its inception, *Scratch* has reached millions [34] and has become a prime example of block-based programming.

Block-based languages like *Scratch* are not only visually appealing but also help beginners by making it harder to make syntax errors. Due to the way instructions are connected by drag-and-drop, constructing a syntactically invalid statement is not possible [35]. However, block-based programming languages are similar to textual programming in that they still require users to mentally convert verbal instructions to state updates. While Scratch takes some steps towards experimentation (e.g., individual blocks can be run by clicking) and making state visible (e.g., variables can be displayed individually), the syntax — the way users program — is still only connected to actual state changes through natural-language-like instructions. For example, the term *variable* requires prior knowledge to understand and is not intuitively clear to children, or those with no background in computer science.

3.4 Programming-by-Demonstration

Programming-by-example (PBE) is a paradigm that allows the programmer to specify their intention via input-output examples [10, 22]. The examples are then used to generate precise instructions via program synthesis. One advantage with this approach is that it is congruent with human communication; we may find it easier to explain complex concepts by using examples instead of definitions or procedures.

Programming-by-demonstration (PBD) is a closely related concept. Many authors consider it synonymous with PBE [17], but it may be distinguished from PBE in that the programmer is more involved; instead of only offering input-output examples, they will demonstrate in some sense how an algorithm is intended to work on one or more input examples. PBD has a long history in robotics where it can reduce the amount of tedious programming [3].

In combination with visual programming, PBD has been used in CS education to teach programming to novices. Smith et al. developed educational software called *Creator* that relies on visual before-after rules and PBD in an effort to "bring the system closer to the user" as opposed to "bringing

the user closer to the system" [28]. Similarly, the system *Melba* [9] uses PBE and visual programming to help novices learn computer programming.

4 Approach

We set out to develop a system that makes programming less difficult and more approachable by bridging the gap between the representation of the program and the representation of the program state while trying to streamline both. The aim of this section is to motivate *why* we arrived at certain design decisions starting from three key insights; the next section will describe the different components and *how* they fit together in detail.

4.1 Key Insights

State Should Be Relatable, Visible and Editable. Many existing programming languages feature complicated state models such as object-orientation with dynamic method binding. Such features can lead to a wide range of issues such as the Fragile Baseclass Problem [21], aliasing, which can make programs difficult to understand and maintain [23], or limitations of structural coverage analysis, which poses a concern in safety-critical systems for aviation [6]. Conventional imperative languages such as C, Java, and C#, also separate state into local variables on the stack and objects living on the heap. This separation can be confusing and lead to misunderstandings. For example, the question 'Is Java "pass-by-reference" or "pass-by-value"?' [1] is one of the highest scoring questions of all times on Stack Overflow with more than 2.4 million views. Even systems designed for beginners like Scratch tend to scatter state by maintaining variables, position for individual objects, or drawings on screen. We believe that a scattered state leads to additional cognitive overhead and instead intend to maintain a single central state. Algot avoids variables as separate entities from other state.

Moreover, the program state is not visible in most existing programming environments. Tools such as debuggers or loggers allow the program state to be inspected, but these tools are often an afterthought [32] and tend to pose an additional hurdle for beginners. Finally, being able to visually manipulate the system state allows for easy experimentation and testing.

Based on these considerations, we choose to model the state as a single graph, where each node stores a numerical value. Graphs possess three characteristics that fit our needs: First, graphs have a common, agreed upon visual representation. Second, graphs can be easily explained both formally or by their visualization with boxes and arrows. Graphs can even be built as a physical model using readily available materials such as rope and paper balls. Third, graphs are among the key concepts in computer science due to their modelling power. For example, data structures, relations or programs can be expressed in terms of graphs.

In our implementation, the main view of the system, which we call the *state view*, conveniently displays the state graph using intelligent automatic layouting and allows the user to apply manipulating operations. The results of applying operations are immediately visible.

Programming Should Resemble What Happens When the Program Runs. Henry Lieberman [17] noted the gap between the programming language and the operations expressed by it when introducing the term programming-by-demonstration:

> There were these things called "programming languages" that didn't have much to do with what you were actually working on. You had to write out all the instructions for the program in advance, without being able to see what any of them did. [17]

In the spirit of Lieberman's programming-by-demonstration concept, we designed an environment where the computer is told what to do by showing it the actions it should take. In Algot, demonstration takes place in an environment that is as similar to the state view as possible.

When demonstrating new operations in Algot, users work with *abstract* nodes that represent a set of *concrete* nodes of the state graph at run-time.[2] Abstract nodes are bound to a fixed set of concrete nodes for a given execution of an operation. Yet, users work with abstract nodes the same way they work with concrete nodes in the state view. Abstract nodes can be selected during demonstration like concrete nodes and the same operations are offered. Moreover, we try to visualize the graph as it changes when applying operations. For example, we try to accurately show new nodes that the user added to the graph.

Lieberman also notes:

> I guessed that youâĂŹd have to learn some special instructions that would tell it what would change from example to example and what would stay the same. [17]

We implement this vision by introducing a way of asking questions about the current program state during operation demonstration and using the answers to these questions as predicates. This effectively allows the programmer to tell what will change from example to example.

Structure Operations in Three Simple Stages with Recursion. Nesting of control flow structures in conventional textual languages is a significant source of complexity. For example, Shin and Williams [26] showed that nesting complexity can predict security vulnerabilities. Another study

[2] We will provide an in-depth explanation of abstract nodes in Section 5.

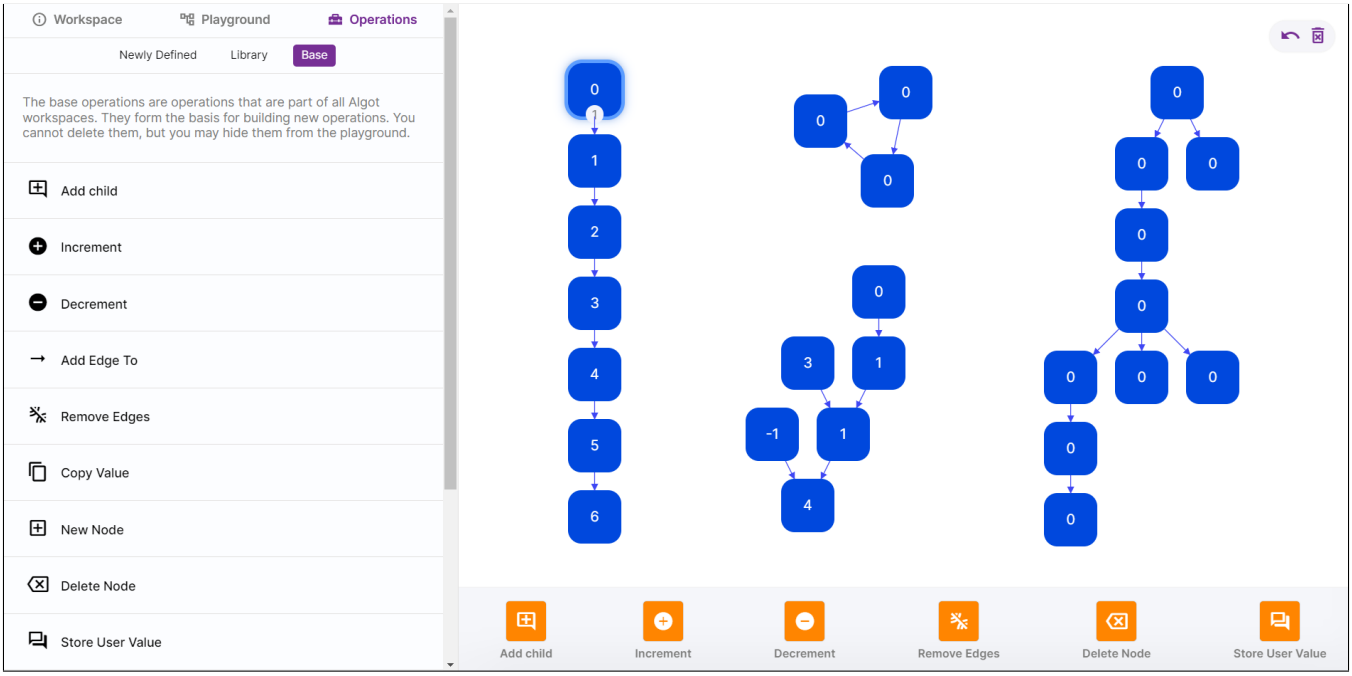

Figure 5. A screenshot of the Algot environment with the *Operations* menu selected. One node in the state graph (top left) has been selected by the programmer and six base operations can be performed on them (see the toolbar at the bottom).

by Ajami et al. [2] found that loops were harder for programmers to interpret and led to more errors than if the same program was expressed only using if statements. Furthermore, the study suggested that flat structures are slightly easier to understand.

Algot takes a radical approach that eliminates most nesting from programs except for predication and operation applications, while relying on recursion for repetitive tasks.

In education, past research [13] has indicated that students who were taught functional programming first had the same abilities as their peers who were taught programming using an imperative language. Moreover, Chakravarty and Keller [5] argue that programming should be taught language agnostically and they propose functional languages because they "help us to replace the tyranny of syntax by a principled approach that (1) conveys elementary techniques of programming, (2) introduces essential concepts of computing, and (3) fosters the development of analytic thinking and problem solving skills" [5].

Although quantitative research on the benefits of functional programming approaches in software engineering is limited [14], it has been suggested [11] that functional programming can be beneficial in constructing correct programs when combined with formal specification.

Finally, to make our programming-by-demonstration paradigm feasible, we impose a specific structure to reduce complexity on operations defined by the user. We take inspiration from the "sense-think-act" [27] paradigm used in robotics to describe the operation of a robot. A robot will first consider its sensor inputs, process the inputs to answer questions according to its purpose and then finally take action based on the insights it has gathered. We implement this via *input*, *query*, and *action stage* (see next section). This rigid structure of operations keeps the user interface manageable and easy to understand.

4.2 Implementation

We implement Algot using web technologies, only requiring a modern web browser to experiment with operations and create programs. This ensures that Algot will be widely accessible and also allows for easy sharing of programs with other people using share links. We rely on state-of-the-art frameworks such as React, Next.js and Redux for our implementation.

5 System Overview

The program state in Algot is a single directed graph called the *state graph*, which is always visible to the programmer. The state graph typically consists of many disconnected components, with each component representing a separate data structure. By applying *operations* to nodes in the graph, the programmer modifies the state graph with an immediate visible effect. Operations consist of either changing the graph structure by adding or removing nodes or edges, or by changing the values of the nodes, or any combination thereof.

To perform an operation that requires input, the programmer first sequentially selects the nodes that should serve as arguments. Algot displays small numbers on the selected nodes that reflect the order in which they were selected. Operations are displayed contextually to the user, meaning that operations with an arity that matches the number of selected input nodes are available. Some operations require no input, for example, when creating a new singleton component; such operations are available when no selection is made.

Our user interface implementation can be seen in Fig. 5. The state graph is displayed on the right (blue nodes) and the available operations are displayed in the bottom *toolbar* (orange icons). Connected components in the state graph are laid out automatically. The system is intended to be unobtrusive to decrease the programmer's cognitive load, but to make the environment more usable, it comes equipped with an undo and a reset button (top right).

5.1 User-Defined Operations

The environment is equipped by default with base operations (see Table 1). Base operations can be used to generate any finite graph, within the limitations of memory. When Algot is used as a programming system, the base operations are used as building blocks to create more complex and powerful operations.

To program a new operation, the user enters the *demonstration view* (see Fig. 6). In contrast to the state view, the demonstration view contains abstract nodes, which have a name and represent a set of concrete nodes. These names serve purely as comments to help programmers keep track of nodes. Input abstract nodes (blue) have unknown values and relations to other nodes, while non-input abstract nodes (orange) are new nodes created by the programmer during the operation definition.

Operations are programmed in three stages:

1. *Input stage*: The input nodes of the operation are specified. Pattern matching allows matching subgraphs starting from input nodes.
2. *Query stage*: The operation queries the current state of the state graph.
3. *Action stage*: The operation manipulates the state graph based on the nodes from the input stage and the query results from the query stage.

We will now provide more detail on each stage.

5.1.1 Input Stage. In the *input stage* (Fig. 7), the programmer specifies how many input nodes the operation takes. For each input the programmer creates, they can build a directed, acyclic connected component such that the corresponding input node is some node in the component. The other nodes in the subgraph are new *pattern matching abstract nodes* (pink). This means that when the operation is executed, Algot will try to match the concrete nodes with the abstract nodes of the pattern matching component. To do this, the subgraph is traversed in both directions starting from the input node.

The matching algorithm keeps track of which nodes have been matched previously when matching new nodes and uses these nodes to anchor further matches. For example, when an input node a is a child of some parent node and has left and right siblings in the pattern graph, pattern matching will match the siblings relative to a. This means that if the pattern graph only shows one left sibling for a but the concrete graph features several left siblings, the right-most left sibling will be matched. Matching for right siblings happens analogously from left to right.

5.1.2 Query Stage. In the *query stage* (Fig. 8), the programmer will run *queries* instead of operations on the input nodes. This means that the programmer can ask closed questions about the nodes gathered in the input stage, whose answers can be used to predicate operation applications during the action stage. All queries that are currently available in Algot can be seen in Table 1.

When the user runs a query, a *query result panel* is added to the demonstration view. Fig. 6 shows the query result panel of applying the *Is Zero?* query on the input node *idx*.

It is important to note that when the operation is executed, the query stage runs strictly before the action stage. That is, all queries are performed in the initial, unmodified state graph and the results stay the same for the entire execution of the operation. As a consequence, modifications to the state graph are not reflected in query results.

5.1.3 Action Stage. The *action stage* (Fig. 9) is similar to the state view. It allows the user to select abstract input nodes and apply operations as within the state view.

Because some operations, such as the *New Node* operation, for example, create new nodes that the user might want to use as part of the operation, we introduce the notion of operation outputs. Table 1 denotes the base operations that have an output value.

When a user applies an operation that produces an output, the output will be displayed in the state view, mimicking the concrete state change when the operation is applied. For example, in Figure 6 the user has added a child node to abstract node c. The new child node d is visualized as a child of c just like in the state view.

Despite this, the graph displayed in the action stage is only an approximation. For instance, applying further operations to c might actually delete the child d again without the graph updating. Correctly visualizing the graph in the action stage is an undecidable problem in general. We choose to over-approximate by conservatively keeping nodes and edges that might be deleted. For any point in the operation definition and any concrete execution at this point, the set of nodes and set of edges displayed in the action stage are supersets of the set of nodes and the set of edges of the concrete state graph, respectively. The abstract graph is thus a

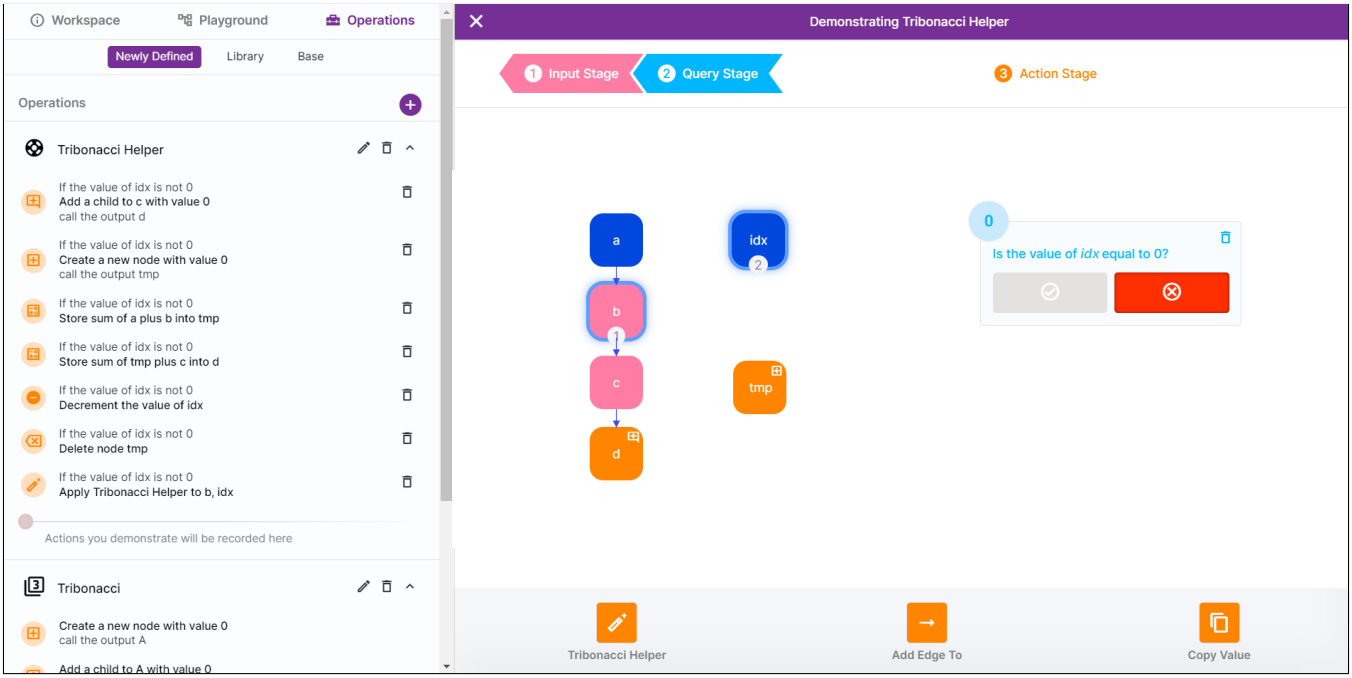

Figure 6. A user defines a *Tribonacci* function in Algot using the demonstration view. Tribonacci is similar to the Fibonacci sequence, but takes the sum of the previous three elements. The programmer is in the process of applying the *Tribonacci Helper* operation recursively to the nodes labelled *b* and *idx*.

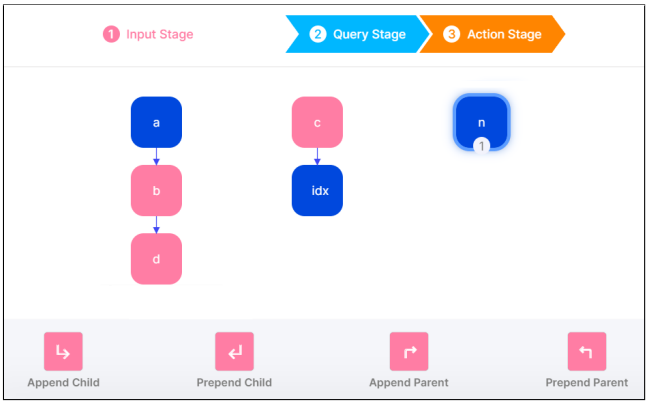

Figure 7. A screenshot of the *input stage*. Here, the programmer can create connected graphs that include the input nodes (blue). In this example, the node *d* is the grandchild of the input node *a*, which can later be used as operation inputs in the *action stage*.

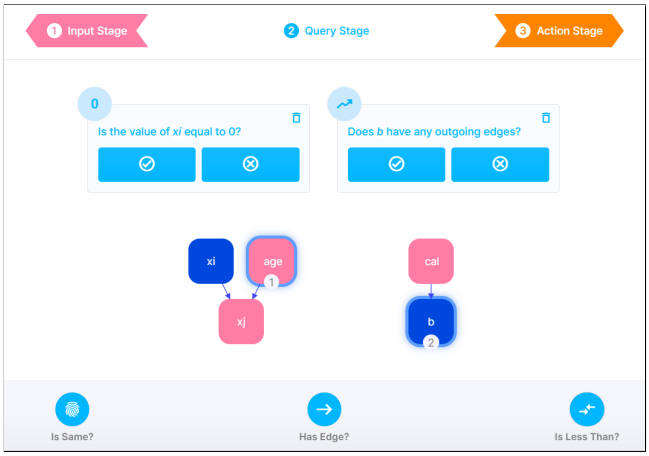

Figure 8. A screenshot of the *query stage*. Two queries have been made (upper middle) and the user has the option to add another one (bottom) based on the two selected elements (*age* and *b*).

sound approximation and displays all nodes the programmer might want to interact with. However, this also implies that the abstract graph might display nodes or edges that do not exist in some concrete executions of the operation. If the programmer interacts with such a node and the node does not exist in a concrete execution, no action will be taken at run-time, which is useful in many cases. On the other hand, this can also hide some programming errors, when the programmer mistakenly assumes that a node that is displayed in the abstract graph will always exist.

To apply query results as predicates, programmers use the checkmark to indicate that the next step should be taken only if the answer to the query is positive, and the cross button if the next step should be taken only in the negative case.

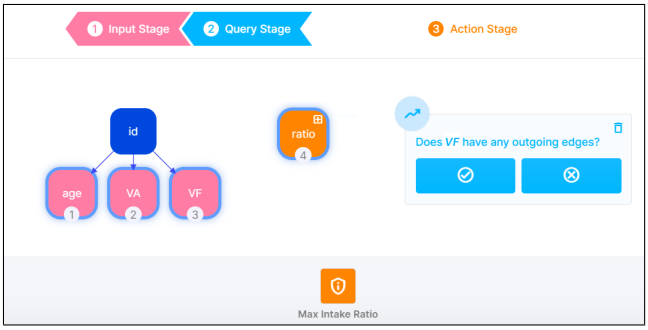

Figure 9. A screenshot of the *action stage*. Note its resemblance to the state view. Four nodes have been selected (*age*, *VA*, *VF*, *ratio*) and a user-defined operation *Max Intake Ratio* that takes four inputs is displayed contextually.

Multiple predicates can be applied at the same time. To do so, the programmer selects multiple predication buttons before demonstrating the operation application. This step will then be taken only if *all* of the predicates hold at run-time.

All applied operations are listed as steps in the left menu (Fig. 6) and users can delete operation applications from the list.

Finally, let us highlight the support for recursion in Algot. Since our paradigm does not feature any explicit looping constructs, any form of repeated execution is achieved via recursion. Recursion is straightforward because the operation currently being demonstrated is also available in the toolbar. Thus, the user simply selects appropriate input nodes for the recursive application and clicks the operation in the toolbar. We will give more examples of recursion in the next section.

5.2 Tutorial

Algot also includes a built-in tutorial that teaches new users the basics of the system. The tutorial (see Fig. 10) is step-by-step and integrates with the system to detect if the user has successfully completed the current task. When the task is completed, the user can proceed to the next step. The tutorial covers the basics of the state view, such as creating nodes, selecting nodes and applying operations to them, and of creating operations, by introducing the input, query and action stage.

6 Examples

To illustrate the full power of the system, we describe some example operations. The first example shows the basics of programming-by-demonstration in Algot. Each example will make use of additional concepts.

6.1 Simple Example — Add Child with +1

In this simple example, we want to combine two base operations to a single operation. Algot supports the *Increment* and the *Add Child* operation as can be seen in Table 1. The goal

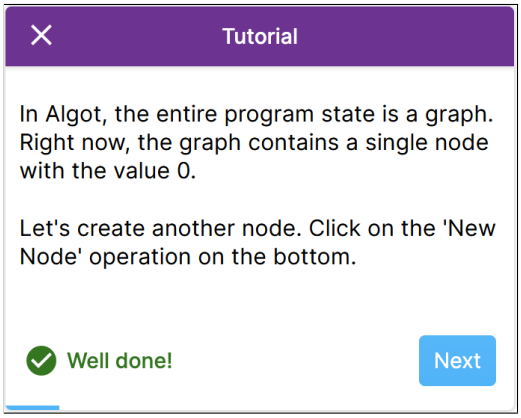

Figure 10. A screenshot of the tutorial window, which the user can freely move inside the Algot user interface.

Figure 11. We demonstrate the steps taken in the action stage by the operation from 6.1 by first incrementing the abstract node *a* and then adding a child to it.

is to define a new operation *Increment and Add Child* that performs both operations at once.

We create a new operation, give the operation the right name, select a matching icon, and then bring up the demonstration view. In the input stage, we add a new input to the operation by clicking the "Add New Input" button. A new input abstract node appears, which represents the only input to our new operation. Because the new *Increment and Add Child* operation performs all its steps unconditionally, we do not need to run any queries and move on directly to the action stage.

Next, in the action stage, we simply select the input abstract node and click *Increment*. Finally, we select the input abstract node again and click *Add Child*. This is illustrated in Fig. 11. These steps are identical to the steps a user would perform to increment a node and add a child in the state view.

Table 1. The base operations and queries in Algot.

Operation	Input	Description	Output
Delete Node	*a*	Removes *a* from the graph	
Increment	*a*	Increases the value of *a* by 1	
Prompt	*a*	Asks the user for a value interactively and stores the value in *a*	
Decrement	*a*	Decreases the value of *a* by 1	
Add Edge	*a*, *b*	Adds an edge from *a* to *b*	
Copy Value	*a*, *b*	Sets the value of *b* to the value of *a*	
New Node		Creates a new node *a* with value 0	*a*
Add Child	*a*	Adds a new node *b* with value 0 to the graph and draws an edge from *a* to *b*	b

Operation	Input	Description
Remove Edges	*a*	Removes all edges from *a* (regardless of direction)
Sum	*a*, *b*, *c*	Takes the sum of the values of *a* and *b* and stores the result in *c*
Subtract	*a*, *b*, *c*	Takes the difference between the values of *a* and *b* and stores the result in *c*
Query	**Input**	**Description**
Is Same?	*a*, *b*	True iff *a* is the same node as *b*
Is Zero?	*a*	True iff the value of *a* is 0
Has Edge?	*a*, *b*	True iff *a* has an edge to *b*
Has Outgoing?	*a*	True iff *a* has any edges to other nodes
Is Less Than?	*a*, *b*	True iff the value of *a* is smaller than the value of *b*

6.2 A Mathematical Function — The Minimum

We will focus next on implementing a more meaningful operation: The minimum of two values. We model the function as an operation taking three inputs: Two nodes *a* and *b* of which we want to compute the minimum and a node *result* into which we will store the computed minimum. We create the new operation, model these inputs in the input stage, and continue to the query stage. We need to base our actions in the action stage on whether *a* or *b* has the smaller value. To do so, we select *a* and *b* in the query stage and click *Is Less Than?* in the toolbar. A query result panel appears that we can use to predicate our actions in the action stage.

Now in the action stage we want to copy the value of the node with the smaller value to *result*. Let us first consider the case where *a* contains the smaller value. We click the checkmark in the query result panel. This indicates to Algot that we want to execute the next action *only if* the answer to the question is yes, i.e. if *a* is less than *b*. After we clicked the checkmark, we select *a* and *result* and click *Copy Value* in the toolbar. (See Fig. 12) We then demonstrate the other case analogously by clicking the cross instead of the checkmark.

6.3 Pattern Matching — Incrementing the Right Sibling

We previously discussed how pattern matching allows us to interact with nodes that are connected to an input node. We showcase this feature by creating an operation that increments the right sibling node of the selected node in a tree. We give an example of a right sibling in Fig. 13, where we have marked the right sibling of the selected node with an 'X' for illustration purposes.

After creating an operation for this task, we add an input node in the input stage. The new input will represent the node whose right sibling we want to increment. Next, we will model the subtree that contains the sibling using pattern matching. When selecting the new input node, the toolbar displays options such as *Append Child*, *Append Parent* and others. We will first select *Append Parent* to model the parent that the input node has. Then, we select the new parent node and click *Append Child* to model the right sibling. The process of creating the pattern and the final pattern is depicted in Fig. 14.

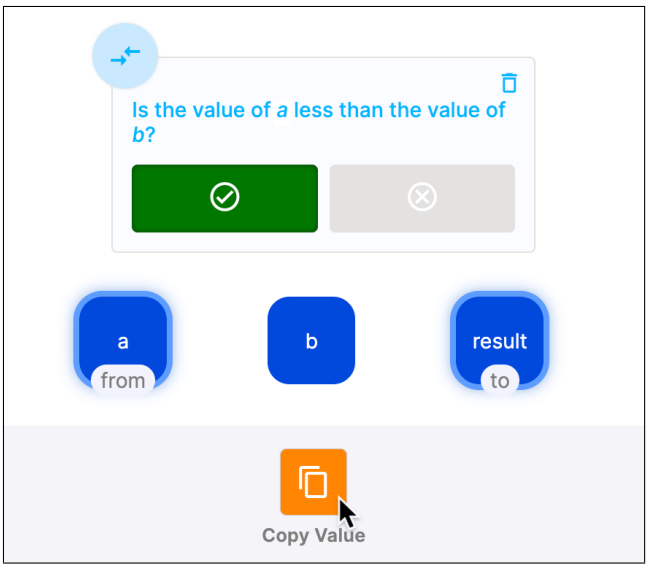

Figure 12. We demonstrate part of our *Minimum* operation from 6.2. The action is predicated and will only be executed if the value of a is less than the value of b.

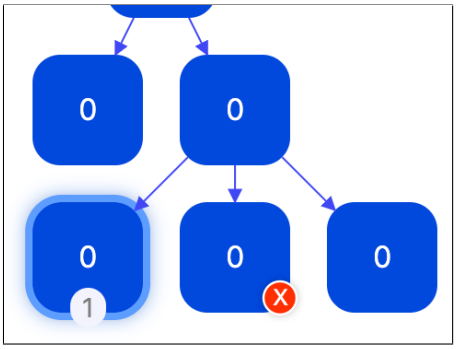

Figure 13. The right sibling (marked with an 'X' for illustration purposes) of the selected node (left).

Finally, let us move directly to the action stage, select the right sibling (called c in Fig. 14) and click *Increment Child* in the toolbar.

6.4 The Power of Recursion — Fibonacci

In this example, we show how Algot makes use of recursion by computing the first n Fibonacci numbers. Our approach works inductively by assuming that we have a linked list of already computed Fibonacci numbers (such as in Fig. 17).

With this assumption, we create a new operation *Fibonacci*. We add an input that represents the tail of the list. Due to our assumption, the tail of the list f_i corresponds to the latest Fibonacci number computed. To compute the next Fibonacci number, we also need the predecessor f_{i-1} of f_i. Because f_i is the tail of the list, we select f_i, click *Add Parent* and obtain a pattern matching node that corresponds to f_{i-1}.

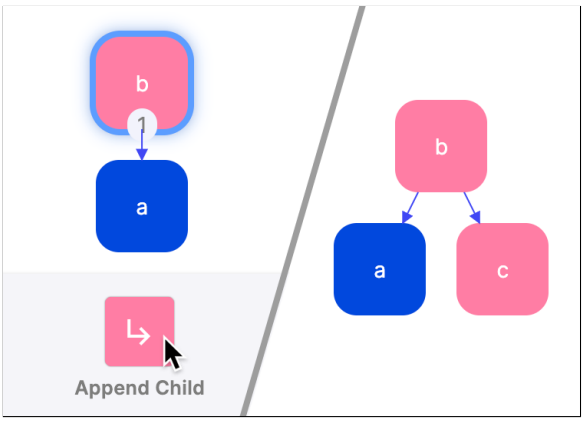

Figure 14. The process of creating the pattern and the final pattern for the operation from 6.3.

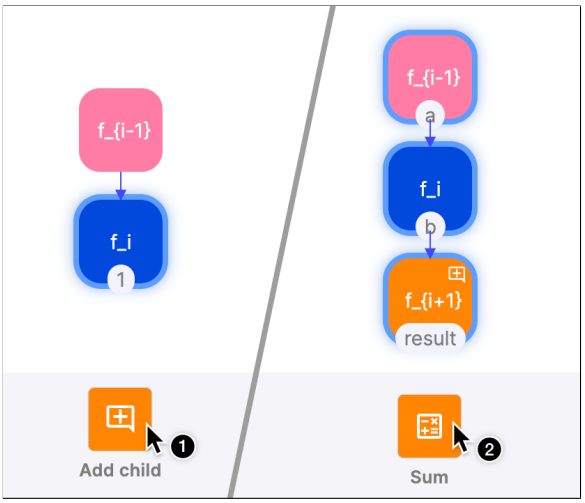

Figure 15. We first add a child for the next Fibonacci number and then calculate its value.

We proceed to the action stage and proceed as follows to calculate the next element of the list: First, select f_i and click *Add Child* to create a tail of the list representing f_{i+1}. Then, select f_{i-1}, f_i and f_{i+1} and apply the *Sum* operation to store the sum of f_{i-1} and f_i into f_{i+1}. All of this can be seen in Fig. 15.

It seems tempting to use recursion right away and apply our new operation *Fibonacci* directly to f_{i+1}. Yet, of course, this would lead to infinite recursion. Instead let us go back to the input stage and add another input *counter*, which indicates how many more Fibonacci numbers the operation is supposed to calculate. If the value of *counter* is zero, the operation does not do anything. To this end, we need to query whether *counter* is zero in the query stage.

Then, we delete our previous demonstration and redemonstrate the steps our operations should take in the action stage:

1. When the *counter* is not 0, add a child f_{i+1} to f_i.

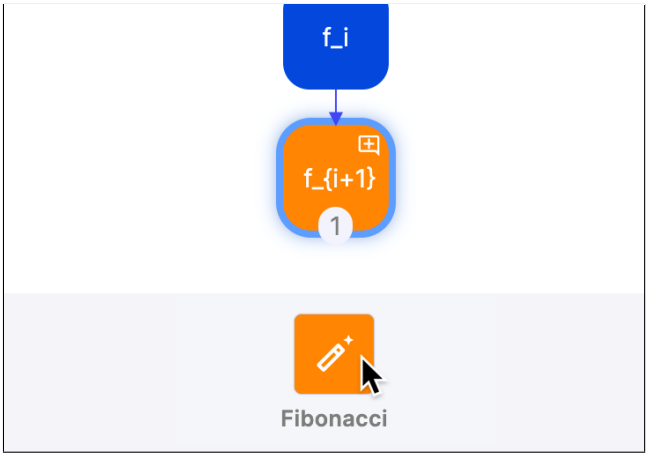

Figure 16. The programmer uses the Fibonacci operation recursively.

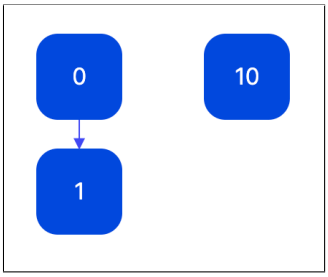

Figure 17. On the left is a linked list with the first two Fibonacci numbers 0 and 1. On the right is the counter of how many Fibonacci numbers to compute with value 10.

2. When the *counter* is not 0, store the sum of f_{i-1} to f_i into f_{i+1}.
3. When the *counter* is not 0, decrement *counter*.
4. Apply the *Fibonacci* operation to f_{i+1} and *counter*.

Back in the state view, we can test our operation by setting up a linked list with the first two Fibonacci numbers and a counter of how many additional nodes to compute. This can be seen in Fig. 17.

6.5 A Data Structure — Binary Search Tree

In this final example, we will implement an operation that works on binary trees. We assume classical binary search trees that store values only in non-leaf nodes. This means that values in leaves do not have any significance. For example, a node with no edges is therefore an empty binary search tree, because the root is also the only leaf.

Let us now define an operation *Insert* that inserts a value into a binary search tree. It will take two inputs: The *root* of the binary search tree and a node *new* containing the value to insert.

After we modeled the two inputs in the input stage, we move on to query stage. The first thing that interests us is

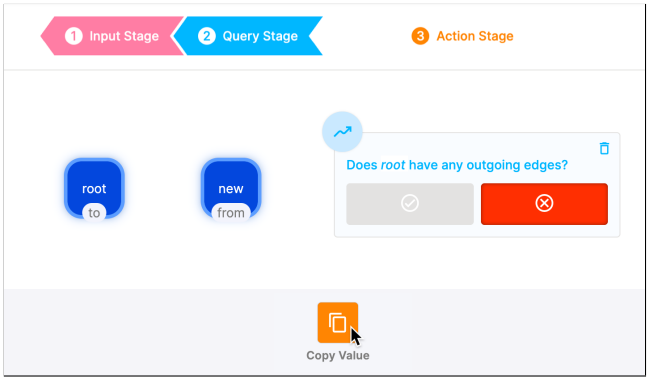

Figure 18. If the *root* does not have outgoing edges it is a leaf and we can copy the value of *new* into it. In the next step, we will add two leaves to *root*.

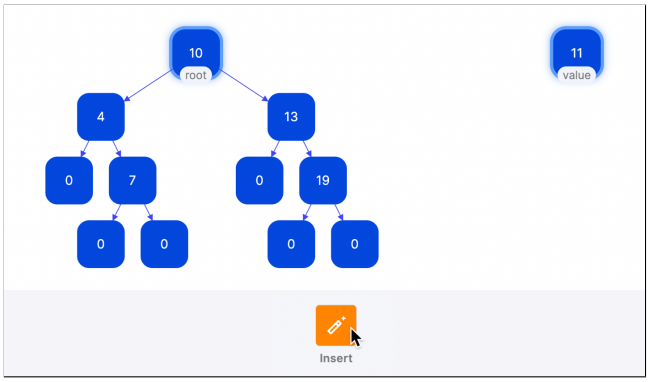

Figure 19. We test the *Insert* operation in the state view. 10, 4, 7, 13, 19 have already been inserted.

whether the root of the tree provided is also a leaf itself. To do so, we use the *Has Outgoing?* query. Clearly, a node without any outgoing edges is a leaf.

Next, in the action stage we need to consider both cases. The case where the root is a leaf is easy: We copy the value from *new* to *root* (see Fig. 18) and add two children to *root* to ensure the binary search tree invariant. The case where the root is not a leaf is more involved. We need to access both the left or right subtree of *root*, which requires pattern matching. Because pattern matching is not available at this stage, we will define a helper operation *Insert into non-leaf*.

For *Insert into non-leaf* we have the same inputs as before and model the subtree using pattern matching in the input stage. Then in the query stage, we ask whether the value of *new* is less than the value of *root*. Finally, in the action stage, we apply *Insert* to the left subtree and *new* if the value of *new* is less than the value of *root* (see Fig. 4), and otherwise apply *Insert* to the right subtree and *new*.

This concludes the implementation. An example of inserting the values 10, 4, 7, 13, 19, 11 in this order can be seen in Fig. 19.

Figure 20. A conceptual design of Algot integrated with example-based live programming. In this case, the programmer can enter the demonstration view by first selecting an example (in this case, two singleton nodes with the values 6 and 0) and then selecting *New* from the operation view to define an operation with two inputs.

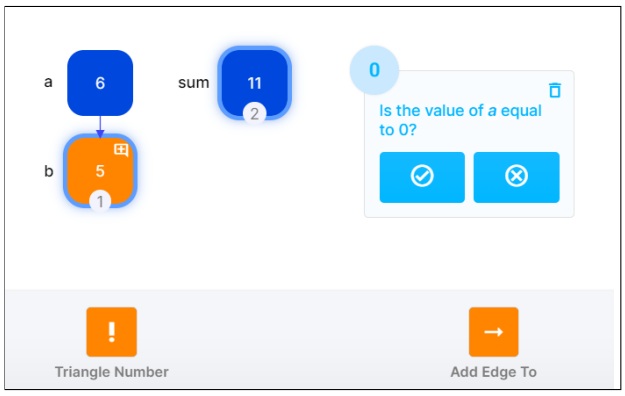

Figure 21. When attempting to define a function that calculates triangle numbers, the programmer can view how the values change during the course of the operation definition.

Let us note that the helper operation is not strictly necessary. Since pattern nodes that cannot be matched against any concrete node are represented by the empty set, we could have also used pattern matching in the input stage of the *Insert* operation. We nonetheless think that the separation into two operations makes this example easier to understand.

7 System Extensions

Algot is a new programming language whose appeal is not only based on its current implementation but also the myriad ways in which it can be expanded. In this section, we explain two possible system extensions that are being worked on but have not yet been implemented.

7.1 Example-Based Live Programming

While the current version of operation definitions in Algot uses programming-by-demonstration, it still requires manipulating nodes without seeing explicit numerical values. This means that when functions involve a large number of mathematical operations, the programmer needs to keep track of many changes without any immediate visual feedback. A possible response to this is *example-based live programming*, which is intended to help programmers by defining and using live examples as a part of the program and thereby "explore the actual behavior of their code during development" [25].

Algot is well-suited towards supporting example-based live programming without any changes to its underlying computational model. To define a new operation, programmers may select an example from the state view (see Fig. 20) that matches the expected arity of the operation.

In the demonstration view, the abstract input nodes will have a name and a concrete value that initially match the values of the example input. As the user applies operations on the abstract nodes, their displayed values change. Fig. 21 shows how programmers can use the value changes of the input nodes in Fig. 20 to help define the *triangle numbers* T_n. After inserting a child node to the first input node, copying the value of its parent (6) and decrementing it by one, it displays the value 5. After the next operation, the second input node will subsequently display the sum of 6 and 5. The programmer proceeds normally by defining a query for the base case and by calling the *Triangle Number* operation recursively on the nodes *b* and *sum*. Note that the underlying model is unchanged, but by displaying example values, it is easier for the programmer to keep track of value changes.

As without the extension, programmers can add queries in the query stage and apply predicates in the action stage independently of the example values. However, if the user applies a predicate that contradicts the original input example, the user is asked to provide additional suitable example values that satisfy the predicates.

7.2 Typed Algot

While the modeling powers of Algot are technically unlimited, it lacks modularity to build larger systems from fundamentals like strings and trees more conveniently. To effectively achieve this, a type system, which will make Algot more suitable for constructing complex applications, can be introduced into Algot in two steps.

First, we introduce two *primitive types* for nodal values. The *numeric* type has been presented as the default in this paper. Numerical values are unbounded, and arithmetic is exact, as proposed for languages such as ABC [24]. The other type is a Unicode code-point, similar to the Haskell `Data.Char` type [20], which we leverage to represent strings and refer to as *char* type.

A string can now be conveniently represented as a linked list of *char* nodes. The user interface will display all code

points using a conventional, suitable representation. An example can be seen in Fig. 22.

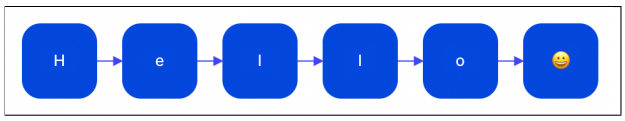

Figure 22. A string with value "Hello" followed by an emoji

In the second step, we introduce composite types as a way to group and designate one or more related connected components to be of a specific type. This makes working with them more convenient and better expresses the programmer's intent. Users can now select composite types as a single unit in the state view and during operation demonstration.

For example, we might introduce a composite type *String* to convey the meaning of a linked list of chars. Moreover, we could model a parcel as a combination of a *String* representing its destination and a numerical node containing its weight. In Fig. 23, the user has selected a string and is about to apply an operation *Uppercase* to the string. Furthermore, an instance of the *Parcel* type can be seen.

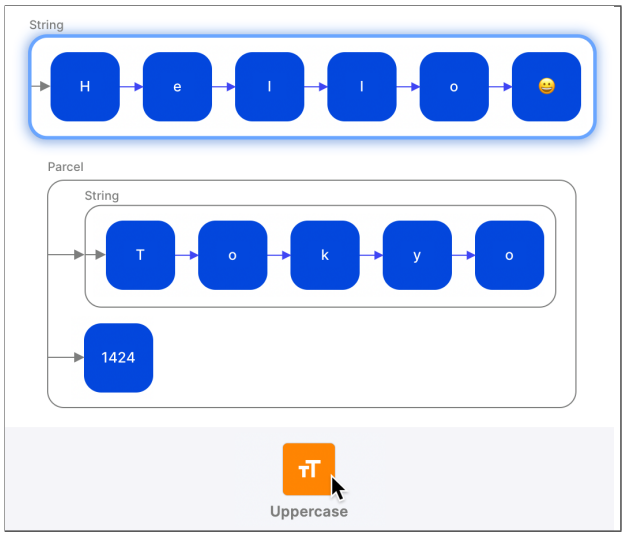

Figure 23. The user has selected a value of composite type *String* and is about to apply the *Uppercase* operation to the string.

When modelling inputs in the input stage now, programmers specify the type of the input. An input can either be a numeric node, a char node or a composite type. When the user selects a composite type, input abstract nodes appear for the composite members, and users are free to use pattern matching within the composite type instance. The toolbar will take into account types and only display suitable operations.

Typing Algot will also be crucial in building more complex applications, allowing for abstraction and encapsulation. Displaying all data structures in the graph layout we presented so far would prove impractical for larger applications. We can improve this by allowing users to define custom, more specific representations for each type of component, similar to how Subtext [8] allows customizing the presentation of the program. For example, it might be better to display strings more compactly than the general linked-list representation the state view shows by default. Moreover, we believe that enabling this kind of customization of the state view can naturally lead to a new way of defining graphical user interfaces as a native representation of state. Finally, we are working on a *library* feature (see top left in Fig. 6) that will allow effortless sharing of operations between projects and with other people; types will be fundamental in ensuring correct use of library code.

8 Discussion and Open Challenges

Our goal with this paper was to show how the programming language Algot with its environment can bridge the syntax-semantics gap of programming. While Algot is in its infancy, we hope that our examples have convinced the reader that this novel approach is a solution to the syntax barrier that can significantly improve how we create programs.

As our language becomes more mature, we hope that it will prove to be useful in educating computer science students. Section 3.1 summarized some of Bret Victor's design principles for programming languages that support learning. We have designed Algot with these principles in mind, and more generally believe that it encourages what Victor calls "powerful way of thinking," in our case via an expressive and intuitive demonstration system.

Algot has limitations that leave room for improvement. First, Algot is currently better suited for program composition than program comprehension; in our work, we have not presented a way to make existing programs easier to understand than those written using conventional textual languages. When the programmer wishes to trace user-defined operations, they can view a textual representation (see the left menu in Fig. 6) which resembles pseudo-code. To better align code comprehension in Algot with our research vision, we are exploring additional options, such as playing a demonstration of the program back to the user, or to introduce an abridged version of our input-query-action panels in the *Newly Defined Operations* view (Fig. 6). Implementing this and studying how this affects the usability of the system remains an avenue for future research.

We believe that Algot has the potential to make computer programming more approachable. Nonetheless, programming remains an iterative and error-prone activity for the foreseeable future. Therefore, Algot needs to enable the user to edit operations with ease, both to expand or redefine their

purpose and correct mistakes. During the development process, we have experimented with several ways to support this, such as allowing the programmer to re-enter the demonstration view of a given operation and replace incorrect queries or operations.

Our short-term vision is that the user can take the same approach as when working in other languages; when the user identifies a mistake in an existing program, they can rectify the issue by selectively redemonstrating individual steps or the entire operation. In the long-term, we aim to introduce debugging tools that closely integrate with operation demonstration. When a user, for example, walks through the actions of an operation one-by-one using the debugger and realizes an error, they can enter the demonstration view right from the debugger to fix the problem. The demonstration view could then take over values of the program execution as examples, as we described in 7.1.

Some early feedback gathered by providing people unfamiliar with the system access to Algot seems promising and might indicate that it has a relatively shallow learning curve. We instructed volunteers to follow the tutorial described in Section 5.2 and freely experiment with the system. Among our testers was an 11-year-old with some prior exposure to programming. After completing the tutorial, she continued playing with Algot for several hours and implemented additional operations. She considered the system intuitive and visually appealing, and therefore found it easy to use and highly motivating. More rigorous, empirical studies of our system, for example, a memory load reduction evaluation, are pending; the results will help us better understand how the system can be used for learning.

Additionally, Algot may turn out to be particularly useful for users who are not proficient in using textual languages. In some ways, Algot resembles *no-code* and *low-code* development platforms that are intended for programming business applications. Many such platforms are limited in their focus on stitching pre-defined components via drag-and-drop operations. After implementing the features discussed in the previous section and adding better support for input-output, the expressiveness and hypothesized intuitiveness of the Algot environment can make it appealing as an application programming language for users with a less technical background.

9 Conclusion

We introduced a new version of Algot, a new programming-by-demonstration system that allows users to easily define complex operations using recursion. Algot includes a state view for easy testing and experimentation, intended to help users get familiar with the mechanics of state changes. The user interface for demonstrating new operations strongly resembles the state view and always presents an approximation of the state resulting from performing the demonstrated operations. Our aim with this is to make the process of programming as similar as possible to manually modifying the state, thereby eliminating cognitive overhead as often introduced by conventional syntax. We believe that Algot provides a new direction in researching more approachable programming paradigms with applications both in industry and education.

Acknowledgments

We thank Tracy Ewen, Karl-Heinz Weidmann, and the anonymous Onward! reviewers for valuable feedback on earlier versions of this paper and Katrin Steigenberger for inputs on the visual design of Algot.

References

[1] Accessed: 2022-07-10. Is Java "pass-by-reference" or "pass-by-value"? Online: Stack Overflow. https://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value

[2] Shulamyt Ajami, Yonatan Woodbridge, and Dror G Feitelson. 2019. Syntax, predicates, idiomsâĂŤwhat really affects code complexity? *Empirical Software Engineering* 24, 1 (2019), 287–328. https://doi.org/10.1007/s10664-018-9628-3

[3] Aude Billard, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal. 2008. Survey: Robot programming by demonstration. *Handbook of robotics* 59 (2008). https://doi.org/10.1007/978-3-540-30301-5_60

[4] Nigel Bosch and Sidney DâĂŹMello. 2017. The affective experience of novice computer programmers. *International journal of artificial intelligence in education* 27, 1 (2017), 181–206. https://doi.org/10.1007/s40593-015-0069-5

[5] Manuel MT Chakravarty and Gabriele Keller. 2004. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming* 14, 1 (2004), 113–123. https://doi.org/10.1017/S0956796803004805

[6] John J Chilenski, Thomas C Timberlake, and John M Masalskis. 2002. *Issues Concerning the Structural Coverage of Object-Oriented Software.* Technical Report. Office of Aviation Research, Washington DC, USA.

[7] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education.* 208–212. https://doi.org/10.1145/1999747.1999807

[8] Jonathan Edwards. 2005. Subtext: uncovering the simplicity of programming. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* 505–518. https://doi.org/10.1145/1094811.1094851

[9] Nicolas Guibert, Patrick Girard, and Laurent Guittet. 2004. Example-based programming: a pertinent visual approach for learning to program. In *Proceedings of the working conference on Advanced visual interfaces.* 358–361. https://doi.org/10.1145/989863.989924

[10] Daniel Conrad Halbert. 1984. *Programming by example.* Ph. D. Dissertation. University of California, Berkeley.

[11] Peter Henderson. 1986. Functional programming, formal specification, and rapid prototyping. *IEEE Transactions on Software Engineering* 2 (1986), 241–250. https://doi.org/10.1109/TSE.1986.6312939

[12] Geert Heyman, Rafael Huysegems, Pascal Justen, and Tom Van Cutsem. 2021. Natural language-guided programming. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* 39–55. https://doi.org/10.1145/3486607.3486749

[13] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. 1993. Teaching functional programming to first-year students. *Journal of Functional Programming* 3, 1 (1993), 49–65. https://doi.org/10.1017/S0956796800000599

[14] Abdullah Khanfor and Ye Yang. 2017. An overview of practical Impacts of Functional Programming. In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*. IEEE, 50–54. https://doi.org/10.1109/APSECW.2017.27

[15] Maria Kordaki, Micael Miatidis, and George Kapsampelis. 2008. A computer environment for beginnersâĂŹ learning of sorting algorithms: Design and pilot evaluation. *Computers & Education* 51, 2 (2008), 708–723. https://doi.org/10.1016/j.compedu.2007.07.006

[16] Shriram Krishnamurthi and Kathi Fisler. 2019. 13 Programming Paradigms and Beyond. *The Cambridge handbook of computing education research* (2019), 377. https://doi.org/10.1017/9781108654555

[17] Henry Lieberman. 2000. Programming by example (introduction). *Commun. ACM* 43, 3 (2000), 72–74. https://doi.org/10.1145/330534.330543

[18] Raymond Lister. 2011. Computing education research programming, syntax and cognitive load. *ACM Inroads* 2, 2 (2011), 21–22. https://doi.org/10.1145/1963533.1963539

[19] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15. https://doi.org/10.1145/1868358.1868363

[20] Simon Marlow et al. 2010. Haskell 2010 language report. (2010).

[21] Leonid Mikhajlov and Emil Sekerinski. 1998. A study of the fragile base class problem. In *European Conference on Object-Oriented Programming*. Springer, 355–382. https://doi.org/10.1007/BFb0054099

[22] Brad A Myers. 1986. Visual programming, programming by example, and program visualization: a taxonomy. *ACM SIGCHI Bulletin* 17, 4 (1986), 59–66. https://doi.org/10.1145/22339.22349

[23] James Noble, Jan Vitek, Doug Lea, and Paulo Sergio Almeida. 1999. Aliasing in object oriented systems. In *European Conference on Object-Oriented Programming*. Springer, 136–163. https://doi.org/10.1007/3-540-46589-8_8

[24] Steven Pemberton. 1991. A short introduction to the ABC language. *ACM SIGPLAN Notices* 26, 2 (1991), 11–16. https://doi.org/10.1145/122179.122180

[25] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming, 2019, Vol. 3, Issue 3* (2019), Article 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[26] Yonghee Shin and Laurie Williams. 2008. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. 315–317. https://doi.org/10.1145/1414004.1414065

[27] Mel Siegel. 2003. The sense-think-act paradigm revisited. In *1st International Workshop on Robotic Sensing, 2003. ROSE'03.* IEEE, 5–pp. https://doi.org/10.1109/ROSE.2003.1218700

[28] David Canfield Smith, Allen Cypher, and Larry Tesler. 2000. Programming by example: novice programming comes of age. *Commun. ACM* 43, 3 (2000), 75–81. https://doi.org/10.1016/B978-155860688-3/50002-6

[29] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–40. https://doi.org/10.1145/2534973

[30] Sverrir Thorgeirsson and Zhendong Su. 2021. Algot: An Educational Programming Language with Human-Intuitive Visual Syntax. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576166

[31] Bret Victor. 2012. Inventing on Principle. https://www.youtube.com/watch?v=EGqwXt90ZqA.

[32] Bret Victor. 2012. Learnable Programming: designing a programming system for understanding programs. Published online: http://worrydream.com/LearnableProgramming.

[33] G Vlachogiannis, V Kekatos, M Miatides, M Kordaki, and E Houstis. 2001. A multi-representational environment for the learning of Bubble sort. In *proccedings of Panhellenic Conference with International Participation âĂŸNew Technologies in Education and in Distance Learning'*. 481–495.

[34] David Weintrop. 2019. Block-based programming in computer science education. *Commun. ACM* 62, 8 (2019), 22–25. https://doi.org/10.1145/3341221

[35] David Weintrop and Uri Wilensky. 2017. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–25. https://doi.org/10.1145/3089799

[36] Reilly Wood. 2019. The hunt for shorter edit/compile/debug cycles. Published online: https://www.reillywood.com/blog/inventing-on-principle.

[37] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22, 6 (2017), 3149–3185. https://doi.org/10.1007/s10664-017-9514-4