CoMo: a Whiteboard that Converses about Code

by

Andrew Thomas Correa Sabisch

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Author . . . .
Department of
Electrical Engineering and Computer Science
August 8, 2014

Certified by . . . .
Randall Davis
Professor
Thesis Supervisor

Accepted by . . . .
Professor Leslie A. Kolodziejski
Chair of the Committee on Graduate Students

CoMo: a Whiteboard that Converses about Code

by

Andrew Thomas Correa Sabisch

Submitted to the Department of
Electrical Engineering and Computer Science
on August 8, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Science in Computer Science

Abstract

Software engineers routinely solve problems by brainstorming at whiteboards. Among other modes, they communicate with speech and sketch. Unfortunately, the whiteboard plays the role of a passive medium. It serves only as a place to draw. But what if it could engage in the conversation, even to a limited degree? Ideally it would help guide the engineers to a solution by being an active participant in the conversation.

This thesis presents an early version of that vision: CoMo, a whiteboard that converses about code. CoMo is capable of engaging its user in a constrained mixed-initiative symmetric-multimodal conversation about a data structure manipulation. When it understands the data structure, it uses a code synthesis system to generate functioning C code. It can successfully hold a limited conversation and synthesize code for 50 manipulations on 8 data structures. This thesis further presents findings from an observational user study that helped guide the interaction with CoMo. Finally, this thesis presents the mixed-initiative code-generation framework that CoMo implements to achieve its interaction, and the insights about having limited natural conversations about data structure manipulations that were gleaned while creating the framework.

Thesis Supervisor: Randall Davis
Title: Professor

Acknowledgments

Finishing this thesis was the most difficult thing I have yet accomplished. There are many reasons for this, not least among which is that it is the first time in my life there was no clear goal; I needed to define one and bring it to fruition myself. I want to thank everyone that helped me through this, because this thesis was too great an undertaking for me to have accomplished without their love, support, understanding, play, and general ability to put up with me for the hardest 3 years. Thank you all.

Firstly I thank my adviser, Randall Davis. I have learned so much while under his wing that I am sure will help me through the rest of my life. Among his plethora gifts, I am most appreciative of my newly improved ability to manage projects and more clearly express ideas. Thank you Randy. This experience has been invaluable. I would not be the person I am today without you.

Most importantly, I thank my wife, Ashley. She calmed me down during the hard times and helped me up during the harder ones. She took care of me: feeding me, bathing me (I'm sorry that was required sometimes), and pointing out when I was being self-destructive in the most gentle way possible. Without her love and support I would not have been able to finish. Thank you, lar. You're the lar. You cray. I love you.

I thank Ali Mohammad for being an understanding ear, always ready with good advice and a calm perspective. I supported him through the difficult final stages of his thesis and, given all the help he gave back, I believe that he remembers (and feels responsible for) giving even more support back. He edited my thesis, talked me through difficult problems, reminded me of the bigger picture, and apologized profusely for not doing more. Thanks Ali. You're a great friend that I'm lucky to have been given the oportunity to become so close with.

I thank Asma Al-Rawi (or, as she's known on the streets, Em). She helped me through some of the more emotionally challenging times. Whether it was by helping me think through my issues over an IM chat, a phone call, or in person, she always managed to calm me down by pointing out that one piece of information I just couldn't see. Due to her vast experience helping doctoral candidates finish their theses (and probably through her stories of woe that nearly always trumped mine in severity), she always managed to talk me off of the metaphorical ledge. Thank you, Em.

I thank Jonathan Eastep. He always had a way of remininding me of the good life: life after school when money would not be an issue anymore and I could leave my work at work. He gave me perspective that helped immeasurably. He focused me on the goal and even bought me dinner when he visited. Really, he was an all-around awesome friend. Thank you Jonathan.

I thank Marek Doniec. He is, without question, the best worst influence I have ever had the privaledge to know. He helped me relax at the end of hard days and easy days alike, and was a readily available gym buddy. As if that weren't enough, he also helped me practice my German. Thank you Marek.

I thank all the memebers of the Multimodal Understanding Group: especially Jeremy Scott and Ying Yin. As academic siblings, we were in a uniqe position of being able to understand one another and help one another with research projects and academic life. Thank you all.

I thank the Agile Robotics group, and especially Seth Teller. Next to my adviser, Seth had the greatest influence on how I think about and perform research, and about how to manage large projects. Thank you AR, and thank you, Seth. I only wish you were here to read this.

I thank all my remaining friends and family, especially my brother, my mother, my father, and my uncle. They grounded me by reminding me of my roots, consoled me with their love, and lifted my spirits with their presence. Thank you all. I love you all.

Finally, I thank the broader CSAIL community, especially the members of TIG and the CSC. They made this place fun. Without them, I would not have wanted to come into work every day. I will never forget this time in my life, and I'll always have wonderful memories thanks to them. Thank you all.

This work was supported by DARPA grant #N66001-10-2-4089, "CANDOR: Clean-Slate System Integrity using Selective Redo" and NSF grant #IIS-1018055, "Enabling

and Exploring Natural Systems."

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software engineers routinely discuss problems at whiteboards to find solutions, engaging in a multimodal dialog, primarily speaking and drawing. Both of these modalities have their advantages and disadvantages: drawing is visual and intuitive while speaking is succinct and rich. At the whiteboard, drawing serves two purposes. It helps engineers visualize the problem and break it down into solvable sub-problems, and presents the problem in a more abstract, human-understandable form. Though drawing is a powerful tool, it has its limitations. Some things are more easily, quickly, and succinctly expressed with speech. For example, responding to some questions by simply saying “Yes” or “No” can be quicker and more natural than any other means. Together, these modalities can create a rich interaction full of potentially useful technical information.

Imagine the whiteboard actively participating in technical conversations of these kinds. It could engage in the conversation with the engineers by speaking, drawing, and asking questions, thereby helping the engineers come up with solutions more quickly. A smart whiteboard system of this kind has the potential to help engineers by guiding them through the problem-solving process, potentially avoiding common mistakes, and asking leading questions, e.g., about corner cases. Once a solution has been found, the system could help with the implementation process by generating code. Even if it can only generate simple code, a system like this has the potential to be a great help to software engineers.

This thesis presents an early version of that vision: a smart whiteboard system, called CoMo, that engages in a conversation-like interaction with a user involving speech and sketch about a data structure manipulation. Once the interaction is complete, CoMo uses a code synthesis tool to generate code implementing the discussed manipulation, then CoMo helps the user test the code by animating it on a drawn structure. This system has the potential to be used by students to better understand how code works by watching it animated on concrete, drawn structures. CoMo is a code generation tool that improves upon using state of the art code synthesis systems alone. This thesis gives example interaction with CoMo, describes an observational user study that guided the design of CoMo's interaction's structure, and details how CoMo works.

1.1 CoMo: the Code Monkey

I implemented CoMo, a smart whiteboard system able to hold a limited conversation involving speech and sketch concerning a wide range of different manipulations on a variety of different data structures. CoMo's name has two meanings. It is both an abbreviation of "Code Monkey" and the Spanish word for "how," used to ask someone to repeat what they said. (In English we use "what.") It is a "code monkey" because it demonstrates its understanding of a conversation by synthesizing C code and animating it on concrete examples, both those the user drew and those that they asked CoMo to draw for them. As far as I am aware, CoMo is the first mixed-initiative, symmetric-multimodal, *conversation-to-code* smart whiteboard system.

CoMo engages in a mixed-initiative, symmetric-multimodal interaction, that is modeled after human-human conversations[1]. The interaction is multimodal in that it involves both speech and sketch. The interaction is symmetric because both modes of input are also modes of output: CoMo both interprets speech and sketch and speaks and draws to ask questions. It is a mixed-initiative system because either CoMo

[1]Since the interaction is strongly constrained, the words "converse" and "conversation" are used sparingly in this thesis. They are used only to frame CoMo in the broader goal of participating as an engineer in the kinds of technical discussions mentioned above.

or the user can initiate an interaction. The user initiates interactions by drawing concrete examples of data structure manipulations, while CoMo initiates interactions by asking questions about concrete examples it draws.

Interactions with CoMo proceed in the five basic phases (Chapter 5 describes these in full):

1. The user initiates the interaction by:
   - asking CoMo if it knows what a specific data structure or manipulation is; or
   - demonstrating a concrete example on a drawn structure; or
   - defining the general data structure to talk about.
2. CoMo asks questions (and interprets answers) about:
   - an inductive general structure definition;
   - a structure with four, one, and zero nodes; or—if the manipulation consists of two structures: a *main* and an *auxiliary* one—the power set of comparisons between the main structure with four, one, and zero nodes and the auxiliary structure with one and zero nodes; and
   - if the manipulation is on an ordered structure that (as above) involves multiple structures (e.g., an ordered linked list insertion)—the power set of constraint comparisons between single-node structures where the main structure's node is less than and greater than the auxiliary structure's node.
3. CoMo uses SPT [28] to synthesize code by:
   (a) generating input based on examples the user drew;
   (b) selecting a control flow graph (CFG) to use;
   (c) running SPT; and
   (d) cleaning code and displaying it to the user.

4. CoMo animates the synthesized code on examples until the user is satisfied it is correct.
5. If the code is incorrect, CoMo reruns synthesis (step 3) with new corrected examples.

Using these phases, CoMo is able to have an interaction with and generate code for a user about 50 manipulations on 8 structures. These structures and manipulations are listed here:

1. Stack
   (a) push
   (b) pop
   (c) poll
2. Queue
   (a) enqueue
   (b) dequeue
   (c) peek
3. Singly Linked List
   (a) insert
   (b) delete
   (c) reverse
   (d) append
   (e) prepend
   (f) remove first
   (g) remove last
   (h) find first
   (i) find last
4. Doubly Linked List
   (a) insert
   (b) delete
   (c) reverse
   (d) append
   (e) prepend
   (f) remove first
   (g) remove last
   (h) find first
   (i) find last
5. Ordered, Singly Linked List
   (a) insert
   (b) reverse
   (c) remove minimum
   (d) remove maximum
   (e) find minimum
   (f) find maximum

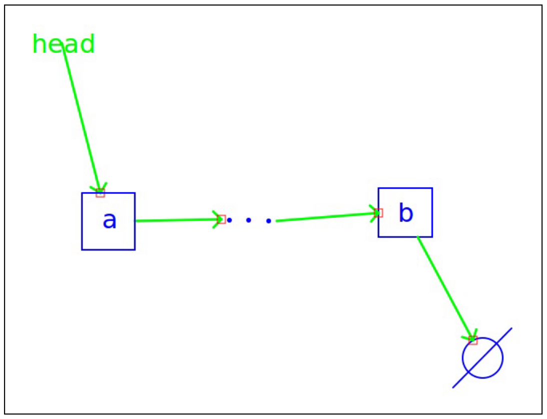
6. Ordered, Doubly Linked List:
    (a) insert
    (b) reverse
    (c) remove minimum
    (d) remove maximum
    (e) find minimum
    (f) find maximum
7. Binary Tree
    (a) left rotation
    (b) right rotation
    (c) left-right rotation
    (d) left-left rotation
    (e) right-left rotation
    (f) right-right rotation
8. Binary Search Tree
    (a) left rotation
    (b) right rotation
    (c) left-right rotation
    (d) left-left rotation
    (e) right-left rotation
    (f) right-right rotation
    (g) find minimum
    (h) find maximum

The next section demonstrates how CoMo interacts with a user by describing one example manipulation from the above list: a linked list reversal.

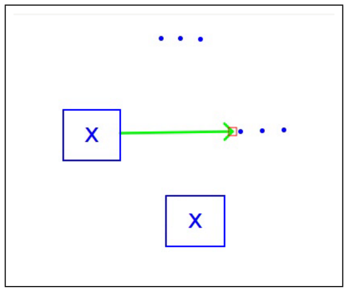
1.2 A Linked List Reversal

The user initiates an interaction with CoMo by asking if CoMo is familiar with a data structure or by drawing an example of a data structure or a manipulation. In this instance, the user chooses to ask CoMo if it knows about linked lists by saying, "Do you know what a *linked list* is?" CoMo does not, so it responds by saying, "No" and awaiting further input.

The user continues by explaining what a linked list looks like. They draw the diagram seen in Fig. 1-1a: a `head` pointer to a list starting with an `a` node and ending with a `b` node. As the user draws the diagram, CoMo recognizes their digital strokes as nodes, pointers, and labels, and replaces the strokes with cleaned-up versions of these symbols in real time. CoMo interprets the ellipsis in Fig. 1-1a to mean a repetition.



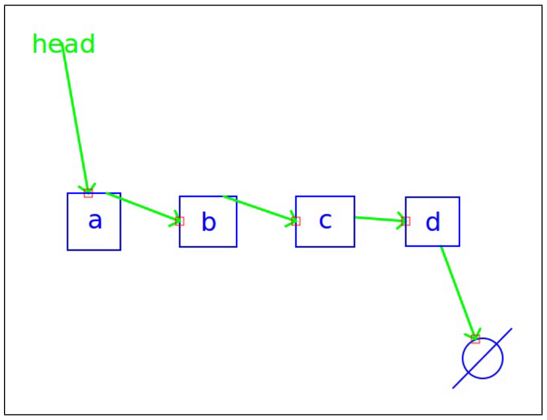
(a) A general list.



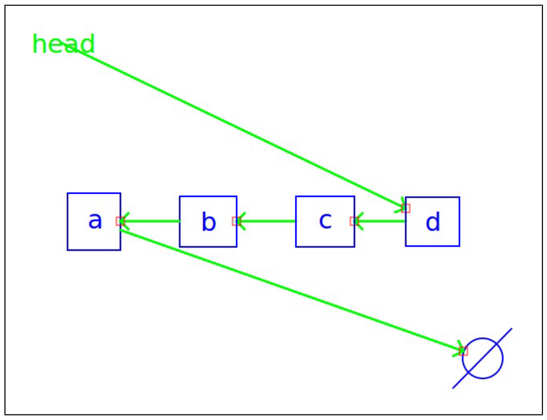
(b) What ellipses represent.

Figure 1-1: Describing a linked list to CoMo.

It does not know what in the repetition repeats, so it asks the user, "What does *this* expand to?" while displaying a new canvas with a copy of the ellipsis from the previous drawing user draws the inductive rules for expanding an ellipsis, seen in Fig. 1-1b: either replace the ellipsis with a node pointing to another ellipsis or with just a node.



(a) CoMo: "How do you handle *this* case?"



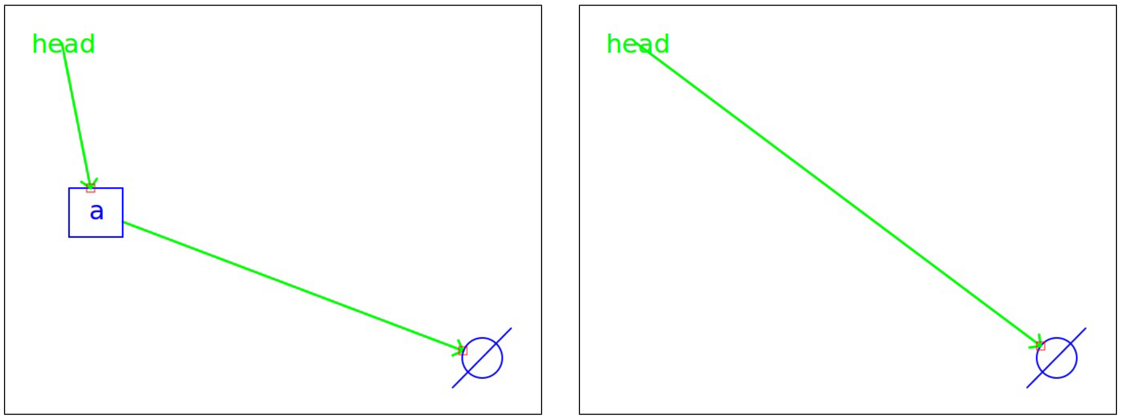
(b) User: "Like *this*."

Figure 1-2: The next question CoMo asks is about a list with four nodes.

Now that the interaction has been initiated and the user has explained what kind of structure they are talking about, CoMo begins to go through the list of cases (above) that the code synthesis system [28] requires to successfully synthesize code[2]. CoMo asks its first question about the manipulation by saying, "How do you handle

[2]The code synthesis system is described in Chapter 5.

this case?" and generating the list with four nodes seen in Fig. 1-2a. The user clicks and drags on the pointers' heads to reverse the list. Once the user is done reversing the list, they say, "Like this."



(a) CoMo: "How do you handle *this* case?" User: "Do nothing."

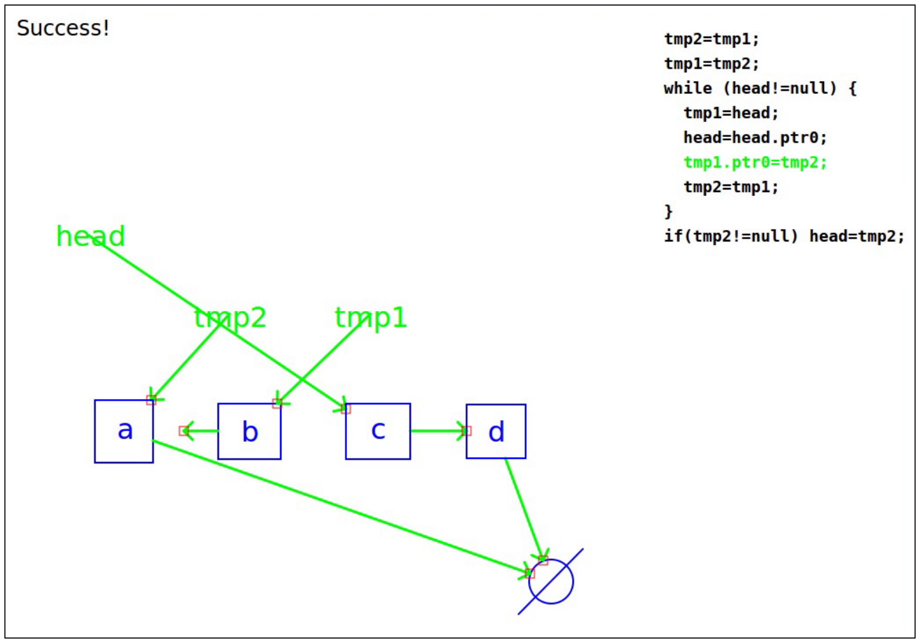
(b) CoMo: "How do you handle *this* case?" User: "Do nothing."

Figure 1-3: The next two questions CoMo asks are about the list with one node and the empty list.

Next, CoMo asks about the empty list and the list with one node. Fig. 1-3 shows the structures that CoMo generates. In response to each question, the user says, "Do nothing" because these lists look the same reversed. CoMo has finished asking all its questions, and so it says, "I think I understand." The purpose of the interaction was to explain to CoMo what a linked list reversal manipulation looks like, so that it could synthesize code. To initiate the code synthesis process, the user says, "Generate code." CoMo responds with, "I will try" and code synthesis commences.

To synthesize code, CoMo generates a description of the interaction that can be used by the Storyboard Programming Tool [28] to synthesize C code. Once code has been synthesized, it is displayed next to all of the examples in the interaction (see Fig. 1-4). The user verifies that the code is correct by having CoMo symbolically execute the code on an example. The user switches to the initial list with four nodes[3] and says, "Animate the code." CoMo resets the list to its initial state and animates the code's execution. As part of the animation, CoMo generates the two temporary

[3]As described in Chapter 4, the user can switch between existing examples by clicking on tabs along the top of CoMo's canvas.



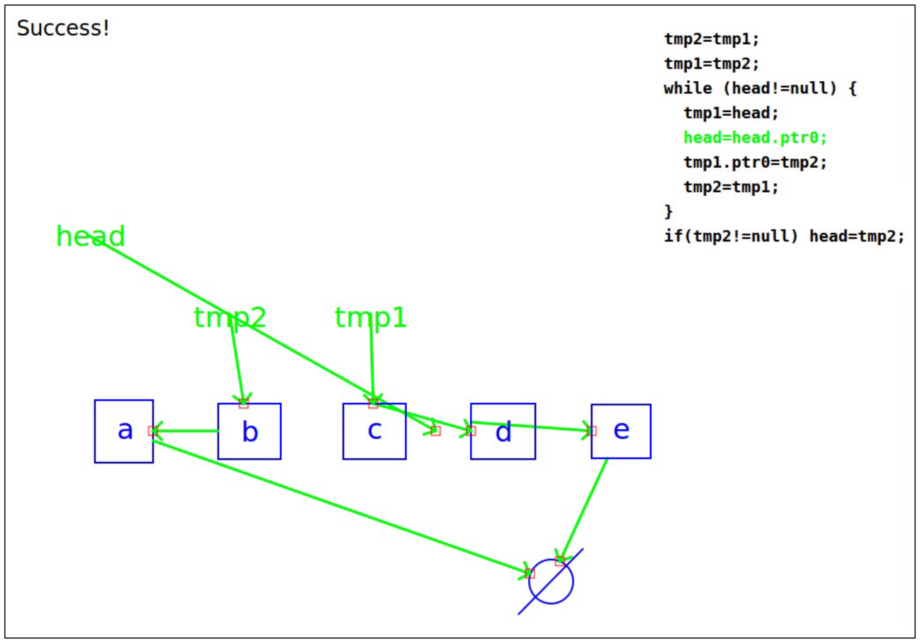
```
tmp2=tmp1;
tmp1=tmp2;
while (head!=null) {
  tmp1=head;
  head=head.ptr0;
  tmp1.ptr0=tmp2;
  tmp2=tmp1;
}
if(tmp2!=null) head=tmp2;
```

Figure 1-4: CoMo animates the code's execution on a structure. The line "`tmp1.ptr0=tmp2`" (highlighted in green) has almost completed its animation; the pointer attached to `b` has almost completed its movement from the `c` node to the `a` node.

pointers that are not present in the original diagram (`tmp1` and `tmp2`; see Fig. 1-2) and highlights the line of code it is executing in green while animating it. Fig. 1-4 shows the middle of the process, as CoMo executes the line "`tmp1.ptr0=tmp2`."

As demonstrated above, CoMo is able to generate example structures. CoMo uses the general description given at the beginning of the interaction by starting with the skeleton (Fig. 1-1a) and repeatedly replacing the ellipsis with one of the provided inductive steps (Fig. 1-1b)[4]. The user would like to test out the code on an example that CoMo has not seen yet, so they say, "Generate a structure with five nodes." CoMo creates the structure and displays it on a new canvas next to the code. "Animate the code," the user says again. Fig. 1-5 shows CoMo animating the reversal as it executes the line, "`head=head.ptr0`." The user verifies that the code works on this example, and is satisfied that the code is functionally correct.

[4]Due to the nature of the list description, CoMo will only be able to generate lists with at least three nodes. Empty structure generation and single-node structure generation are handled by copying existing elements from other examples and assigning all pointers to `null`.



```
tmp2=tmp1;
tmp1=tmp2;
while (head!=null) {
  tmp1=head;
  head=head.ptr0;
  tmp1.ptr0=tmp2;
  tmp2=tmp1;
}
if(tmp2!=null) head=tmp2;
```

Figure 1-5: CoMo animates the code's execution on a new structure. The line "`head=head.ptr0`" is in the middle of being animated; the head pointer is half way through its transition between that `c` and `d` nodes.

1.3 Code Synthesis

CoMo synthesizes code using a system called the Storyboard Programming Tool (SPT) [28]. A code synthesis system (e.g., [30, 27, 14]) attempts to synthesize code based on a textual description of a manipulation. SPT's description consists of a set of example data structures as they are configured before and after the manipulation, along with an outline of the expected code. The code outline contains simple logical constructs (e.g., `while` and `if` statements) and placeholders where synthesized code must be inserted. Systems like these model code synthesis as a search through a symbolic representation of a constrained space of programs. These systems must have two key properties: the ability to constrain the symbolic space of programs, and the ability to search in it. These systems use the code outline to define this symbolic search space, and the pointer assignments in the examples' input and output states to constrain it. More succinctly put, code synthesis is the process of solving the constraint problem defined by the code outline and the set of examples consisting of structures' input and output states. Engaging CoMo in its mixed-initiative symmetric-multimodal interaction improves upon using a text-based code synthesis system (like SPT) alone by:

1. Removing rudimentary dead code.
2. Engaging in a mixed-initiative, symmetric-multimodal interaction and converting data structure manipulation diagrams into the input and output states that the code synthesis system requires.
3. Providing feedback about what kinds of scenarios to provide, namely that the following examples suffice for SPT [28] to synthesize correct code for many simple data structure manipulations:
   (a) Examples containing structures with four, one, and zero nodes.
   (b) Examples enumerating all size comparisons between the *main* structure when it has 4, 1, and 0 nodes and any *auxiliary* structures with one or no nodes.
   (c) Examples comparing single-node structures with different values in multiple structure scenarios where the structure is ordered.
4. Removing the need to provide a code outline (called a *control flow graph*) to the code synthesis system by automatically selecting one.
5. Aiding the user in verifying synthesized code's correctness by animating it, thereby facilitating the user's own checking of the code.

1.4 Interaction

CoMo's goal is to satisfy the code synthesis system's requirements in as natural an interaction as possible. Previous work provides several insights into what kinds of interactions are natural and effective, namely:

1. *Direct-manipulation* interfaces—those where the user can directly manipulate a representation of the system's underlying model—facilitate users working with

the system by exposing them to a representation of the system's model. This makes it easier for the user to understand why the system behaves the way it does.

2. *Symmetric-multimodal* interfaces (e.g., [1])—those that employ multiple modes of interaction for both input and output—are more flexible and feel more natural than interfaces that use only a single mode of input. The modes can make up for each others' weaknesses.

3. *Mixed-initiative* systems (e.g., [22])—those that allow both the user and the system to initiate interactions—are more effective than those that require the user to follow a prescribed script.

The remainder of this section describes these points in more detail.

Direct-manipulation interfaces are interfaces that present a system's underlying model in a concrete form that can be manipulated in a direct way, usually by pointing and clicking. For example, CoMo allows users to demonstrate manipulations by updating a data structure in-place; users click and drag pointers to reassign them. CoMo demonstrates how direct-manipulation interfaces can make abstract actions (in this case reassigning pointers) more easily understandable by replacing them with physical ones (in this case grabbing and moving arrows).

Symmetric-multimodal interfaces are interfaces where all modes of input are also modes of output. Different tasks are more quickly and succinctly expressed by different modes of communication. For example, work by Adler [1] demonstrated that tasks like specifying an object's direction of motion are more easily expressed by drawing a line, while answering a yes or no question is more easily expressed with speech—by simply saying "Yes" or "No." Multimodal interfaces are able to take advantage of the strengths of the modes they employ, while interfaces using only one mode of input must deal with that mode's weakness.

Mixed-initiative interfaces are interfaces that allow both the user and the system to initiate or guide an interaction. These sorts of interfaces remain dynamic and responsive by guiding the user through a task (e.g., by keeping a list of steps to perform) while allowing them to change the interaction's focus (e.g., jumping ahead a step) at any time. CoMo's interface is mixed-initiative because either the user or CoMo can initiate an interaction, e.g., by asking a question or giving a command.

Interactions differ by domain. For example, circuit diagrams are drawn with different symbols than chemistry diagrams. To draw data structure manipulation diagrams, I needed to determine how people draw operations on data structures. To find out, I conducted a user study that paired 10 experts with 10 novices, and asked the experts to explain a linked list reversal and a binary tree left rotation to the novices. Chapter 3 discusses the user study and describes the interesting interactions observed, namely that:

1. Teachers tend to ask students what they know before starting to explain something.

2. Teachers tend to describe manipulations in terms of before and after states, but sometimes explain the process of the manipulation in detail.

3. Visual vocabularies are easily learned, understood, and used by people.

4. Teachers sometimes used an unannounced, unexplained drawn shorthand that was immediately understood by students.

5. Interactions were mixed-initiative in the sense that teachers generally initiated interactions by explaining a manipulation, but students also initiated interactions by asking questions.

6. One subject *drew* a question when verbal communication repeatedly failed.

1.5 Contributions

The contributions presented fall in three categories. The first contribution includes the findings from a user study, presented in Chapter 3. The second contribution is CoMo, the first conversation-to-code smart whiteboard system, capable of holding a mixed-initiative symmetric-multimodal interaction about 50 manipulations on 8 structures and correctly synthesizing functioning C code. The third contribution is the novel mixed-initiative code-generation framework (MICGF) that CoMo implements to engage in interactions about data structure manipulations, described in Chapter 5.

Chapter 2

Related Work

CoMo is motivated by two fields of research: programming by example (PBE; sometimes called programming by demonstration, or PBD, since the examples are demonstrated) and multimodal systems—systems that employ multiple modes of input (and sometimes output, also). As Lieberman [21] explains, work in programming by example has two dominant approaches: creating systems that have a strong background in AI and theory, and systems that rely heavily on heuristic approaches. Generally these systems' implementations start by performing exploratory user studies to determine how best to structure the user interaction and often end with evaluative user studies that determine how effective the system was at interacting with the user. This chapter discusses some exploratory user studies, the systems that were created with the insights from those studies, and finally it discusses some interesting evaluative user studies that reveal more about sketch-based interfaces.

Adler and Davis [2] and Bischel et al. [6] performed two separate studies showing that multimodal human interactions are complex, and interaction practices may vary slightly from domain to domain. Adler and Davis sought to determine what natural interactions about circuit diagram drawings look like. They explored the uses of color and gesture with a multimodal system by asking a subject to explain a mechanical system to a researcher. The study's setup allowed the researcher to guide the interaction but removed the ability to observe how a natural human-human interaction of this kind would proceed without the influence of the researcher. Since I wanted to observe natural interactions between an expert and a novice, my study removed a researcher from the dialog. Bischel et al.'s continued work sought to create a sketch recognition system capable of accurately distinguishing gestures from non-gestures. Their user study focused on mechanical diagram drawings also, however like my study, the researcher was not involved in the discussion.

Plimmer and Apperley [26] provide more motivation for CoMo. They conducted a user study that found the flexibility inherent in sketch-based interfaces leads users to do more revisions of a graphical user interface design, which results in better final designs. Cossairt et al. [11] showed a similar result in the domain of mathematics problems. Their system recognized written set equations and generated a Venn diagram representation. In a user study they performed, they found that students using the system were better able to complete a math quiz on set equations than those using pen and paper alone. This result is likely due to the fact that there is a difference between how easily understood a visual representation (like a Venn diagram) is and how much harder it is to parse a written representation (like a set equation). By parsing the set equations for students, they were able to focus on parsing and understanding the visualization and did not have to parse the set equations. This is a very similar task to CoMo's—parsing, abstracting, and visualizing a user's sketch in a way that makes it easier to understand—the two major differences being the different domain and the direction of the task CoMo performs. Instead of converting a textual representation (set equation) of a problem to a diagrammatic one (Venn diagram), it converts the diagrammatic representation (data structure manipulation diagram) to a textual one (code).

Another motivating user study is presented by Good et al. [12]. Their study found that humans are not adept at remembering specifics like corner cases when specifying the exact rules to describe an observed behavior in a video game. In fact, errors of omission were the most frequently performed kind of error the subjects performed (74% of all errors). This observation greatly motivated CoMo's questions; it is another reason that CoMo asks questions about corner case examples.

Based on Adler and Davis' user study (above), Adler built a system that served as

CoMo's greatest motivation. He called the system Midos [1] and coined the phrase "symmetric-multimodal user interface" to describe it. Midos is a system that allows users to qualitatively describe a physical system (like a Rube Goldberg machine) by specifying things like which block hit which block, and how the resulting collision affects them. It determines which question to ask by qualitatively simulating the physics of the scenario, determining what it does not know about that simulation, and asking a clarifying question. There are two major differences between Midos and CoMo: the domain and the exact method used to generate questions. The second difference results directly from the first: while Midos asks about holes in its understanding of a simulation, CoMo has no simulation to query, and so uses a predetermined list of examples shown to work with its code synthesis system.

Traynor et al. [34] report on a study about their system with very similar goals as CoMo: enabling non-experts with a system that facilitates interaction. Their system attempts to enable non-technical end users to to exercise the capabilities of a geographic information system by using Modungo et al.'s [24] programming by demonstration language. Their study found that, despite its shortcomings, their visual PBD interface is preferred over alternatives such as using SQL. Similarly, Kahn's ToonTalk [19] is a system that attempts to make expert tasks more accessible to non-experts—in this case focusing on programming. Like CoMo, this system abstracts programming concepts with real-world objects. However, instead of boxes and arrows, it uses the analogy of a town with working robots in it.

Bauer et al.'s Trainable Information Assistants (TrIAs) [5] cooperate with users to solve a problem. Like CoMo, TrIAs are agents that are specialized to help a user with a single task in a mixed-initiative interaction. Users can do things like give examples and hints to the system while the system can do things like ask for more examples and present its current hypothesis to the user. Unlike CoMo, these agents do not interact with the user multimodally with speech and sketch; instead, interaction is limited to a traditional WIMP interface [16].

Buchanan et al.'s CSTutor [8] is very similar to CoMo; differences arise from its different goal: CSTutor *teaches* students simple data structure manipulations. In contrast, CoMo is capable of discussing a data structure manipulation and gleaning enough information to initiate code synthesis with. This difference can be anthropomorphized by saying, CoMo is a student that attempts to *learn* a new data structure manipulation whereas CSTutor is (as the name suggests) a tutor, attempting to *teach* a data structure manipulation. As a result, CoMo was programmed with knowledge of what pointers and nodes are instead of, like CSTutor, with the details of its structures and manipulations.

Alvarado et al. [4] created a system called LogiSketch, that simulates drawn digital circuit diagrams. LogiSketch is similar to CoMo in that it automates a task for users, however it differs both in the method of interaction (LogiSketch does not interact multimodally) and in the domain. CoMo's interaction was, in part, motivated by two findings from the pilot study they performed: that large sketches tend to get messy and hard to understand after some time, and that subjects were engaged in the interaction. Like LogiSketch, CoMo cleans users' diagrams to help de-clutter the canvas but also generates sketches itself whenever possible, thereby further avoiding user-introduced visual clutter.

Chen et al. [9] created a system called SUMLOW that recognizes Unified Modeling Language (UML) diagrams and allows them to be exported to a third party CASE tool. They evaluated their system based on Green and Petre's "Cognitive Dimensions" [13] and found that, when compared with traditional CASE tools, SUMLOW benefited from the flexibility that a sketch-based interface allowed, but suffered from sketch recognition errors.

Kaiser [20] showed with a user study of their system that multimodal speech and sketch help to disambiguate one another, creating user interfaces that aid users in avoiding errors by using the strengths of each mode of input. Further, the system attempts to learn to better itself, which is roughly related to what CoMo attempts to do.

Mangano et al. made a system, called Calico [23], with a goal similar to CoMo's: aiding users in design. They evaluated Calico by running a pilot study involving 16 pairs of participants designing a traffic light simulator with the system and found that Calico helped users even if they did not use all of its features. Mangano et al. determined that more training may be necessary for people to take full advantage of their system's features.

One final system that warrants mention is Roy's SYNBAD [27]. Like SPT [28], SYNBAD uses a SAT solver to synthesize code from constraints derived from a set of concrete examples of manipulations, but boasts an improved synthesis time over SPT. Future versions of CoMo may wish to take advantage of SYNBAD's improved runtime.

Chapter 3

Observational User Study

In this chapter, I describe a user study that I conducted to observe how data structure manipulations are explained at a whiteboard. I asked pairs of subjects to discuss two data structure manipulations and ensured that the pairs consisted of one subject who was familiar with the manipulations (e.g., a computer science undergraduate) and one who was completely unfamiliar with them, even how they were drawn. The user study presented here served as a guide to CoMo's interaction. In this user study I concluded that CoMo should:

1. Remember the names of the data structures and manipulations it has encountered and recall them when asked.

2. Use direction to determine what kind of pointer an arrow represents.

3. Allow the user to describe manipulations either as input and output states or with detailed pointer manipulations.

4. Allow the user to edit structures in place.

3.1 Participants

I recruited 20 undergraduate volunteers and asked each one:

1. whether they were a CS major

2. whether they had heard of a linked list

3. whether they had heard of a binary tree

4. when they were available to participate

Based on their response, each was classified as either a "teacher" or a "student." Teachers and students were paired based on their availability, and asked to spend an hour in a conference room teaching and being taught the manipulations. Subjects were not notified of whom they were paired with before the session began.

	Teachers	Students
Male	9	2
Female	1	8
Right-handed	9	9
Left-handed	1	1
Electrical Eng. and Comp. Sci. (EECS)	7	-
EECS & Math	3	-
Undeclared	-	4
Mechanical Engineering	-	3
Brain and Cognitive Sciences (BCS)	-	1
Chemistry & Linguistics and Philosophy	-	1
Music and Theater Arts & BCS	-	1

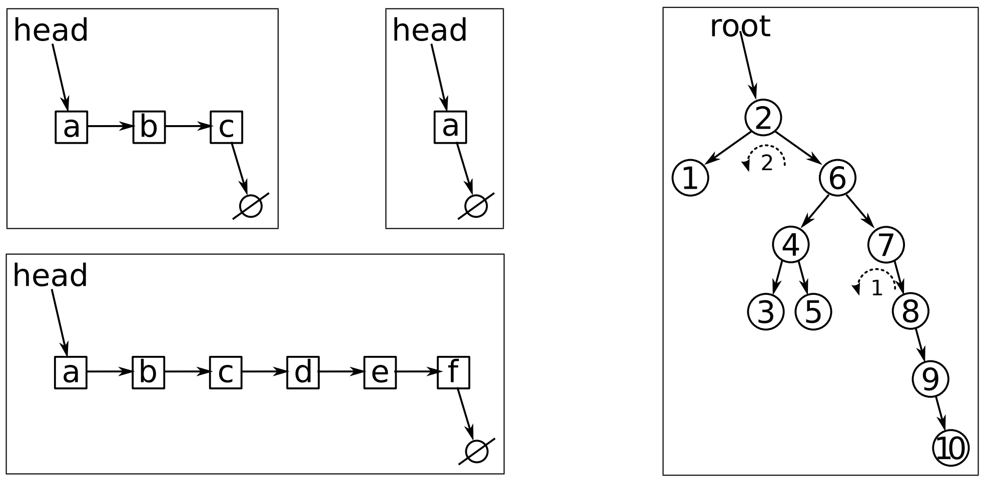
Table 3.1: Participants' Demographic Information

Table 3.1 describes the participants' demographic information[1]. Teachers had taken an average of three relevant computer science courses that included topics such as "Introduction to CS," "Software Engineering," and "Algorithms." Two out of the ten students had taken a "CS for non-CS Majors" course. One student had taken "Advanced Placement CS" in high school. Two students had some past experience programming. No students had been taught anything about either of the data structure manipulations before the session.

[1]Two of the undeclared students planned on declaring EECS.

3.2 Procedure

Each session consisted of a teacher describing two data structure manipulation descriptions to a student, one manipulation at a time. I chose two simple data structure manipulations: a singly-linked list reversal and a binary tree left rotation. The teacher was asked to teach the linked list reversal first because it was simpler and students were complete novices. I assumed that easing them into the abstract concept of data structure manipulations would make it more likely that students could learn the manipulations effectively, thereby engaging in interesting interactions to study. Unlike [2], I decided against participating in the conversations, as I wanted to observe how data structure manipulations were both learned and taught as well as what form the learning interactions would take without my influence.



(a) Reverse these lists.

(b) Balance this tree.

Figure 3-1: Exercises teachers were asked to give students.

To test the students' understandings, teachers were asked to give their students a set of prepared exercises (Fig. 3-1) at the end of each description.

Sessions were held at a whiteboard in a private conference room and videotaped. The camera was visible to the subjects throughout the session. Before the session began, subjects were briefed privately in the same room. Each briefing consisted of asking the subject to read and sign a consent form. Next, the subject was asked to review printed instructions and ask any questions they might have. The teacher was briefed before the student. The subject not being briefed was asked to step out of the room during their partner's briefing. When both subjects had been briefed and all their questions had been answered, the camera was started and they were asked to begin the linked list reversal description.

During the descriptions, I remained in the room working at a computer with my eyes averted from the subjects. My purpose was to start and stop the camera and ensure it was working during the descriptions. I did not interact with the subjects or answer questions during either description, however, subjects were given an opportunity to ask questions between descriptions. I took them out of the room to answer any questions they might have individually.

After both descriptions were done, the subjects were given a two-page questionnaire that presented ten statements the subject was asked to evaluate on a five point Likert scale, asked them to provide demographic information including their previous programming experience, and gave them an opportunity to give any additional comments on the session. Subjects were informed that their partner would not see their responses, and that I would not look at them until they left the room. The teachers' and students' questionnaires differed slightly in their Likert statements (discussed below). When they completed their questionnaires, subjects were thanked and compensated with two movie ticket vouchers.

3.2.1 Instruction Sheets

Teachers and students were given different instruction sheets. The teachers' instruction sheets consisted of two double-sided pages, the first page outlined the linked list reversal, the second outlined the binary tree left rotation. The fronts of these pages contained brief descriptions of the manipulations they were to teach, and the backs contained exercises to give to the student after they understood the manipulation (Fig. 3-1). I ensured that the teachers were already familiar with the manipulations presented; the data structure manipulation instructions were there only to ensure the teachers had an aide to get back on track, should the need arise. A cover sheet was included with the teachers' instructions. This stressed four key points (discussed below) to ensure a rich interaction took place. The students' instructions were much simpler. They consisted of one single-sided page explaining, at a high level, what data structures and data structure manipulations are. I ensured that students had little or no experience with data structures, so their instruction sheets presented them with some example data structure diagrams as well (Fig. 3-3).

Teachers' reminders:

1. **Explain the process**—don't just write pseudo code.
2. Don't worry about being perfect, do your best. You are acting as a teacher—**this is not a job interview**.
3. Do your best to **make sure your student understands** the manipulation before showing them the examples.
4. Use this prompt only as a guide. **Do not show your student this prompt** as a teaching aide.

Students' reminders:

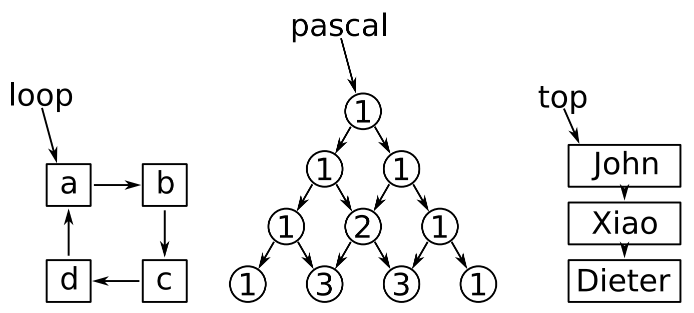
5. Ask questions if something is confusing.
6. It's okay to draw on the board, yourself.

Figure 3-2: Reminders presented to subjects in order to encourage a rich interaction. These were placed on the front sheet of subjects' instructions.

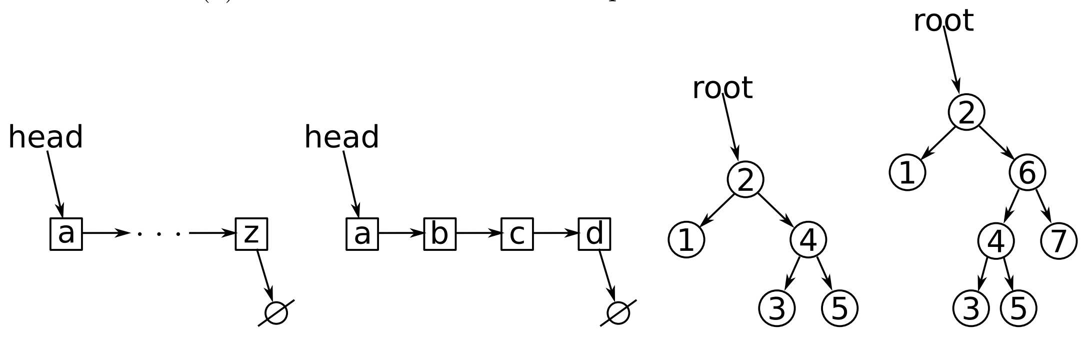
Teachers' cover sheets and students' instruction sheets made an effort to stress things that would encourage a rich interaction. Fig. 3-2 quotes the reminders in the instructions I gave. I wanted to avoid situations where the teacher simply wrote pseudo code to explain the manipulation (item 1), was overly nervous about getting the details right (item 2), or showed the student the images on the prompt instead of drawing on the whiteboard themselves (item 4). I also wanted to ensure the student understood the manipulation before being given the exercises (item 3), and was willing to ask questions (item 5) and use the whiteboard themselves, without making the student feel as though they *must* (item 6).

3.3 Hypotheses

Before beginning the study, I posited several hypotheses. These hypotheses came from my previous experience in teaching computer science, and from my previous work in designing and creating sketch interfaces. The first hypothesis—implicit in the teachers' instruction sheets—was that having someone successfully complete exercises can determine whether a person understands a manipulation.



(a) Data structure vocabularies presented to students.



(b) Data structure vocabularies presented to teachers.

Figure 3-3: Sample visual data structure vocabularies presented to subjects.

I hypothesized that for the purposes of creating a system capable of holding a limited multimodal conversation, fixing the system's sketch vocabulary—the set of symbols the system recognized—would simplify implementation without sacrificing usability. I further hypothesized that the sketch vocabulary consisting of nodes drawn as boxes or circles and pointers drawn as arrows would be one such vocabulary. Fig. 3-3 shows a sampling of the example structures I presented to subjects in their instruction sheets. I settled on this vocabulary, because I use it myself, and I have noticed teachers and colleagues use it.

In traditional classroom environments, students draw only on their notepads. The teacher is the only one expected to draw on the whiteboard unless a student is explicitly instructed to do so. I hypothesized that—despite being explicitly labeled as "student" and "teacher"—the student would naturally interact with the whiteboard during the sessions, because people learn better when interacting with the same surface.

My final hypothesis was that, not only do people prefer interacting with the same *surface*, but they prefer interacting with the same *structure*. That is, I hypothesized that when teachers drew the required exercises to test their students' understanding, the student would *prefer* to draw over the same structure as opposed to drawing a new structure that reflected the changes.

3.4 Limitations

Students' prompts may have overly biased them to draw at the board by including item 6, weakening my ability to draw a positive conclusion on the "same surface" hypothesis.

Studying pairs consisting of one expert and one novice prevented me from determining how people might cooperate to settle on a visual vocabulary. With this study's setup, I cannot determine whether more experienced students may have attempted to alter the visual vocabulary their teachers presented.

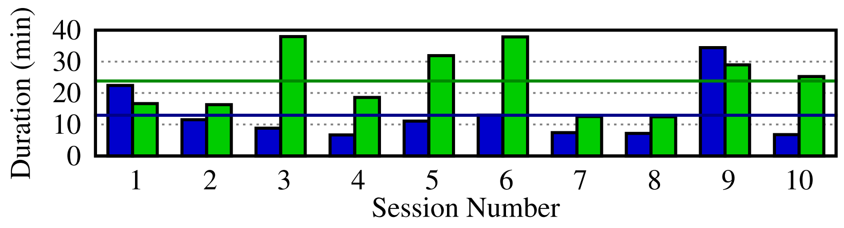
Asking teachers to give students exercises may have biased them toward confirming the hypothesis that successfully completing exercises accurately tests students' knowledge. On the other hand, it gave me the opportunity to see whether they strongly disagreed with it, as they were free to choose not to give any exercises.

I asked participants to discuss only simple data structure manipulations. Had I asked the subjects to discuss more complex manipulations, I may have gotten more interesting observations involving aspects of more complex manipulations. For example, the structure of this study did not allow me to determine how people describe the balancing algorithm involved in an AVL tree. I decided against using more complex manipulations for practical reasons; finding a student demographic that was knowledgeable enough to understand these more complex manipulations but that had not yet seen them would be prohibitively difficult.

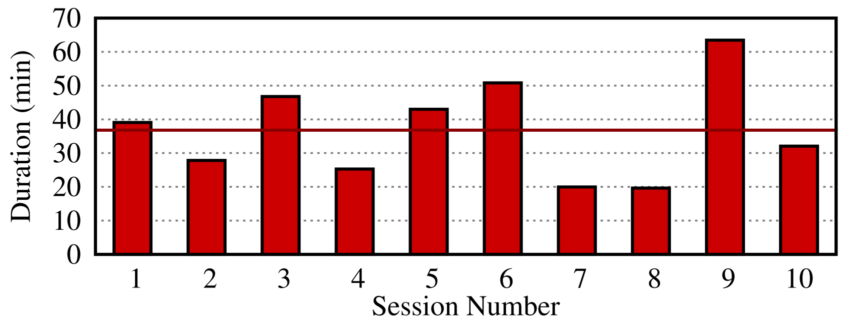
3.5 Results

The videos and questionnaires were reviewed and aggregated. This section reports the aggregate results.

3.5.1 Analysis



(a) The durations of each description by session, with averages.



(b) Combined total of both descriptions by session, and average.

Figure 3-4: The time it took each session's descriptions and their sum.

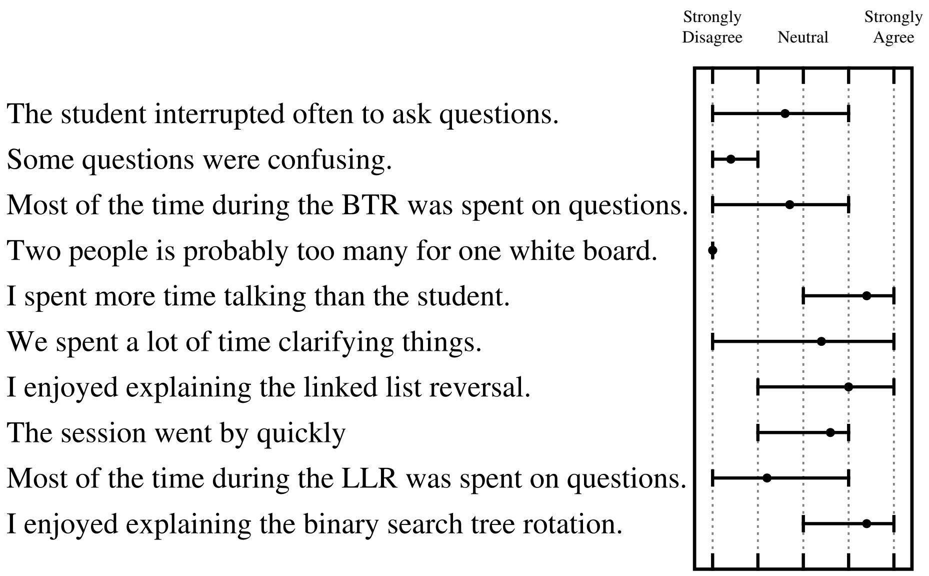
Fig. 3-4 shows each description's duration. The blue bars (left) in Fig. 3-4a show the first session (linked list reversal) and the green bars (right) show the second session (binary tree rotation). The blue line (lower) denotes the average linked list time of 12:58 and the green line (upper) denotes the average binary tree time of 23:51. Tree descriptions took 84.4% longer on average. Plots in Fig. 3-4b are the sum of the plots in Fig. 3-4a. The shortest session (8) took 19:39 and the longest (9) took a total of 63:26. Average session length was 36:47 (denoted by the horizontal line in Fig. 3-4b).

All durations were measured between the time the teacher began talking about the data structure manipulation and the time the student completed the final exercise.

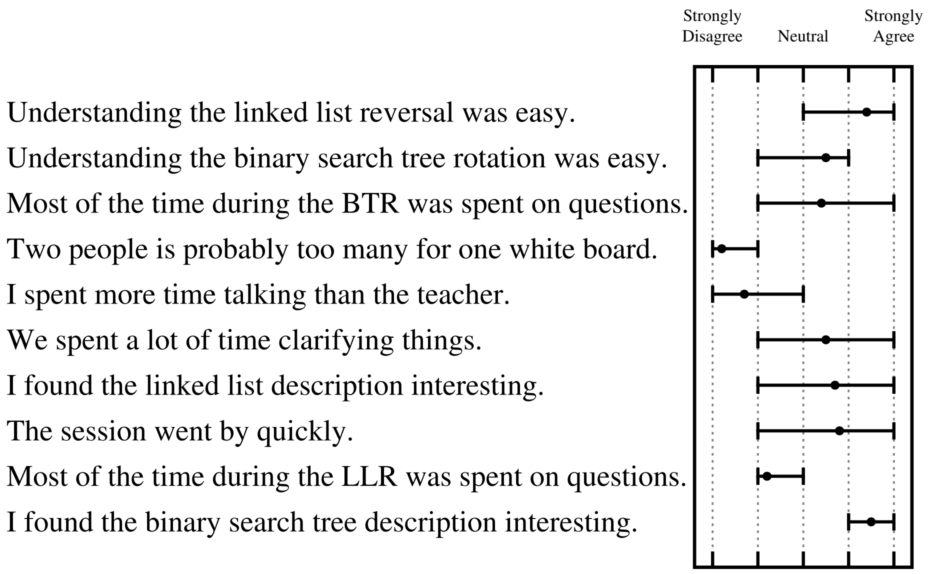
Unsurprisingly, list reversals took much less time than tree rotations on average. However, in sessions 1 and 9 the list reversals took more time than the tree rotations. After reviewing the recordings of these sessions, I determined that there were two different reasons for this. In session 1, the teacher was quite nervous and ended up stumbling over themselves. By the second description, they had relaxed enough that the description went more smoothly and thus more quickly. In session 9, the participants happened to know each other[2]. The teacher knew that the student wanted to be a computer science major, so in addition to describing the manipulation in great detail, they also described the inner workings of a computer to clarify why algorithms must proceed as they do. For example, the teacher explained why temporary pointers are needed: they ensure memory references are not lost. This extensive background knowledge did not need to be repeated, so the second description (though more complex) took less time.

Though students were explicitly told they *could* draw on the board, their instructions did not *require* them to do so. As a result, the students in sessions 3 and 9 sat at a table and took notes on their instruction sheets. They did not get up to interact with the teacher at the board before the exercises were given. To reference something on the board while asking a question, these students pointed at the item they were referencing without getting up from their seats. Student 9 came up only to perform the exercises the teacher gave them. Interestingly, student 3 attempted to do the exercises on their instruction sheet. For the simple linked list manipulation, this was not a problem. For the tree manipulation, however, the student had trouble working through the exercise on their own, and approached the board to work with the teacher's initial drawing. This suggests that working with the same drawing may make it easier to think through manipulations than attempting to memorize the manipulation and work on a separate surface. However, since this is this study's only observation of this kind, I cannot draw a concrete conclusion.

[2]Remember: subjects were not told whom they had been paired with until entering the room.



(a) Statements presented to teachers.



(b) Statements presented to students.

Figure 3-5: Subjects' responses to qualitative questions on a 5-point Likert scale. Average responses are displayed between minimum and maximum values. Note that the manipulation names were abbreviated to fit in the figure; subjects were presented with unabbreviated text.

Fig. 3-5 shows the aggregate results of subjects' level of agreement or disagreement with the statements. These questions were meant to determine whether the general structure of the interaction was correct, i.e., whether interactions of this kind were enjoyable for the subjects. I found that teachers did not think an unusual amount of questions were asked and that they felt they understood the questions that were asked. Unsurprisingly, students found the tree manipulation to be more difficult to understand than the list manipulation and were slightly more critical of themselves than the teachers were of them. Teachers and students agreed that teachers spent more time talking than students and that subjects were comfortable writing at the board together. In all, I determined that the interaction was enjoyable and productive for both subjects.

In seven of the ten sessions (eight of the twenty descriptions) the teacher began by asking the student whether they had heard of the data structure, for example by asking, "Have you heard of a linked list before?" or "Do you know what a linked list is?" In each case, the student replied that they had not, so the teacher started by explaining what the data structure was, why one might use it, and (in some cases) how it differed from similar structures.

Session 3 was one such session. The teacher asked the student, "Do you know what's a tree in computer science?" Due to the teacher's accent, the student misunderstood the teacher to be asking if the student knew how to represent a three in binary. After repeated verbal attempts to clarify what they meant, the teacher finally drew a biological tree and a binary tree. They pointed to the drawing of the binary tree and asked the student if they had seen one before. This was an unexpected observation. When verbal communication failed, the teacher resorted to the next available mode of interaction: sketch. This successfully communicated the question and the interaction was able to continue.

In eleven of the twenty descriptions, teachers used the visual vocabulary presented in the instruction sheet. In eight of twenty descriptions, the visual vocabulary used was very similar, but not identical. In only one instance (the list description in session 1) was the vocabulary used significantly different from the one presented in the teacher's instructions. Fig. 3-6 shows an example of the vocabulary used in session 1. This suggests that the vocabularies I chose were reasonable, supporting my hypotheses. Further, in all descriptions, students copied their teachers' visual vocabularies with no apparent difficulty[3]. This, again, supports my hypothesis that setting a visual vocabulary does not interfere with a person's ability to understand a data structure; even complete novices are able to quickly learn and understand the meaning of structures drawn with this visual vocabulary. Once teachers chose a vocabulary, they did not change it. However, in four of the twenty descriptions teachers spontaneously introduced a shorthand sketch vocabulary. Presumably to speed up drawing, teachers omitted the outline of nodes (drawing only its contents, e.g., "`a`" or "`7`") and the heads on arrows (drawing them as simple lines). The only characteristics defining the data's structure were nodes' and lines' relative positions. For example, binary trees were drawn with left and right children at their bottom left and bottom right, connected by a line. Fig. 3-7 shows an example of the type of shorthand observed (top-right) next to an example of the visual vocabulary used throughout most of the conversation (left). The placement of the nodes and pointers was unambiguous and consistent enough for the student to understand the new visual vocabulary immediately, without the need for an explanation.

In six of the tree descriptions (but none of the list descriptions), teachers gave their students additional exercises before giving them the exercises in their instructions. This may indicate that one way people choose to verify someone else's understanding is to quiz them. However, this observation may have been due to the bias present in the instruction sheets; asking the teachers to give examples likely influenced them. Due to this bias, I cannot conclude that successfully demonstrating examples necessarily means a person understood a manipulation fully.

In all but session 9, teachers demonstrated the manipulations by drawing the state of a structure with a concrete number of nodes as it appears before and after the manipulation. They did not go into the details of explaining the need for temporary

[3]Recall that this was the first time students had seen drawings of this kind.

pointers or how to manage memory[4]. This suggests that the way humans think about data structure manipulations is in terms of input and output states. There is an important distinction between the desired manipulation (defined by its results) and the code that implements that manipulation (largely influenced by the constraints of computer architectures and programming language requirements). The difference lies in details such as how to maintain memory (e.g., not dropping nodes and avoiding dangling pointers). This observation demonstrates the difficulty students have when learning to program, validating one of the findings from Good et al. [12]; people do not naturally think about code and algorithms, they think about states and effects.

The final interesting qualitative observation I made was that three students attempted to perform exercises on the binary trees their teachers drew. Instead of redrawing the trees in their rotated states, these students erased and redrew elements of the tree to reflect the change. However, in all cases they gave up their attempts to edit the original structure and instead drew the end state next to the original drawing. After reviewing the videos, I determined this decision likely due to the complexity of tree rotations combined with the tedious nature of erasing and redrawing parts of a data structure with physical ink.

3.6 Architectural Implications

The above analysis suggests that CoMo needs to be able to do the following:

1. Remember the names of the data structures and manipulations it has encountered and recall them when asked.

2. Use the direction an arrow is drawn in to determine what kind of pointer it represents.

3. Allow the user to describe manipulations either as input and output states or with detailed pointer manipulations.

[4]Recall that in session 9 the teacher and student were friends. The teacher gave a thorough explanation of how to *implement* the manipulations, not just a description of what they were.

4. Allow the user to edit structures in place.

In this section I describe the features included in CoMo that resulted from these requirements.

3.6.1 Remembering Names

As described above, descriptions often began with teachers asking questions like, "Do you know what a linked list is?" Had the student answered affirmatively, the teacher would not have had to explain the structure and the interaction could have moved on to the explanation of the manipulation. Presumably, they may have even asked "Do you know how to reverse a linked list?"

As a result of this observation, CoMo remembers the names of structures and manipulations that it has encountered. With this simple memory, it can answer questions of the sort seen in the study. For example, when asked "Do you know what a linked list is," it can answer "Yes." and display a linked list description from a previous interaction. This results in a far more succinct explanation, and it eliminates one source of possible user errors: explaining the structure.

3.6.2 First-Pass Geometry Heuristic

Based on the observation that subjects understand geometrically consistent shorthands, CoMo's initial classification of arrows as pointers relies on the direction the arrows were drawn in. For example, when drawing a doubly linked list, CoMo takes note of the direction that `next` and `prev` pointers are drawn. When a new pointer is drawn, CoMo compares the new pointer's direction to the direction of all previously-drawn `next` and `prev` pointers, and classifies the new arrow as the type of pointer with the most similar direction.

3.6.3 Input and Output States

Since the majority of the descriptions observed in the user study consisted of teachers describing only before- and after-state pairs, CoMo assumes the user will draw a structure and edit it in-place. For example, when reversing a linked list, CoMo assumes the user will draw a list with some concrete number of nodes and then click-and-drag the pointers' heads to reorder the list in reverse. CoMo then uses these before and after states to synthesize code. Requiring the user to think about the order of operations, dropped nodes, and temporary pointers are implementation details that are not important for the purpose of understanding the manipulation, so CoMo fills in these omitted implementation details.

However, since one pair of subjects worked through the exercises on a pointer-by-pointer basis, CoMo also allows users to work through the process of a manipulation, pointer-by-pointer as a computer would. As it turns out, the code synthesis system that CoMo uses accepts only input- and output-state pairs. So, regardless of whether the entire process is outlined or just the before and after states, CoMo (and its code synthesis system) use only the first and last states of each example to synthesize code.

3.7 Summary

This chapter presented results from an observational user study I performed. I asked pairs of subjects consisting of one expert and one novice to discuss a linked list reversal and a binary tree left rotation. I confirmed the hypotheses that a visual vocabulary consisting of nodes drawn as boxes or circles and arrows is reasonable and that fixing a vocabulary does not appear to interfere with a person's ability to understand a data structure manipulation. I made some interesting and unexpected observations, including:

1. Teachers sometimes ask about whether their students had heard of the structure before beginning the explanation

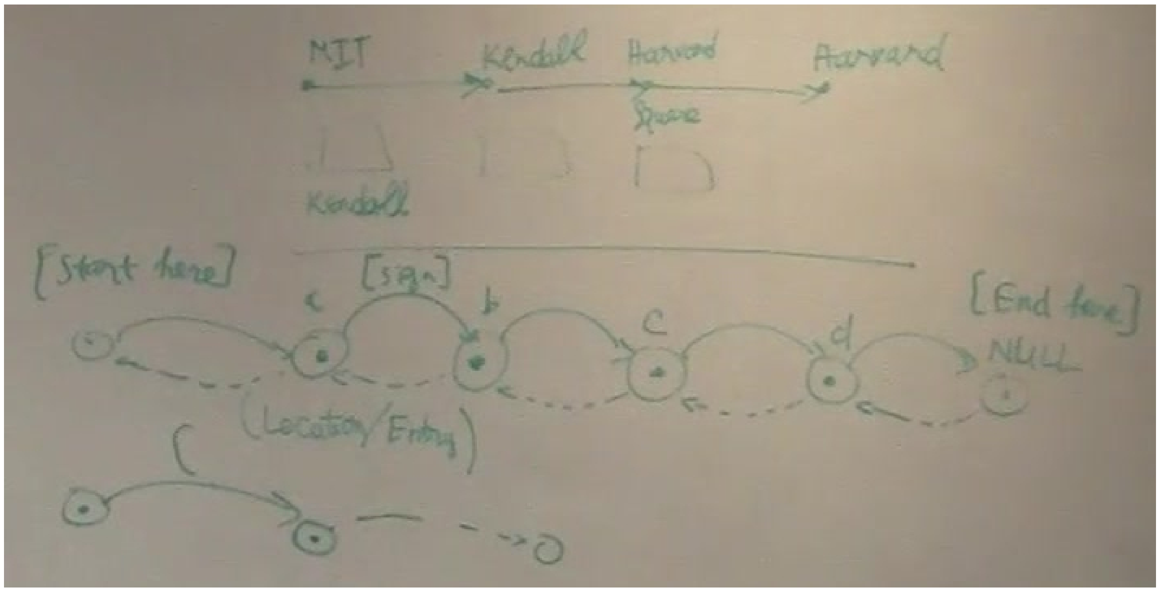
2. One teacher *drew* a question when verbal communication repeatedly failed

3. The vast majority of teachers explained manipulations as a set of input and output states on concrete examples.

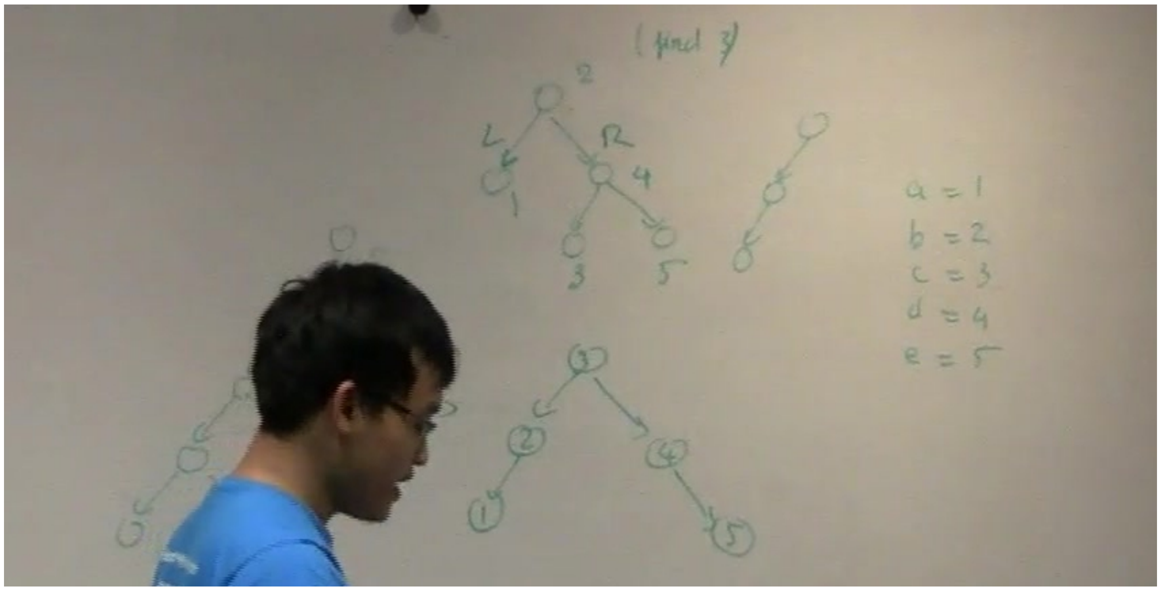
The findings and observations from this study guided CoMo's implementation. The following features were implemented as a direct result of the study:

1. CoMo remembers the names of the data structures and manipulations it has encountered and recalls them when asked.
2. CoMo uses direction to determine what kind of pointer an arrow represents.
3. CoMo allows the user to describe manipulations either as input and output states or with detailed pointer manipulations.
4. CoMo allows the user to edit structures in place.
5. CoMo attempts to mimic the user's drawing style by copying their drawn symbols when generating structures of its own.

The next chapter describes CoMo from a user's point of view. It goes through an example interaction with CoMo, and highlights several interesting features. Chapter 5 continues the description of CoMo, by explaining the mixed-initiative code-generation framework that CoMo implements to guide the interaction.



(a) An example of the vocabulary used in session 1 for the linked list reversal. Note that it is significantly different from the vocabulary presented in the instruction sheet.



(b) An example of the vocabulary used in session 1 for the binary tree left rotation. Note that it is nearly identical to the vocabulary presented in the instruction sheet.

Figure 3-6: Examples of the vocabularies used in the first session.

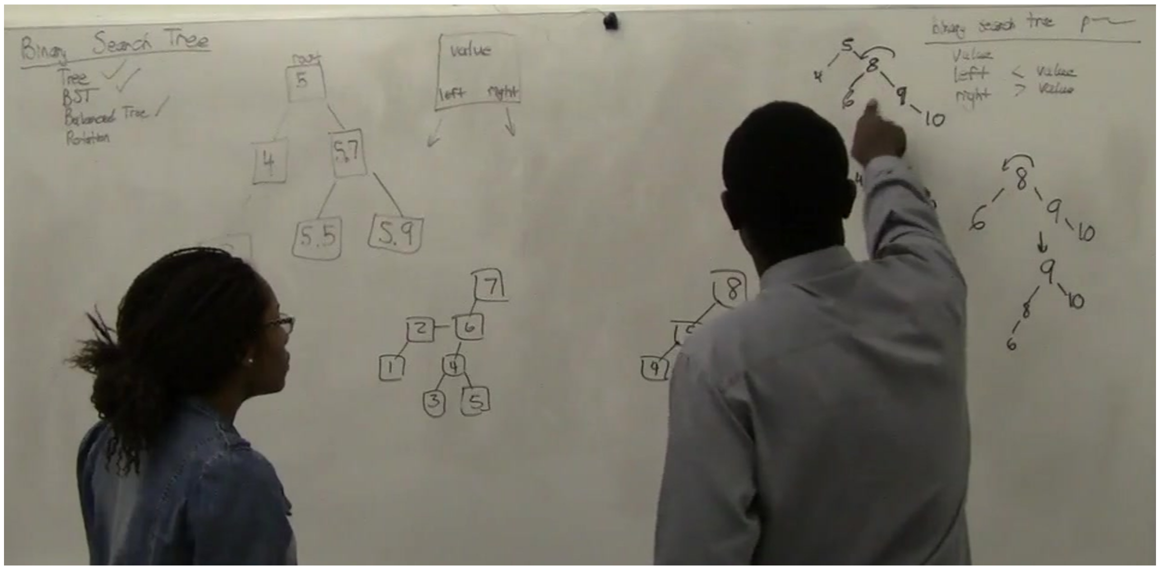


Figure 3-7: An immediately understood visual shorthand.

Chapter 4

CoMo: an Overview

This chapter describes the interaction with CoMo from a user's point of view. It describes what CoMo requires and how that affects its interaction with the user. Chapter 5 describes the same interaction presented in this section, but from an implementation perspective.

4.1 Basis on the User Study Findings

Based on the findings from the user study presented in Chapter 3, CoMo implements the following features:

1. CoMo remembers the names of data structures and manipulations and is able to recall them when asked by the user to do so.

2. CoMo recognizes pointer types based on the direction they are drawn in.

3. CoMo allows the user to describe manipulations either as input-output state pairs or as a detailed manipulation process.

4. CoMo allows the user to edit structures in place.

5. CoMo attempts to mimic the user's drawing style by copying their drawn symbols when generating structures of its own.

Before demonstrating these features in use by describing an example interaction, the next section describes the user interface that CoMo presents the user. The following section describes an interaction with the assumption that the user is completely familiar with CoMo.

4.2 The User Interface

Before diving into the details of an example interaction with CoMo, this section describes the interface presented to the user.

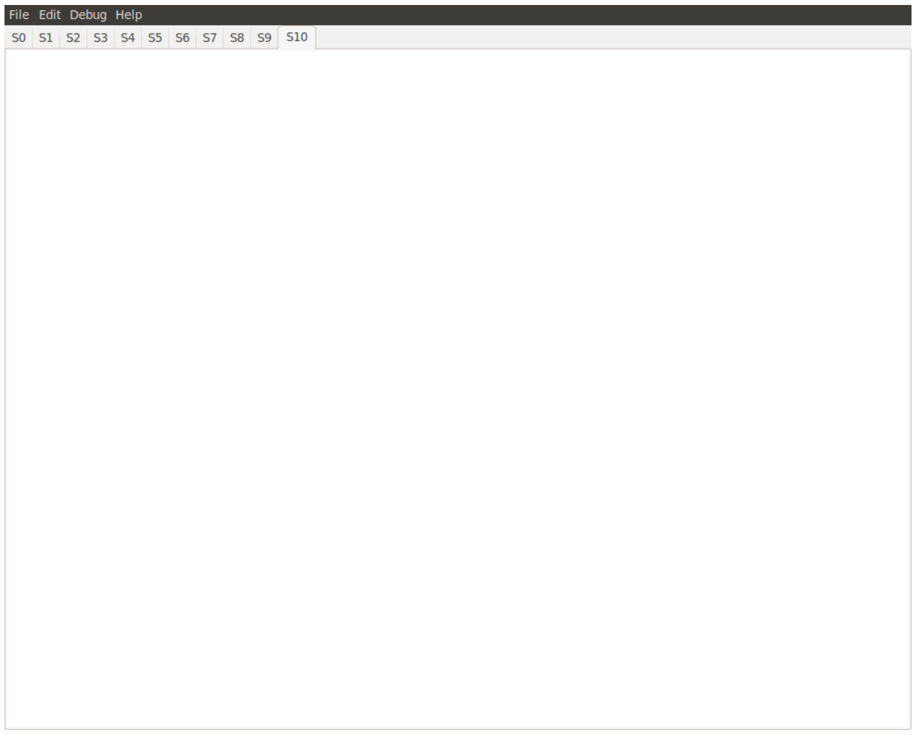
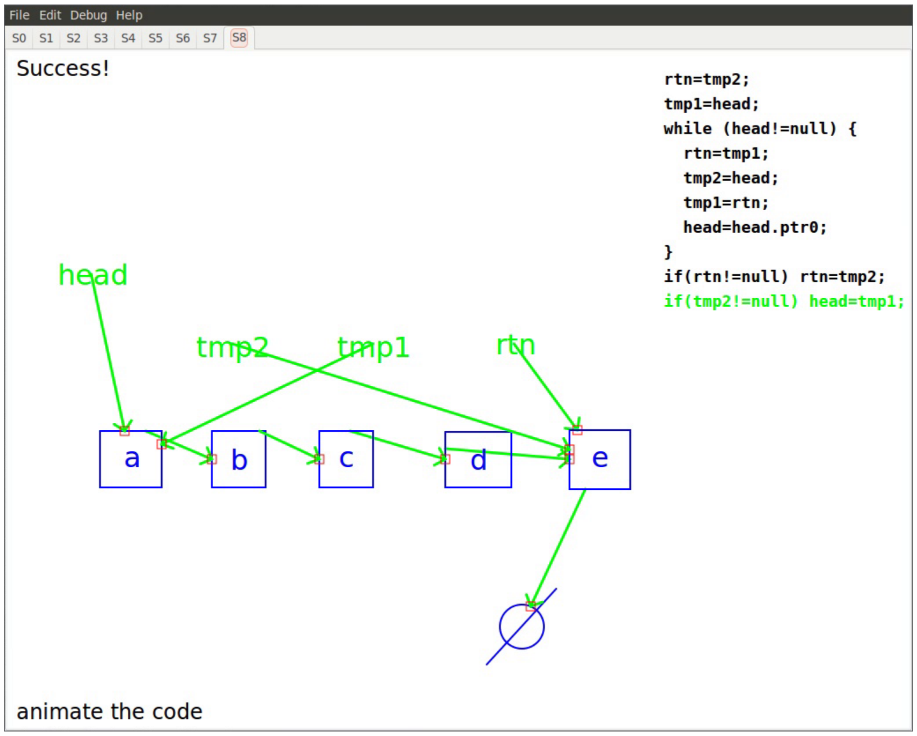


Figure 4-1: CoMo's User Interface

Fig. 4-1 shows CoMo's user interface. The largest space is reserved for the canvas—the area of the screen that can be drawn on and serves as the primary mode of input to and output from CoMo. The user draws in this area and CoMo interprets the drawings as data structures. Above the canvas is a row of tabs corresponding to separate canvases. As the interaction with CoMo continues, more tabs appear as more canvases are created—either when the user says, "Make a new canvas" or when

CoMo creates one to ask a question about an example data structure.



```
rtn=tmp2;
tmp1=head;
while (head!=null) {
  rtn=tmp1;
  tmp2=head;
  tmp1=rtn;
  head=head.ptr0;
}
if(rtn!=null) rtn=tmp2;
if(tmp2!=null) head=tmp1;
```

Figure 4-2: CoMo's Interface: Populated

Fig. 4-2 shows the canvas after some interaction with CoMo. The bottom-left corner of the canvas displays the most recently heard utterance, the top-left corner displays a status message to the user, and the top-right is reserved for synthesized code. In this figure, we see CoMo after having completed animating the code seen in the top-right. The phrase "animate the code" appears at the bottom, because it was the user's utterance that initiated code animation. Code is highlighted in green when playing; we see that CoMo has just finished animating the final line of code.

4.3 Linked List Find Last

This section goes through an example of a user describing to CoMo how to find the last element of a linked list.

4.3.1 Initiating the Discussion

The user must initiate the interaction by either drawing a concrete example of the manipulation or asking if CoMo has heard of a linked list before. The user begins by asking, "Do you know what a linked list is?" CoMo responds by saying, "No" and awaiting further input.

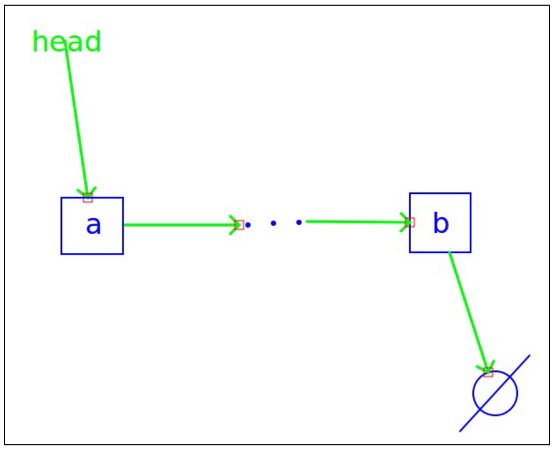
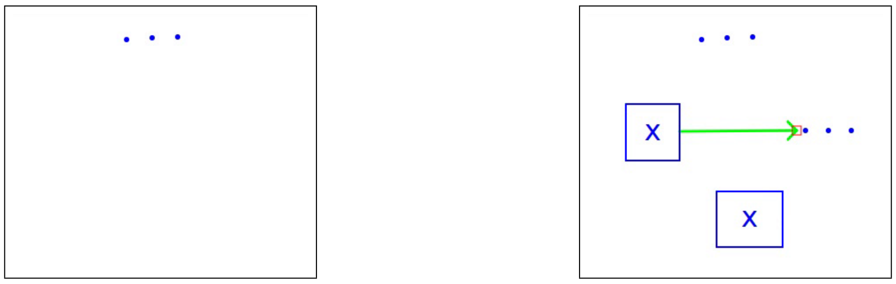


Figure 4-3: The user draws a general list.



(a) CoMo: "What does *this* expand to?"

(b) User: "To *this* or *this*."

Figure 4-4: CoMo questions how to inductively define the ellipsis.

The user continues by explaining what a linked list is, drawing the sketch seen in Fig. 4-3. CoMo sees an ellipsis in the general structure and interprets this to mean there is some repeating pattern. It cannot infer what that patten is from the information given, so it asks, "What does this expand to?" while displaying a copy of the ellipsis at the top of a new canvas. Fig. 4-4 shows the user's response, consisting of two options: a node linked with another ellipsis, and a single node.

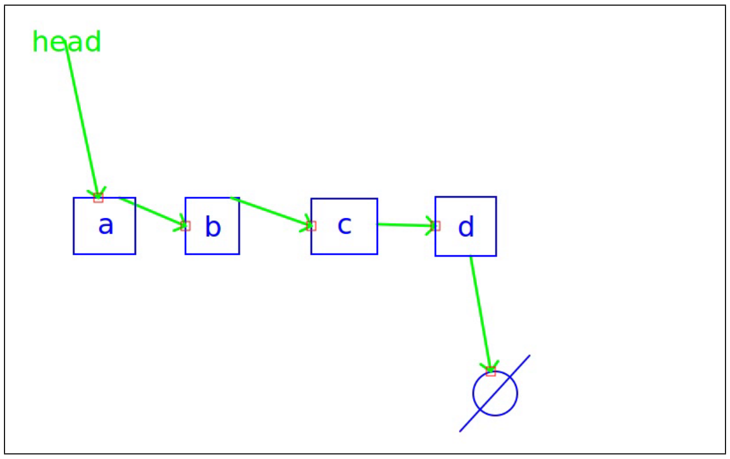


Figure 4-5: An automatically generated list with four nodes that CoMo uses to ask its first question: "How do you handle *this* case?"

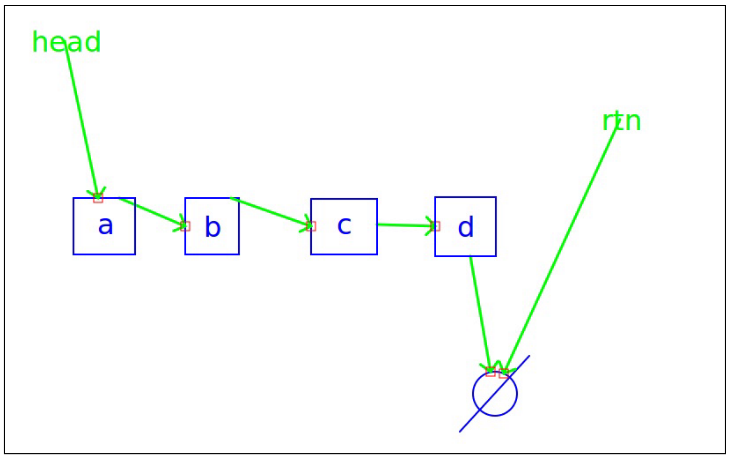


Figure 4-6: The user adds the `rtn` pointer to make this first state reflect how the manipulation begins.

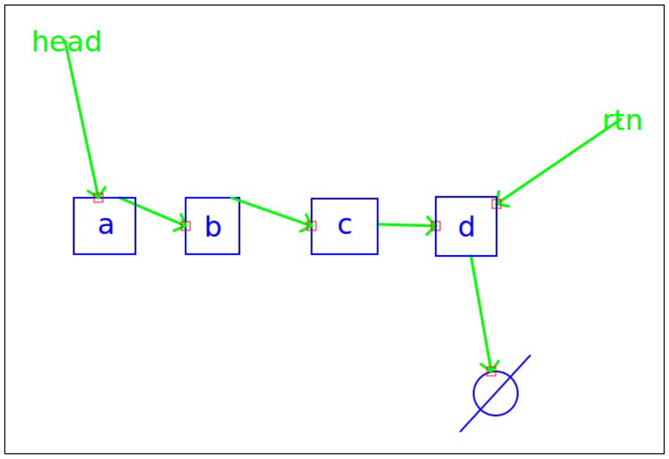


Figure 4-7: The user demonstrates the manipulation and says, "Like this."

Now that CoMo understands the structure that the user is talking about, it generates the list with four nodes displayed in Fig. 4-5 and asks, "How do you handle this case?" There is one element missing from this example: the pointer that marks the end of the list. The user draws a `rtn` pointer pointing to the null symbol in the drawing[1] (Fig. 4-6), clicks and drags the head of the `rtn` pointer to point at the end of the list, and says "Like this." CoMo reacts to the user's input in real time, displaying the last-heard utterance at the bottom left of the canvas and moving the arrow as the pen moves across the canvas, snapping it to the `d` node when the user releases the arrowhead by lifting the pen. Fig. 4-7 shows the digital canvas after the user makes the addition, update, and utterance.

While not as complex as a conversation between two humans, the interaction is more similar to a typical human-human interaction than it is to a normal computer programming session. The user approached the board with a manipulation in mind and, as though CoMo were human, started the interaction by asking a simple question: "Do you know what a linked list is?" When CoMo responded that it did not, the user explained by describing what that structure is, and then what a "find last" operation on one looks like.

CoMo allows the user to focus on describing the details that are important to the manipulation. Not once in the interaction has the user had to worry about memory management, the order of operations, or the structure of the code required to perform this manipulation. The user is able to describe nodes' contents abstractly, using labels like "`a`" and "`b`." Each node's pointer may be named "`next`," but that is a detail that can wait until the user is ready to address it. Further, since CoMo has remained responsive to the user's input by updating the sketch as the user draws new elements and modifies them, a user familiar with CoMo will know how CoMo interpreted the interaction.

[1]The user must draw the `rtn` pointer pointing at the null symbol initially to indicate its initial state to CoMo. If the pointer were initially drawn pointing at the final element in the list, CoMo would not know that the manipulation is to start with `rtn` pointing at null; it would assume that the manipulation *starts with* `rtn` pointing at the last element.

4.3.2 Continuing the Discussion

Aside from the explicit knowledge of what the input and output states of the provided example look like and the description of the general list, CoMo has been able to infer several pieces of information from the above sketches, namely:

1. Two structures are required in this manipulation.
2. One structure has the pointer: `head`.
3. The other structure has the pointer: `rtn`.
4. Nodes in this structure have one pointer, name unspecified.

With this information, CoMo is able to determine what questions remain to be asked. This section goes through the dialogues that take place about these questions.

At this point, the user has finished explaining what a linked list looks like, and drawn the result of the find-last manipulation being performed on a list with four nodes. Ideally, CoMo could start synthesizing code, however as detailed in Chapter 5, for the Storyboard Programming Tool (SPT) [28] to synthesize code correctly, several representative cases must be addressed. These cases are:

- A structure with four, one, and zero nodes, or—if there are multiple structures—the power set of comparisons between the main structure with four, one, or zero nodes and the auxiliary structure (or structures) with one or zero nodes[2].
- Comparisons between structures with one node each when one structure's node has a value greater than or less than the other's[3].

For the linked list find-last manipulation presented to CoMo here, CoMo identifies `head` as the main structure and `rtn` as the auxiliary structure. As a result, the rules above equate to the power set of comparisons when `head` points to a structure with

[2]E.g., in a stack push the main structure is the stack, and the secondary structure is the node to push onto the stack.

[3]This question requires the structure to be ordered and it requires that there be multiple structures involved in the manipulation, e.g., an ordered list insertion. SPT cannot handle comparisons between nodes with equal values, so the case where the nodes have equal values is not included.

four, one, and zero nodes crossed with `rtn` pointing to a structure with one or zero nodes. In other words, CoMo asks about the following cases:

1. The `head` pointer points at a list with four nodes and the `rtn` pointer points at one node.
2. The `head` pointer points at a list with four nodes and the `rtn` pointer points at null.
3. The `head` pointer and `rtn` pointer point at separate nodes.
4. The `head` pointer points at a list with one node and the `rtn` pointer points at null.
5. The `head` pointer points at null and a `rtn` pointer points at a node.
6. The `head` pointer and `rtn` pointer both point at null.

The remainder of this section describes the questions CoMo asks to find out what to do with these examples and how the user answers those questions.

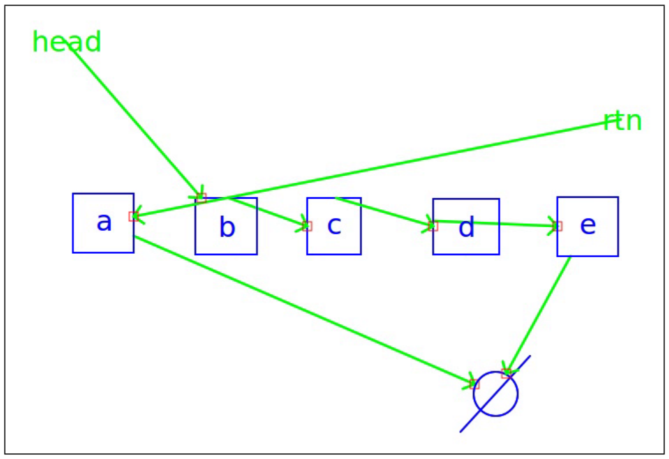


Figure 4-8: CoMo: "How do you handle *this* case?" User: "That does not make sense."

CoMo asks the first three questions in the same way: by generating an example structure and saying, "How do you handle this case?" The questions corresponding to cases 1, 2, and 3 above can be seen in Fig. 4-8, Fig. 4-9, and Fig. 4-10, respectively.

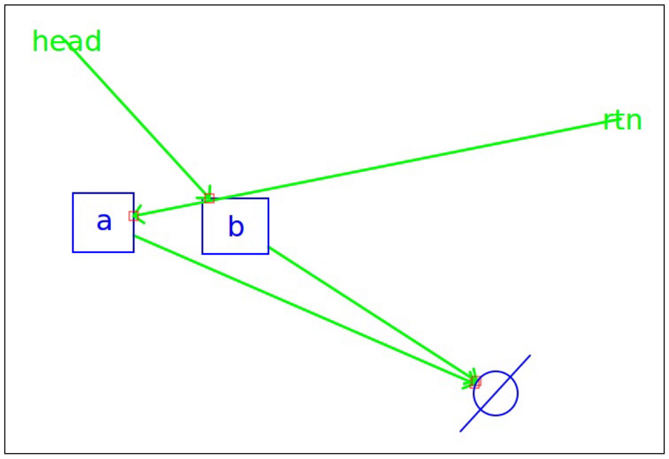


Figure 4-9: CoMo: “How do you handle *this* case?” User: “That does not make sense.”

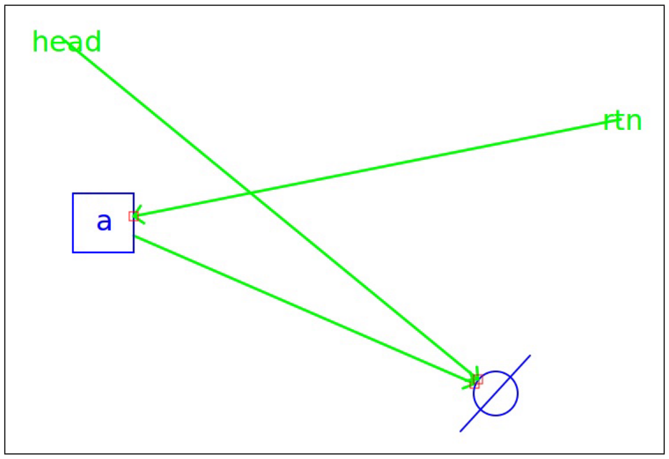
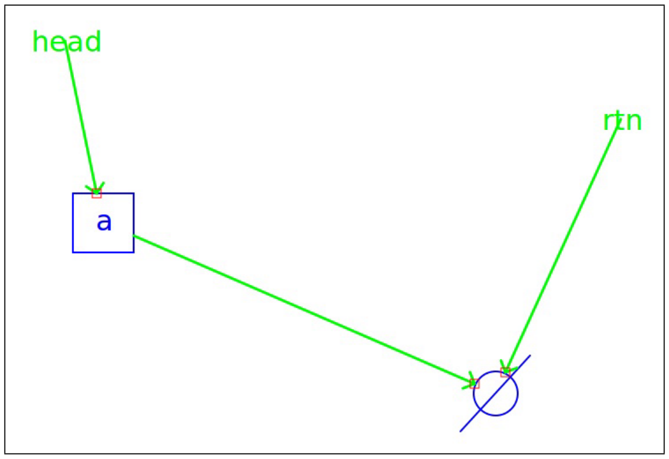
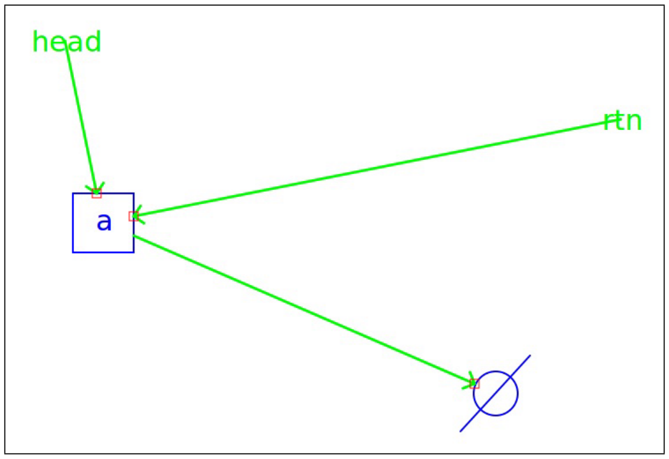


Figure 4-10: CoMo: “How do you handle *this* case?” User: “That does not make sense.”

All three of these questions involve the `rtn` pointer pointing at a node separate from the list. These questions do not make sense to the user because the `rtn` pointer is meant to act as a return statement in this manipulation. As a result, the user always wants `rtn` to start out pointing at null, so the additional node invalidates the diagram, as it does not have a place in the list. The user answers these questions by saying, "That does not make sense." This informs CoMo that these examples should not be included in SPT's description. CoMo acknowledges the user's verbal answer by displaying the understood text at the bottom-left and saying, "Okay" before asking the next question.



(a) CoMo: "How do you handle *this* case?"



(b) User: "Like this."

Figure 4-11: A question and the corresponding answer about a list with one node.

Next, CoMo asks the question seen in Fig. 4-11a, where the list has one node and the `rtn` pointer points to null. As before, CoMo generates and displays the structure while verbally asking "How do you handle this case?" Here, something must be altered during the manipulation—the `rtn` pointer must end up pointing to the end of the list. The user demonstrates the process to CoMo by clicking and dragging the `rtn` pointer to point at the `a` node. Fig. 4-11b shows the state of the canvas after the user finishes answering the question. Once they have finished updating the diagram, the user says, "Like this." CoMo interprets the utterance as a signal that the user is done answering the question and acknowledges by saying "Okay."

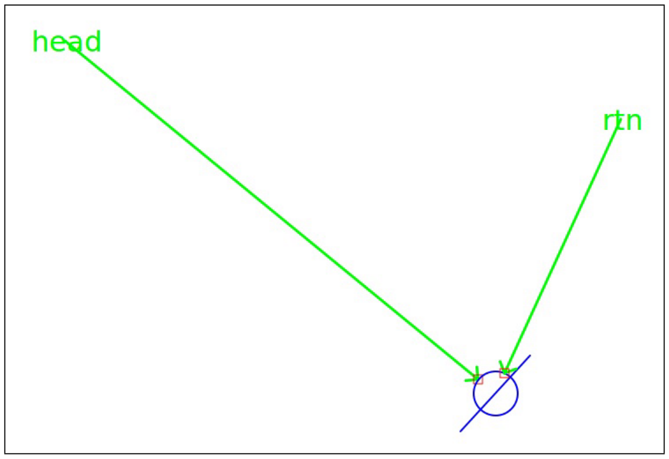


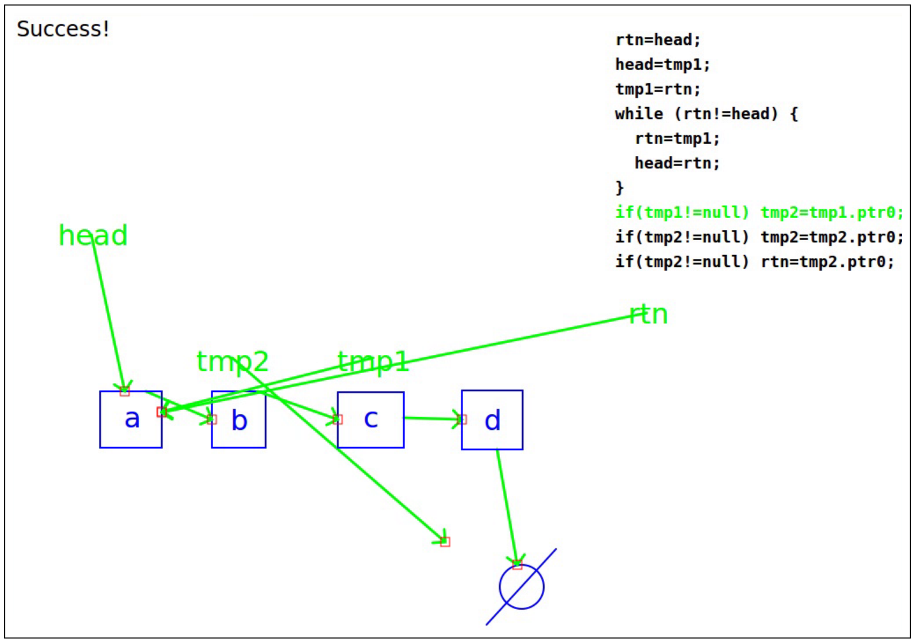
Figure 4-12: A question and the corresponding answer about an empty list. CoMo: "How do you handle *this* case?" User: "Do nothing."

Next, CoMo asks about what to do with the empty list (Fig. 4-12). In this case, there is no element to point the `rtn` pointer to. The desired output state is the same as the input state. The user says, "Do nothing" to answer this question. CoMo understands this to mean that the input and output states in this example are identical to one another. Again, CoMo signals it understands by saying "Okay."

At this point, all of CoMo's questions have been answered, so it says, "I think I understand." The user cannot think of anything else to add to the discussion, so they initiate code synthesis by saying "Generate code." CoMo responds by saying, "I will try" and begins the code synthesis process.

4.3.3 Synthesis & Verification

Code synthesis is handled by the Storyboard Programming Tool (SPT) [28] and takes less than one minute to complete on a 3.9GHz machine running Ubuntu 13.10 with 16GB of DDR3 RAM. Once the code is synthesized, CoMo displays it in the top-right corner of all the examples that have been discussed, along with "Success!" in the top-left corner, indicating code synthesis was successful.



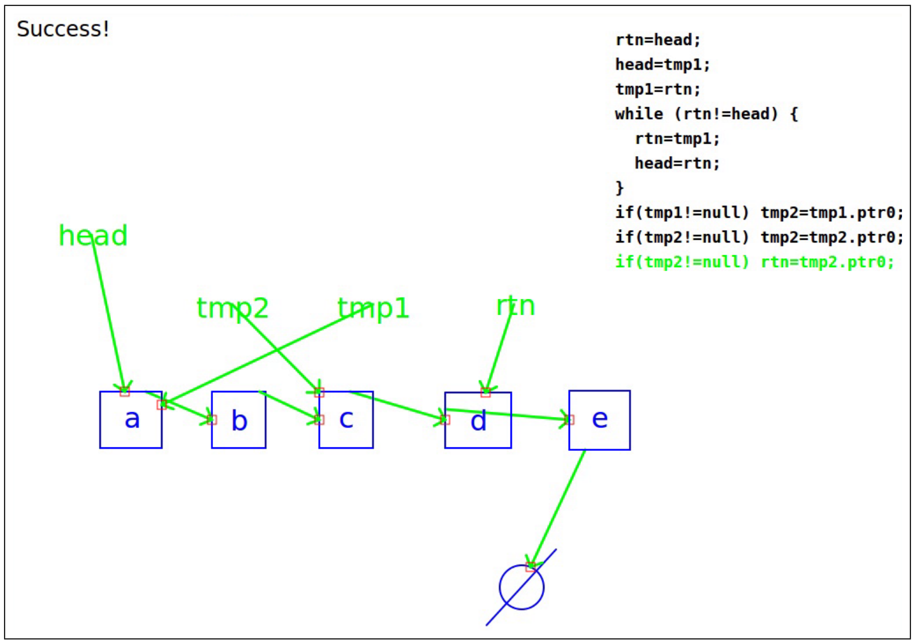
```
rtn=head;
head=tmp1;
tmp1=rtn;
while (rtn!=head) {
  rtn=tmp1;
  head=rtn;
}
if(tmp1!=null) tmp2=tmp1.ptr0;
if(tmp2!=null) tmp2=tmp2.ptr0;
if(tmp2!=null) rtn=tmp2.ptr0;
```

Figure 4-13: CoMo animates the code on the initial example with four nodes.

As discussed in Chapter 5, the code that SPT produces is unintuitive. The only way a user might verify that it works, is by running it through a test suite, or simulating it by hand on examples. CoMo facilitates the code's verification by symbolically animating the code on drawn examples. To begin verification, the user switches to the first example by clicking on the tabs provided at the top of the screen, and initiates code animation by saying "Animate the code." CoMo resets the drawing to its original state, adds the temporary pointers in the code that are missing from the sketch[4], and begins animating the synthesized code. As a line of code is animated, it turns green, and its effect is shown by smoothly moving the affected pointer from its origi-

[4]In the code seen in Fig. 4-13, these are the `tmp1` and `tmp2` pointers. SPT assumes that these nodes were initialized to null, so CoMo points them toward the null symbol on the sketch.

nal target to the target defined by the line of code being animated. Fig. 4-13 shows what the user sees as the line "`if(tmp1!=null) tmp2=tmp1.ptr0`" is animated: `tmp2` moves smoothly from the `b` node to the null symbol.

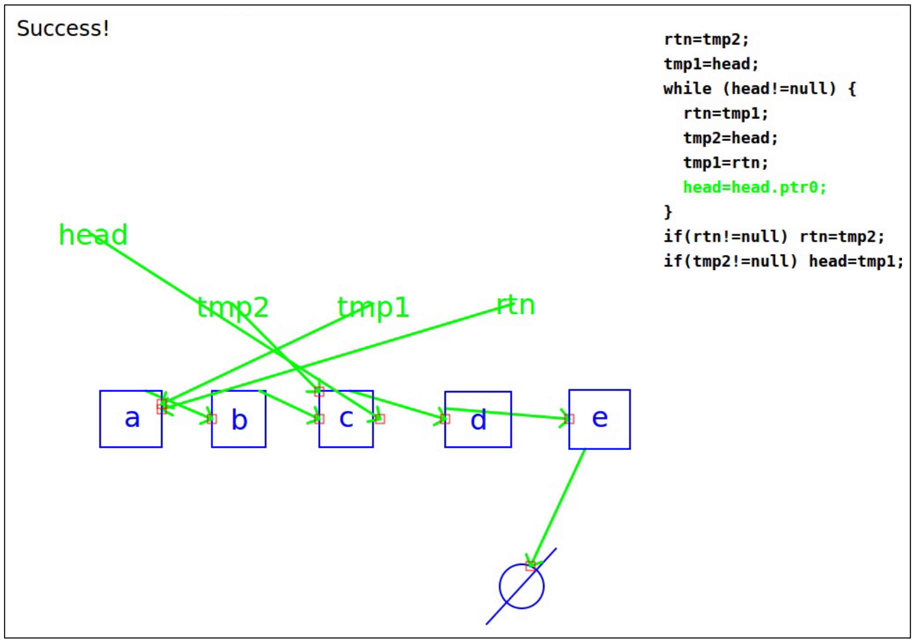

Success!

```
rtn=head;
head=tmp1;
tmp1=rtn;
while (rtn!=head) {
  rtn=tmp1;
  head=rtn;
}
if(tmp1!=null) tmp2=tmp1.ptr0;
if(tmp2!=null) tmp2=tmp2.ptr0;
if(tmp2!=null) rtn=tmp2.ptr0;
```

Figure 4-14: CoMo generates a structure with five nodes and animates the code.

As the user watches the code animate, they notice that the loop appears to execute only once in all cases. The final line in the while loop's body (`head=rtn`) assures that the while condition (`rtn!=head`) will evaluate to `false` after the first iteration, regardless of the size of the structure. The user knows that the only way to find the last element of a singly linked list is to iterate down it. The user wants to test the code on an example that CoMo has not yet seen, to verify that the code will not iterate down a longer list. CoMo facilitates this by generating structures. The user asks CoMo to generate a structure with five nodes by saying, "Generate a structure with five nodes." CoMo presents a new canvas with the requested structure and the user says, "Animate the code." Fig. 4-14 shows the final state of the manipulation, with the `rtn` pointer pointing at the fourth node in the list—not the desired state. The user corrects CoMo by saying, "No, like this" as they move the `rtn` pointer's head to the final element of the list. CoMo responds, "I think I understand." The user attempts code synthesis again by saying "Generate code." "I will try" replies

CoMo, and code synthesis begins for the second time.



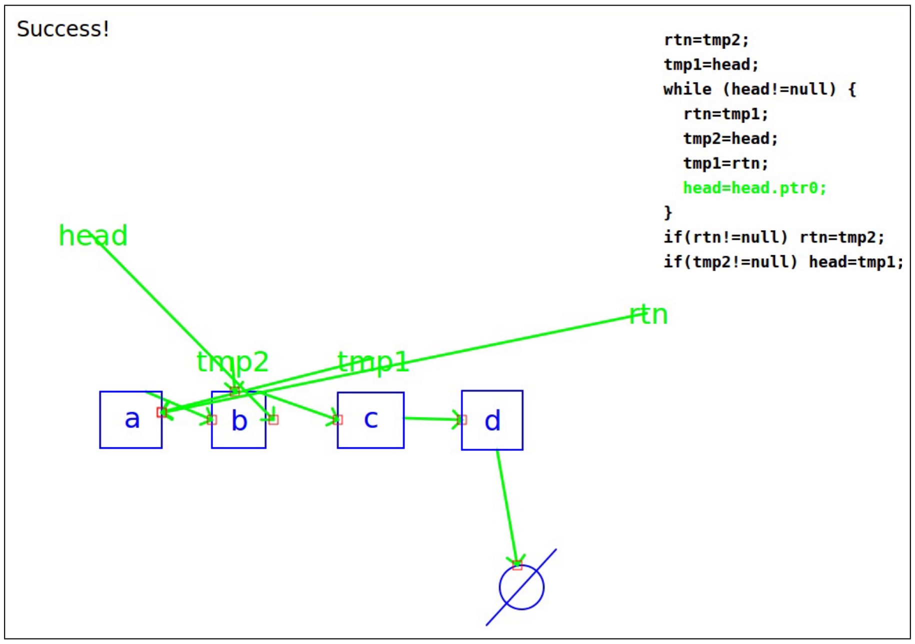
```
rtn=tmp2;
tmp1=head;
while (head!=null) {
  rtn=tmp1;
  tmp2=head;
  tmp1=rtn;
  head=head.ptr0;
}
if(rtn!=null) rtn=tmp2;
if(tmp2!=null) head=tmp1;
```

Figure 4-15: CoMo animates the new code on the same structure.

With the additional complexity introduced by the new example, code synthesis takes on the order of two minutes to complete. The code seen in Fig. 4-15 is synthesized and displayed in the top-right corner of the canvas. The user prompts code animation by saying "Animate the code" and observes how the code progresses. As the user watches this code animate, they notice that the while loop proceeds as expected, implementing an iterative solution to finding the end of the list. This code successfully executes the manipulation, so the user verifies that the new code works on the list with four elements (Fig. 4-16). As a final test, the user decides to try out a list with three nodes. They say "Generate a structure with three nodes" and, once the list is generated, "Animate the code." CoMo animates the code on the structure with three nodes successfully (Fig. 4-17), the user is satisfied that the code is correct and the interaction is complete.

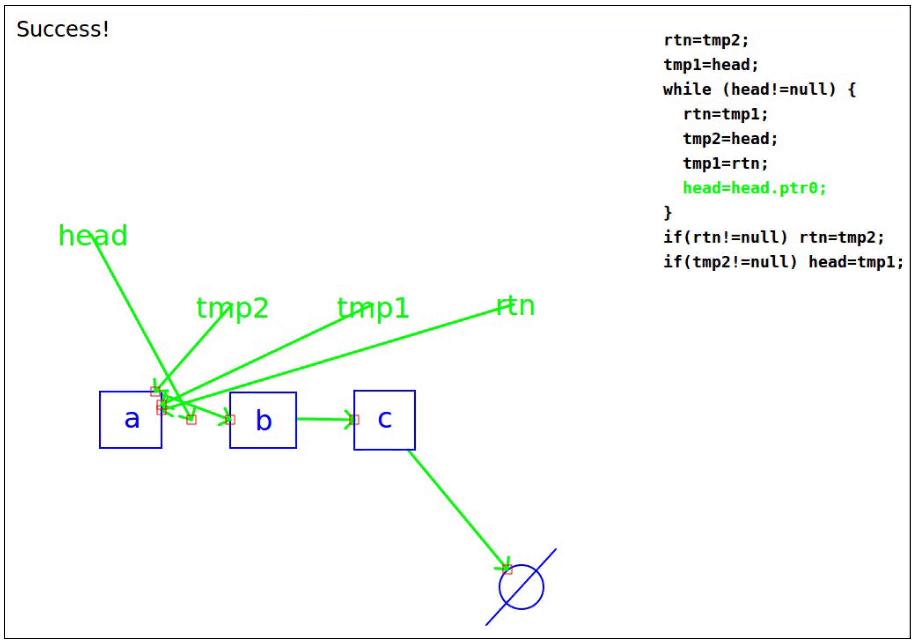
4.3.4 Labeling the Data Structure and Manipulation

As a final step, the user labels this data structure and manipulation by saying, "This is a linked list find-last," which CoMo parses using a simple regular expression. Recall



```
rtn=tmp2;
tmp1=head;
while (head!=null) {
  rtn=tmp1;
  tmp2=head;
  tmp1=rtn;
  head=head.ptr0;
}
if(rtn!=null) rtn=tmp2;
if(tmp2!=null) head=tmp1;
```

Figure 4-16: CoMo animates the new code on the list with four nodes.



```
rtn=tmp2;
tmp1=head;
while (head!=null) {
  rtn=tmp1;
  tmp2=head;
  tmp1=rtn;
  head=head.ptr0;
}
if(rtn!=null) rtn=tmp2;
if(tmp2!=null) head=tmp1;
```

Figure 4-17: CoMo generates a structure with three nodes and animates the code.

that the first question the user asked CoMo was, "Do you know what a linked list is?" CoMo answered "No" because it had not ever encountered one. Now that the user has labeled this data structure and manipulation, the structure will be recalled in future interactions when the user asks this question, and it will recall the code and examples when asked, "Do you know what a linked list find-last is?"

4.4 Additional Functionality

The linked list find-last example above illustrates a subset of CoMo's abilities. This section describes the remaining functionality.

4.4.1 A More Educated CoMo

Imagine that the user had already discussed a singly linked list reversal with CoMo before the above scenario began. When asked whether it knows what a linked list is, CoMo would have said "Yes" and displayed the structure definition from that reversal interaction. This ability speeds up the interaction with CoMo and eliminates one source of user error: describing the data structure. Similarly, the user could have asked "Do you know what a linked list find-last is?" and CoMo would have said "Yes" and brought up the examples and the code.

4.4.2 Labeling Pointers

CoMo recognizes that each node has the same number of the same type of pointers. Each pointer is given a default name[5] so that code synthesis functions correctly, but users can label pointers by drawing a name on them. Labeling pointers has no effect on synthesis other than changing pointers' names in the code. For example, had the user labeled the pointers above by drawing "`next`" over one of them, the code would have contained "`next`" pointers instead of "`ptr0`" pointers, but the example would be otherwise identical.

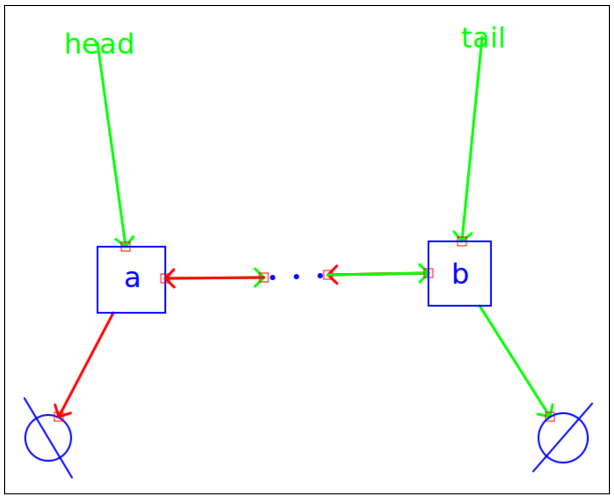
[5]As may be apparent from the example above, that default name is "`ptr`n," where n is the index of that pointer's type's appearance, starting at 0. This is detailed in Chapter 5.

4.4.3 Constraints

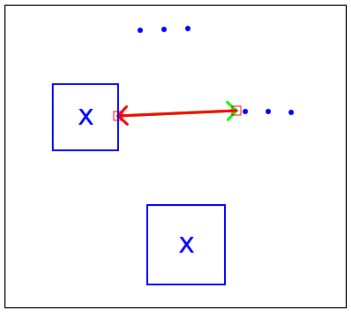
CoMo can discuss ordered data structures by recognizing drawn constraints. For example, to discuss an ordered linked list containing three nodes labeled `a` through `c`, the user can specify that the list is ordered by writing "`a < b < c`" (or the equivalent for any pair of neighboring nodes) on the canvas. This specifies a relationship between the nodes' values that is used both by CoMo and SPT. When performing an ordered linked list insertion, a constraint of this kind is present in the description. CoMo has been programmed with the knowledge that it must ask questions comparing the value of the node in a single-node list with the value of the node to insert. During synthesis, SPT uses these values to synthesize conditions comparing values to one another.

4.4.4 Multiple Pointers

Some structures have multiple pointers. For example, binary trees have `left` and `right` pointers pointing to their left and right children and doubly linked lists have `next` and `prev` pointers pointing to the next and previous node. CoMo is able to distinguish pointers from one another by matching a pointer with the direction it is generally drawn in.



(a) A general doubly linked list.



(b) The ellipsis expansion.

Figure 4-18: A doubly linked list general structure definition.

Fig. 4-18 shows the diagrams drawn to describe a doubly linked list to CoMo. As with the equivalent diagram for a singly linked list, first the user draws the general structure of the list (Fig. 4-18a) and after being asked what the ellipsis represents, they draw an explanation of what the ellipsis represents (Fig. 4-18b). CoMo is able to recognize that the pointer coming from the ellipsis in the ellipsis expansion description is a `prev` pointer, because it compares its direction with the directions of the pointers in the previous drawing, and sees that its direction is more similar to the direction of `prev` pointers than to the direction of `next` pointers.

4.5 Summary

This chapter demonstrated how an interaction with CoMo about a linked list find-last manipulation produced correct, executable code. CoMo and the user interacted by asking each other questions and providing answers to each others' questions. The user queried CoMo's knowledge of a specific data structure, and CoMo asked the user about specific instances of the manipulation they chose to discuss. Finally, code was synthesized. CoMo helped the user determine that the code was incorrect, so they added a clarifying example and code was re-synthesized. The discussion above demonstrated how CoMo interacts about:

1. Querying what kinds of structures CoMo is familiar with.
2. Describing what general structures look like (employing ellipses to denote repetition in structures).
3. Asking questions and interpreting answers about concrete structures.
4. Synthesizing code and verifying its correctness with animations.
5. Revising code by discussing more examples with CoMo.
6. Labeling the data structure and manipulation that was discussed.

This chapter described the features that were not included in the example, namely that CoMo is able to:

1. Recall structures and manipulations by name, thereby speeding up the interaction and removing possible sources of errors: repeating the explanation of the structure or the manipulation.

2. Label pointers by drawing on them, thereby producing code compatible with existing systems.

3. Order structures by interpreting drawn constraints and synthesizing order-aware code.

4. Handle structures with multiple node pointers, like binary trees and doubly linked lists.

The next chapter describes how CoMo guides the interaction with the user, and how it achieves these capabilities.

Chapter 5

CoMo: Functionality

The previous chapter discussed an interaction with CoMo about a linked list find-last operation. This chapter describes the same interaction but focuses on how CoMo determines which questions to ask, how it asks them, and how it interprets their answers. It also describes the Storyboard Programming Tool, the system that CoMo uses to synthesize code from examples.

5.1 Mixed-Initiative Code-Generation Framework

The algorithm CoMo uses to decide what to ask, when to ask it, and how to ask it is a mixed-initiative code-generation framework (MICGF). The framework is mixed-initiative because it allows either the user or the system to initiate an interaction. The MICGF's purpose is getting the information for code synthesis that SPT requires (see §5.2) in a natural, human-like way, as determined by the user study (Chapter 3). At a high level, the MICGF consists of five basic steps:

1. The user initiates the interaction by:
    - asking CoMo if it knows what a specific data structure or manipulation is; or
    - demonstrating a concrete example on a drawn structure; or
    - defining the general data structure to talk about.

2. CoMo asks questions (and interprets answers) about:

    - an inductive general structure definition;
    - a structure with four, one, and zero nodes; or—if the manipulation consists of two structures: a *main* and an *auxiliary* one—the power set of comparisons between the main structure with four, one, and zero nodes and the auxiliary structure with one and zero nodes; and
    - if the manipulation is on an ordered structure that (as above) involves multiple structures (e.g., an ordered linked list insertion)—the power set of constraint comparisons between single-node structures where the main structure's node is less than and greater than the auxiliary structure's node.

3. CoMo uses SPT [28] to synthesize code by:

    (a) generating input based on examples the user drew;

    (b) selecting a control flow graph (CFG) to use;

    (c) running SPT; and

    (d) cleaning code and displaying it to the user.

4. CoMo animates the synthesized code on examples until the user is satisfied it is correct.

5. If the code is incorrect, CoMo reruns synthesis (step 3) with new corrected examples.

CoMo performs these steps while remaining responsive to unexpected input. The MICGF allows CoMo to guide the interaction in such a way that the user will provide enough information to SPT for SPT to produce correct code.

5.2 The Storyboard Programming Tool

Program synthesis is the field of research dedicated to synthesizing a program that meets a set of constraints. In our case, the constraints consist of the state of a data structure as it appears before and after a manipulation. This section discusses the program synthesis tool that CoMo uses, the Storyboard Programming Tool (SPT) [28], and what influence its requirements had on the creation of the MICGF.

5.2.1 Programming with Examples

Code synthesis systems like SPT [14, 17, 31, 18, 33] allow the user to avoid programming and instead simply supply before and after examples, consisting of nodes and pointers. Instead of requiring the user to think of lines of code and the intricacies of memory management (as with traditional programming), users of SPT specify the desired input and output states of a manipulation.

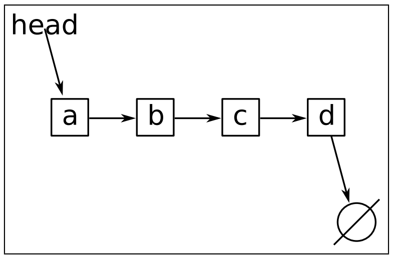
As the following sections demonstrate, SPT requires examples of concrete data structures. Concrete examples are far easier to conceptualize than more abstract, general case examples. For instance, instead of having to specify what a list reversal looks like on an arbitrarily long list. The example below demonstrates a user giving SPT several concrete examples of different lists being reversed, and shows the code that SPT produces as a result.

5.2.2 An Example SPT Run-Through

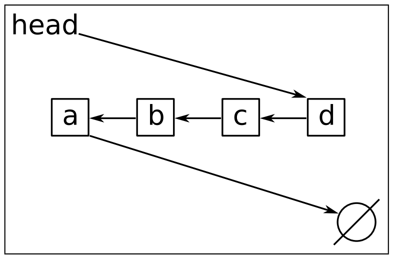
In order for CoMo to use SPT, the details of how SPT works must be considered: what kind of input SPT requires and what kind of output it produces. This section goes through the process of using SPT to synthesize code capable of reversing a singly linked list, and highlights features and limitations that have consequences for CoMo.

A Conceptual Description

To synthesize code for a linked list reversal, SPT must be given concrete examples of a data structure before and after the reversal, and an outline of the general structure of



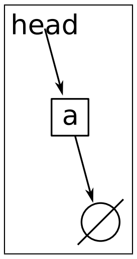
(a) A list with four nodes.



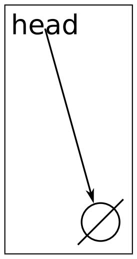
(b) The reversed list.

Figure 5-1: A linked list with four nodes and the same list reversed.

the code to synthesize. Assume the user guesses that three examples will suffice. The first example can be seen in Fig. 5-1, a list with four nodes. SPT is given a description of the list as it appears before (Fig. 5-1a) and after (Fig. 5-1b) the reversal. Being a proficient programmer, the user knows to make sure corner cases are handled, so they select the empty list and the list with only one node (Fig. 5-2) as the other two examples. These lists are identical when reversed, so only one state is shown in the figures.



(a) A list with one node.



(b) An empty list.

Figure 5-2: Linked lists that look the same reversed.

Next, the user must specify the general structure of the code to synthesize. The code structure SPT requires is sufficiently detailed that it often requires the user to know more than they normally would unless they had seen linked list reversal code already; it requires knowing the number of temporary pointers, the overall control structures to be used, and the numbers of lines of code in every location of the control structure. Being a proficient programmer, the user knows that manipulations require temporary pointers to keep track of data, some initialization statements, a set of statements that perform the manipulation (possibly in a loop that iteratively performs it), and some clean-up statements. Since the user does not already have the list-reversing code, they have to guess the answer to each of SPT's required elements. They might guess that at most three temporary pointers, three initialization statements, five statements in a loop, and three clean-up statements suffice to perform a singly linked list reversal. If code synthesis fails, the user can increase these numbers, and if synthesis succeeds, they can attempt to shorten synthesis time and simplify the synthesized code by decreasing the number of statements or temporary pointers.

Information Encoding

```
scenario scenario0
input head -> a, a.next -> b, b.next -> c, c.next -> d, d.next -> null
output head -> d, d.next -> c, c.next -> b, b.next -> a, a.next -> null;

scenario scenario1
input head -> a, a.next -> null
output head -> a, a.next -> null;

scenario scenario2
input head -> null
output head -> null;

 list_reverse(Node head) {
   Node tmp1, tmp2, tmp3;
   ??(3)
   while (**) {
     ??(5)
   }
   ?=(3)
 }
```

Figure 5-3: Example SPT input corresponding to linked list reversal.

As input, SPT requires the examples shown above (Fig. 5-1 and Fig. 5-2) to be written in a textual description language. The input consists of a list of *scenarios* and a *control flow graph* (CFG) that describe the examples and code outline, respectively. Fig. 5-3 shows the three scenarios and the CFG for the list reversal example. Each scenario consists of a label, an input state, and an output state. More succinctly, the Backus-Naur Form (BNF) for a scenario is:

```
scenario <name>
input <assignment> [,<assignment> ...]
output <assignment> [,<assignment> ...] ;
```

The CFG's syntax mimics C's syntax. The only differences arise in specifying placeholders for SPT to synthesize code. The "`??(`n`)`" placeholder specifies that SPT should synthesize n statements, the "`?=(`n`)`" placeholder specifies that SPT should synthesize n statements that are preceded by an `if` statement, and the "`**`" placeholder specifies that SPT should find a conditional expression of the form $x \neq y$, $x = y$, `true` or `false`, where x and y are either temporary variable names, dereferenced variables, or `null`[1].

```
typedef struct {
  Node *next;
} Node;
```

(a) The node definition SPT infers.

```
typedef struct {
  Node *head;
} List;
```

(b) The list definition SPT infers.

Figure 5-4: Information SPT is able to infer from its input.

This input format serves to abstract the user from the code further than may be initially apparent. Neither an explicit node definition (Fig. 5-4a) nor an explicit structure definition (Fig. 5-4b) are needed. The node structure is inferred from the pointers present in the scenarios. In this case, the `next` pointer is referenced in the scenarios and no other pointers are mentioned, so SPT can infer that nodes each have a single `next` pointer. SPT does not need to know about the structure's definition, since it treats the manipulation as a set of node- and pointer-based constraints, not explicitly as a structure.

Synthesized Code

Fig. 5-5 shows SPT's verbatim output synthesized with the input given in Fig. 5-3. Synthesis takes about one minute on a 3.9GHz machine running Ubuntu 13.10 with 16GB of DDR3 RAM[2]. Note that the synthesized code matches the CFG in the input almost exactly. Note also that in this particular case there is a statement

[1]If statements synthesized with the "`?=(`n`)`" may simply synthesize "`true`" or "`false`" as their if statement's condition, thereby making the conditional useless. Similarly, while loops' conditions (synthesized with the "`**`" placeholder) may simply generate "`false`." A statement beginning with "`if(true)`" is equivalent to the same statement without the conditional, and a statement beginning with "`if(false)`" or a block beginning with "`while(false)`" is equivalent to no statement.

[2]Note that SPT does not take advantage of multiple CPUs.

```
void main(){
  tmp2=head;
  tmp2=tmp1;
  tmp2=tmp2;
  while(head != null){
    tmp2=head.next;
    head.next=tmp3;
    head=head;
    tmp3=head;
    head=tmp2;
  }
  if(true) head=tmp1;
  if(true) tmp3=tmp3;
  if(true) head=tmp3;
}
```

Figure 5-5: C code produced by SPT.

without an effect: "`head=head`." The user can take this redundant line as evidence that five lines of code are more than are strictly necessary for this manipulation. Further, all the conditional statements begin with `if(true)`, making the conditionals unnecessary. After experimenting with different, shorter CFGs, the user tries the CFG in Fig. 5-6a. Using this, the code in Fig. 5-6b is synthesized. With fewer lines to synthesize, synthesis takes only ten seconds. This conceptually simple change results in an improved synthesis time and more succinct code.

```
list_reverse(Node head) {
  Node tmp1, tmp2;
  ??(1)
  while (**) {
    ??(4)
  }
}
```

(a) A shortened CFG.

```
void main(){
  tmp2=head;
  while(tmp2 != null){
    head=tmp2;
    tmp2=head.next;
    head.next=tmp1;
    tmp1=head;
  }
}
```

(b) Code produced with the shortened CFG.

Figure 5-6: A shortened CFG and the resulting code.

5.2.3 SPT Limitations & Ramifications for CoMo

SPT is a powerful tool that takes a significant step in bridging the gap between the concept of what a manipulation is and the code that implements it. Even so, it has five significant limitations that CoMo is able to address. This section discusses these limitations and briefly describes how CoMo addresses them.

Dead Code

The first limitation is that SPT often synthesizes redundant statements, i.e., dead code. These result from the randomness in the algorithm SPT employs. Some of this code comes in the form of duplicate lines of code, and some is much more difficult to find. The most straightforward types of redundancies are seen in Fig. 5-5. The middle of the while loop has the line "`head=head`", which has no effect, and the bottom three statements are all preceded by "`if(true)`," which also has no effect. SPT also synthesizes duplicate lines, where the same line of code is repeated in succession or the same variable is assigned twice. For example, the lines:

```
head = temp2;
head = temp2;
```

or the lines:

```
head = temp1;
head = temp2;
```

do the equivalent of assigning `head` to `temp2`. Removing the first assignment in these cases would result in functionally equivalent code. CoMo attempts to remove trivial cases like these without removing seemingly redundant cases like this one:

```
head = head.next;
head = head.next;
```

Even more complex cases exist that make it impossible to remove all dead code; dead code elimination is a form of compiler optimization in the same class as unreachable code elimination and redundant code elimination [3]. CoMo removes simple dead code; for two sequential lines of code, CoMo removes any the first line if the pair meets all of the following conditions:

1. The statements' assignments have the same destination.

2. The statements' assignments have the same condition (or no conditions).

3. The statements' assignments' destination is not a substring of its source (e.g., `x = x.next`).

Interaction

Next, specifying a manipulation in SPT's scenario description language is not as natural as drawing it. Much like computer code, the user must learn SPT's input format and fix syntax errors. For example, forgetting the semicolon at the end of a scenario description will cause SPT to fail. CoMo addresses this by engaging the user in a mixed-initiative, symmetric-multimodal interaction and converting drawn diagrams into SPT's textual input format.

Scenario Selection

The third limitation is that SPT has no way of knowing *which* scenarios are required; it can only find code capable of performing the specified manipulation on the scenarios it was given. Knowing which scenarios to give is left to the user; it is a matter of knowing what the manipulation is and which examples are representative. For example, the example presented in section 5.2.2 is the result of an educated guess on the part of a programmer very familiar with list reversals. A less-experienced user may have initially made incorrect assumptions, synthesized incorrect code, and required more trial and error[3]. CoMo addresses this by asking the user a list of questions in order to attempt to provide a representative set of examples for SPT. These questions are:

1. What do you do with an empty structure?
2. What do you do with a structure with one node?
3. What do you do with a structure with at least four nodes?
4. What do you do with each of the comparisons between pairs of structures where the main structure has four, one, or zero nodes in it and the other structure (or structures) has one or zero nodes[4]?

[3]For example, the user might assume that including only a list with four nodes would suffice. SPT would produce code that reverses only that list and has undefined behavior on any other list.

[4]This requires a manipulation that has multiple structures, e.g., in a stack push the main structure is the stack, and the secondary structure is the node to push onto the stack.

5. What do you do with comparisons between structures with one node each when one structure's node has a value greater than or less than the other's[5]?

Selecting a CFG

Fourth, specifying the CFG accurately requires much a priori knowledge of the structure of the code. Should it include a loop? How many statements should be used? Will they need conditional statements? In the example above the user erred on the side of too many statements and temporary variables in the first synthesis run. After several attempts, the user finds the CFG with the fewest statements and temporary pointers that produces succinct, correct code. In essence, requiring the user to provide the CFG is equivalent to asking the user to bridge half the gap between the concept of the manipulation and the implemented code. CoMo addresses this by automatically selecting a CFG and converting the examples drawn by the user into SPT's input format so that code can be synthesized (see §5.5.2 for a complete description of how CoMo handles CFGs).

Code Verification

The code SPT synthesizes is difficult to understand because it does not seem to follow any kind of underlying logic. It has no comments and seems to be nothing more than a random set of assignments that were put together and found to work. As it turns out, this is exactly the method SPT uses to synthesize code. SPT uses a constraint-based synthesis algorithm [29] using a SAT solver to find *some* lines of code that work in the CFG on the provided scenarios. This is reflected in how difficult it is to follow the logic of the code.

Further, the code SPT synthesizes cannot be compiled for three reasons. First, SPT does not include function signatures or temporary pointer declarations. None of the variables in the output code have been declared in this scope. Second, even if

[5]This question requires the structure to be ordered and it requires that there be multiple structures involved in the manipulation, e.g., an ordered list insertion. SPT cannot handle comparisons between nodes with equal values, so the case where the nodes have equal values is not included.

the variables were declared in scope, the `Node` type has not been declared. Finally, "`true`" and "`false`" are not technically valid C constants.

The final limitation that CoMo is able to address, is that due to how difficult the code is to understand and the fact that it cannot be compiled, verifying the synthesized code is either a tedious manual task involving going through several examples by hand, or requires a test suite to be built and run independently of SPT. CoMo addresses this by animating the synthesized code on examples to aid the user in verifying the code's correctness. While this does not do full automation (the user must still initiate the tests and determine for themselves whether the code is correct), it replaces a tedious manual task with a semi-automated one.

5.2.4 Summary

At a high level, CoMo's primary task can be seen as engaging a human in a natural interaction while collecting the information that SPT needs. The next section describes how the MICGF guided the interaction with the user in the example discussed in Chapter 4.

5.3 Linked List Find-Last

This section describes the linked list find-last interaction from Chapter 4, but focuses on the mechanisms CoMo uses to achieve the interaction.

5.3.1 Describing the Data Structure

The user initiates the interaction by asking CoMo, "Do you know what a linked list is?" CoMo maintains a database of data structures and manipulations that it uses to recall the data structures and manipulations by name. Since this is the first interaction CoMo has had, it cannot find a data structure called "linked list," and so, says "No."

The user draws the general data structure seen in Fig. 5-7. CoMo interprets the

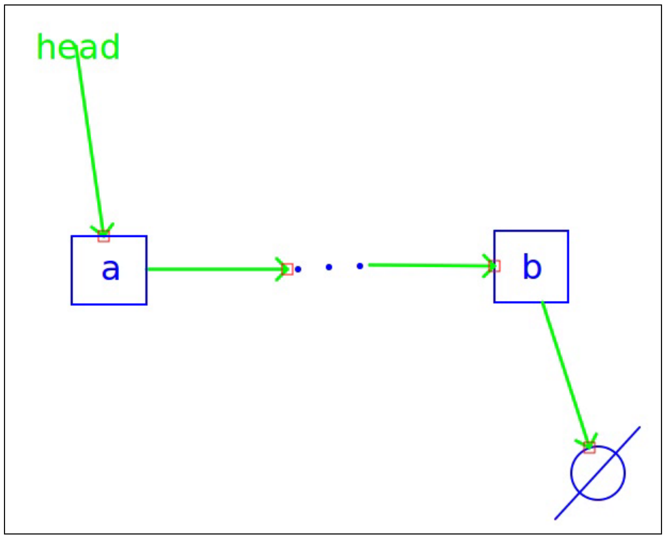
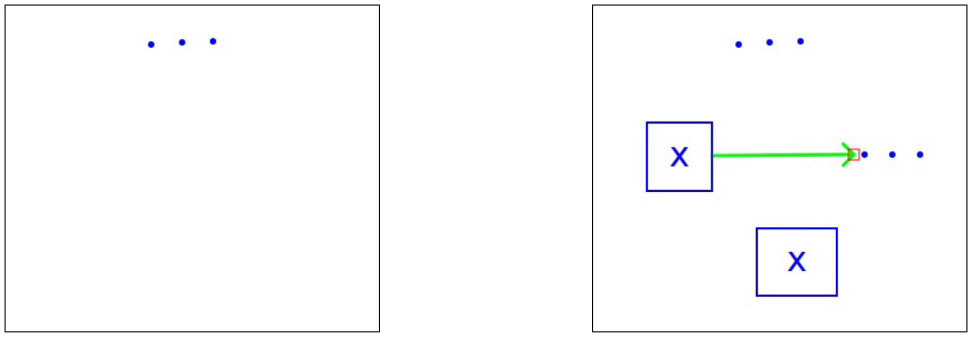


Figure 5-7: The user draws a general list.



(a) CoMo: “What does *this* expand to?”

(b) User: “To *this* or *this*.”

Figure 5-8: The question and answer about the inductive definition of the ellipsis.

presence of an ellipsis to mean that this is a general structure (as opposed to an instance of a data structure). It does not have a definition for the ellipsis in this structure, so its first question is, "What does this expand to?" as it displays a copy of the ellipsis at the top of a new canvas, seen in Fig. 5-8a. CoMo expects the user to answer the question by drawing options in the space below the ellipsis. Fig. 5-8b shows what the user drew in response.

Sketch Generation

CoMo uses the drawings in Fig. 5-7 and Fig. 5-8 to generate lists of arbitrary length. It parses the ellipsis expansion sketch by modeling the drawing as a graph and doing a depth-first search through the elements' pointers (ignoring the ellipsis it drew at the top). As discussed in §5.5.1, each node and pointer in the sketch is grouped based on connectivity—in this case into two groups—and each group is treated as one option for expansion. CoMo classifies groups by how many nodes and ellipses they have; the group with only a single node is classified as a base case expansion, meaning that it will always be the final expansion when generating a structure.

Structure generation proceeds by starting with the general case (Fig. 5-7) and iteratively replacing ellipses with one of the options (Fig. 5-8) until the desired number of nodes has been reached. In the example above, the two options that CoMo parses are the option where an ellipsis can be replaced with a node pointing to a new ellipsis, or just a single node. Since the general list has two nodes and one ellipsis (Fig. 5-7) and the smallest expansion has just a single node (Fig. 5-8), CoMo computes that the smallest structure that can be generated with these drawings has three nodes. Asking CoMo to generate a list with two nodes would result in CoMo saying, "I can't do that." Had the user decided to draw the general list (Fig. 5-7) without the `a` or `b` nodes and instead drawn a `head` pointer pointing directly at the ellipsis and the ellipsis pointing directly at the null symbol, then any nonempty list could be generated with the definitions. Since the user did not draw in this way, discussing an example involving two nodes would require the user to draw it[6].

[6]Future versions of CoMo could also include the ability to ask questions of the form: "Can you

Generating structures with zero or one nodes proceeds differently, since it is a much simpler task and does not require a structure definition. This process starts by finding all the common pointers in the drawn examples (in this case there is only a `head` pointer) and cloning[7] them onto a new canvas. Next, a null symbol from an existing example is cloned and placed on the new canvas. If the empty structure is being generated, then all pointers (just `head`, in this case) are directed toward the null symbol, and the generation process is done. If the structure with one node is being generated, a node from another canvas is cloned and placed onto the new canvas. The pointers are assigned to the new node, all of the node's pointers are assigned to null, and the sketch generation process is done.

5.3.2 Asking an Example Question

To determine its next question, CoMo goes through the list of question types described in §5.2.3. Currently, the user has indicated that there is a structure with one pointer, labeled "`head`," so applying the rules yields the following questions:

1. How do you handle a list with four nodes?

2. How do you handle a list with one node?

3. How do you handle an empty list?

CoMo starts with the first question about a list with four nodes. As with all questions about how the manipulation proceeds on a specific example, CoMo expects the user to do one of three things in response:

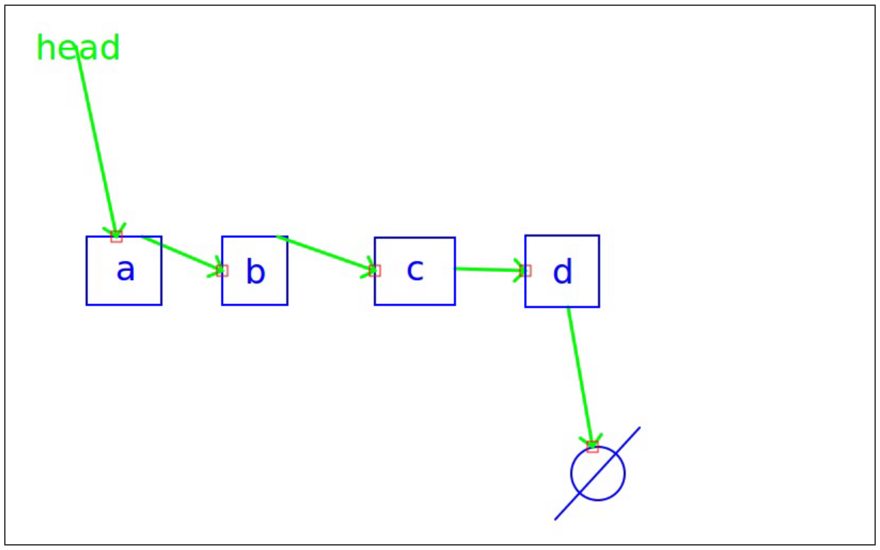
1. Demonstrate how the manipulation is performed (possibly with additions to the sketch) and saying, "Like this."

2. Say, "Do nothing," denoting that the input and output states in this example are identical.

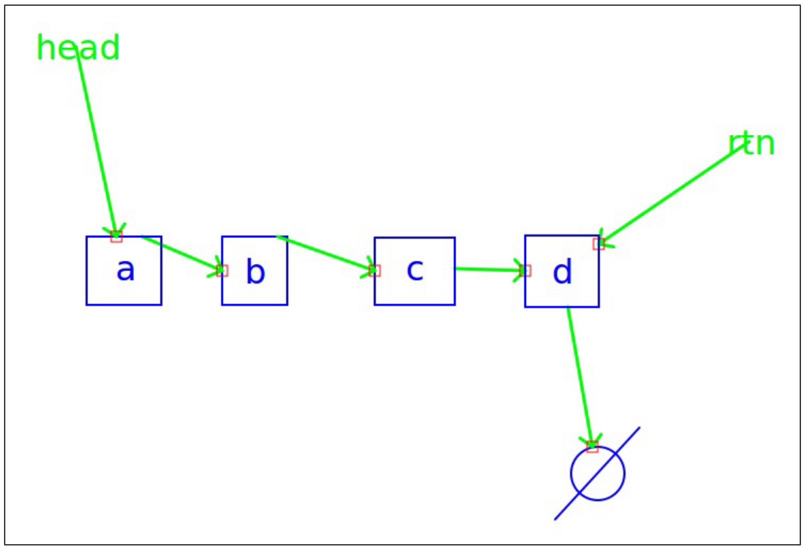
draw me an example with two nodes?" This would allow CoMo to ask questions about structures it cannot generate.

[7]Nodes and pointers are cloned (as opposed to being created from a template) to more closely mimic the user's drawing style. Notice that the list's nodes in Fig. 5-9a have slightly differing sizes; these are the sizes that the user drew the nodes in Fig. 5-7 and Fig. 5-8.

3. Say, “That does not make sense,” denoting that this example should not be included in SPT’s input.



(a) CoMo: “How do you handle *this* case?”



(b) The user creates the `rtn` pointer, points it at null, reassigns it to `d`, and says, “Like this.”

Figure 5-9: The first questions (Fig. 5-9a) that CoMo asks and the user’s response (Fig. 5-9b).

Fig. 5-9 shows the question that CoMo asks and the answer that the user gives. As part of the answer, the user adds a new pointer: `rtn`. CoMo interprets the new pointer to be a new, auxiliary structure because (unlike `head`) it is not mentioned in the general structure. CoMo modifies its list of question-generating cases to be:

- a list with four nodes
- a list with one node
- an empty list

Crossed with:

- `rtn` pointing at one node
- `rtn` pointing at null

More concretely, the questions CoMo asks are:

1. How do you handle the case where the list has four nodes and `rtn` points at one node?
2. How do you handle the case where the list has four nodes and `rtn` points at null?
3. How do you handle the case where the list has one node and `rtn` points at a separate node?
4. How do you handle the case where the list has one node and `rtn` points at null?
5. How do you handle the case where the list is empty and `rtn` points at a node?
6. How do you handle the case where the list is empty and `rtn` points at null?

The distinction between a structure and an auxiliary structure accounts for why the above list of questions does not include the case where `rtn` points to its own list.

5.3.3 Nonsensical Questions

Three questions from the above list involve `rtn` pointing at its own node and `head` pointing to a list with four nodes (Fig. 5-10), one node (Fig. 5-11), and no nodes (Fig. 5-12). These nonsensical questions are the result of CoMo interpreting the `rtn` pointer to be a completely separate structure, instead of a pointer indicating which node is to be returned. The user answers all of these questions identically: by saying, "That does not make sense." CoMo displays the utterance in the bottom-left corner, marks these questions as nonsensical, and proceeds.

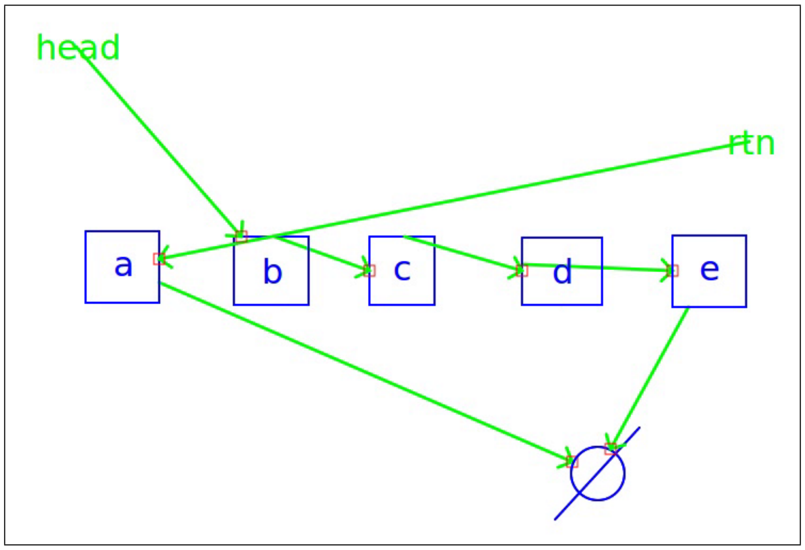


Figure 5-10: CoMo: “How do you handle *this* case?” User: “That doesn’t make sense.”

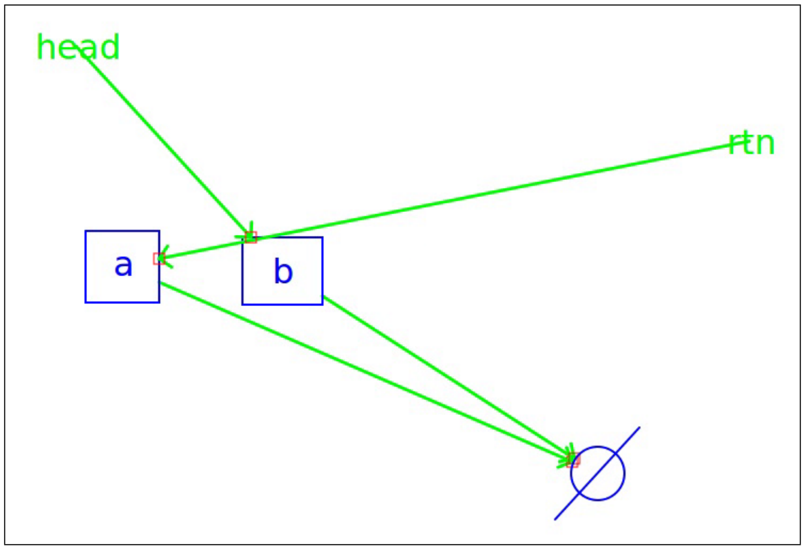


Figure 5-11: CoMo: “How do you handle *this* case?” User: “That doesn’t make sense.”

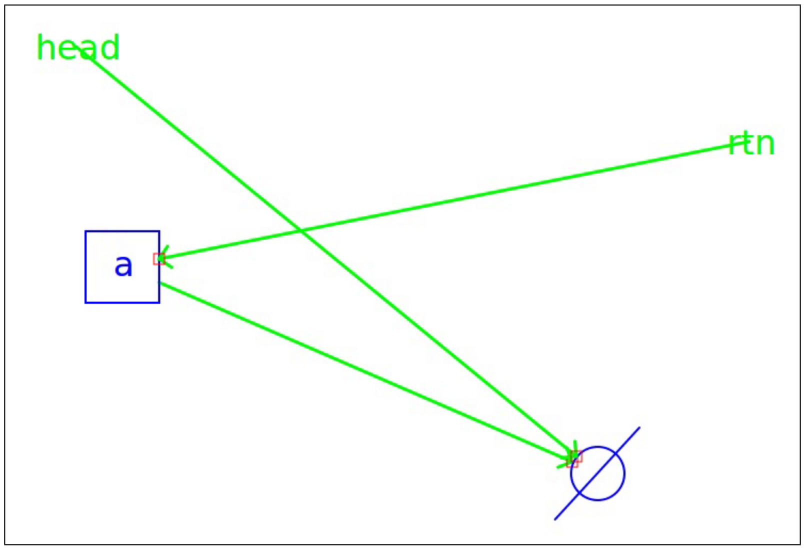
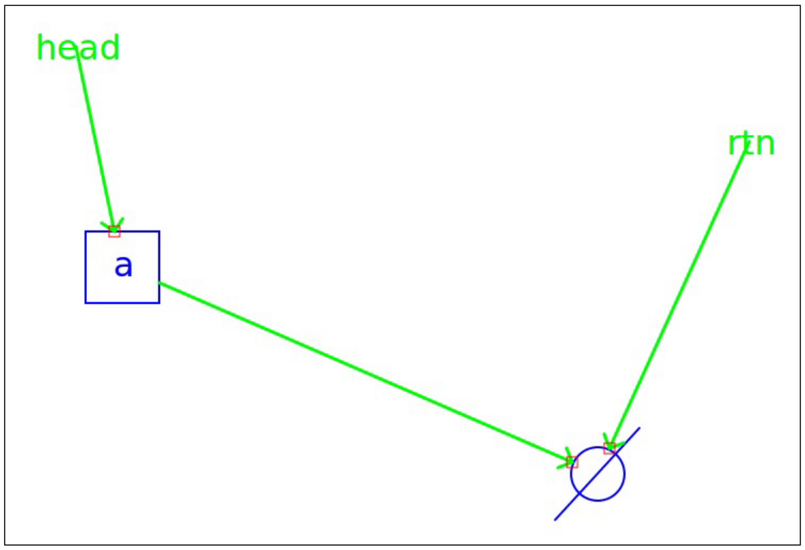


Figure 5-12: CoMo: "How do you handle *this* case?" User: "That doesn't make sense."

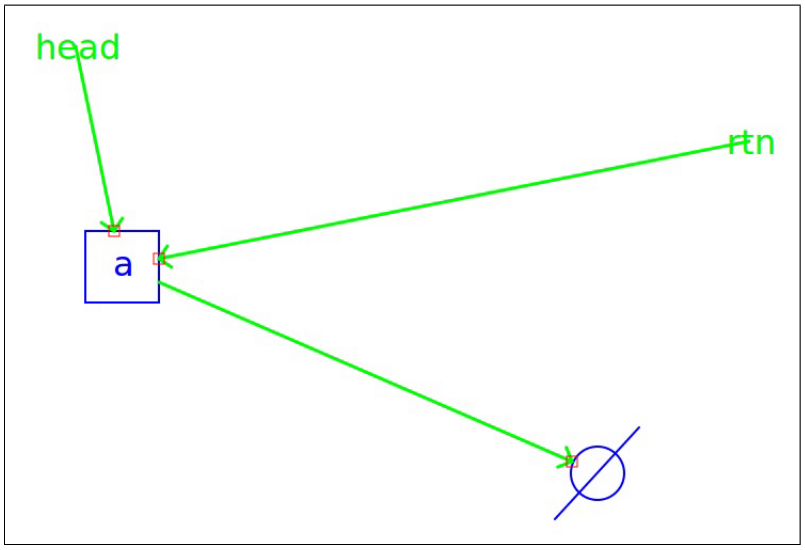
5.3.4 Remaining Questions

The final two questions that CoMo asks involve the `rtn` pointer starting out pointing at null, and the list containing one node (Fig. 5-13) and no nodes (Fig. 5-14). CoMo does not ask about the case where the list has four nodes and `rtn` points at null, because that question was already answered by the user. Despite CoMo asking the more general question about what to do with a list with four nodes, the user answered what to do with a list with four nodes and a `rtn` pointer pointing at null. CoMo did not know about the `rtn` pointer when asking the question, but after reviewing the answer to the question, it recognized that the example represented the case with a list with four nodes and a `rtn` pointer pointing at no nodes, so it did not ask about the case again.

The user answers the question about the list with one node by clicking and dragging the `rtn` pointer to the `a` node and saying, "Like this." (Fig. 5-13b) As with the first question, CoMo takes note of the before and after states, and asks the final question, which the user answers by saying, "Do nothing." CoMo notes that the final example has identical input and output states. At this point, CoMo has asked all its questions, and thus code synthesis can be attempted. CoMo signals this to the user



(a) CoMo: “How do you handle *this* case?”



(b) The user moves the `rtn` pointer to point at the `a` node, and says, “Like this.”

Figure 5-13: The question and answer about a list with one node.

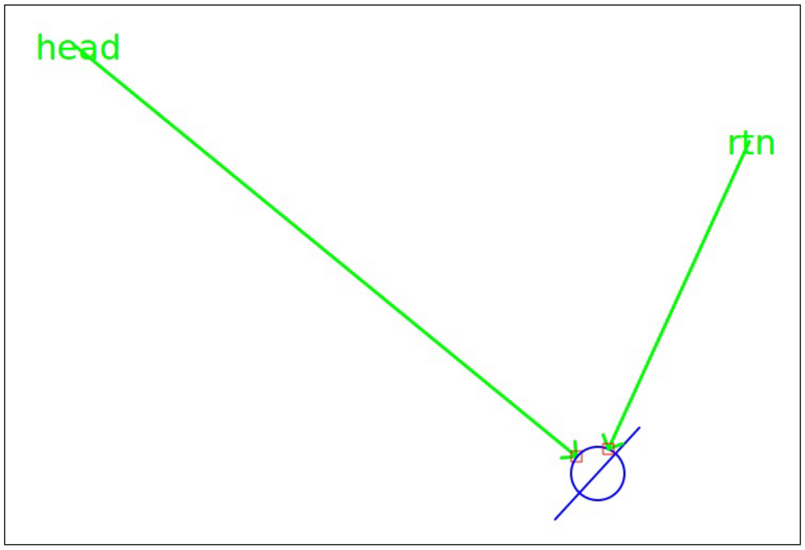


Figure 5-14: A question and the corresponding answer about an empty list. CoMo: "How do you handle *this* case?" User: "Do nothing."

by saying, "I think I understand" and awaits further input. Just because CoMo is satisfied with the interaction does not necessarily mean that the user is. The user can add additional examples by saying, "Make a new canvas" and drawing another example or having CoMo generate an example by saying, e.g., "Generate a structure with five nodes." The user can add as many examples as they feel are representative of the manipulation before initiating code synthesis.

5.3.5 Code Synthesis & Verification

To synthesize code, CoMo converts each of the examples into a scenario. Fig. 5-15 shows the SPT input generated from the examples shown above. CoMo names the scenarios based on when they appeared in the discussion. Since the first and second canvases in this discussion described the general structure of the list, there is no `scenario0` or `scenario1`; the first scenario is labeled `scenario2` and corresponds to the question about the list with four nodes. Scenarios 3–5 represent the nonsensical examples CoMo presented, so they do not appear in the input either. The next scenarios are `scenario6` and `scenario7`, which correspond to the lists with one and zero nodes, respectively.

```
scenario scenario2
input head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      rtn -> null
output head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      rtn -> d ;

scenario scenario6
input rtn -> null, head -> a, a.ptr0 -> null
output rtn -> a, head -> a, a.ptr0 -> null ;

scenario scenario7
input rtn -> null, head -> null
output rtn -> null, head -> null ;

main(Node head, Node rtn) {
  Node tmp1, tmp2;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 5-15: The input CoMo produces for SPT.

```
main(/*pointers*/) {
  /*tmps*/
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 5-16: The default CFG used by CoMo. §5.5.2 explains how this was selected and how it can be altered.

CoMo starts with the default CFG seen in Fig. 5-16 that consists of three statements followed by a while loop with five statements in its body and ending with three statements preceded by if statements. It inserts all the pointers in the function signature and any temporary variables below it. This CFG was selected as the default because it is general enough to work with most of CoMo's manipulations (see §5.5.2). CoMo knows that manipulation implementations usually require temporary pointers (e.g., to avoid dropped nodes) so if the user did not add any temporary pointers to the examples, CoMo adds two. The CFG in Fig. 5-15 reflects this addition. The first line in the body of the CFG is "`Node tmp1, tmp2;`" This line informs SPT that it has these two pointers at its disposal in the code. Since the states of `tmp1` and `tmp2` are not specified in any of the scenarios, SPT will not pay attention to their starting or ending states; it will assume that they start out pointing to null and will place no constraints on where they end.

```
void main() {
  rtn=head;
  head=tmp1;
  tmp1=rtn;
  while(rtn!=head){
    rtn=head;
    rtn=rtn;
    rtn=head;
    rtn=tmp1;
    head=rtn;
  }
    if(tmp1!=null) tmp2=tmp1.ptr0;
    if(tmp2!=null) tmp2=tmp2.ptr0;
    if(tmp2!=null) rtn=tmp2.ptr0;
}
```

(a) Raw output.

```
rtn=head;
head=tmp1;
tmp1=rtn;
while(rtn!=head){
  rtn=tmp1;
  head=rtn;
}
if(tmp1!=null) tmp2=tmp1.ptr0;
if(tmp2!=null) tmp2=tmp2.ptr0;
if(tmp2!=null) rtn=tmp2.ptr0;
```

(b) Cleaned code.

Figure 5-17: SPT's output before and after CoMo removes dead code.

After approximately 10 seconds, SPT produces the output seen in Fig. 5-17a. CoMo removes simple dead code and displays the result (Fig. 5-17b) on the canvases.

Animation

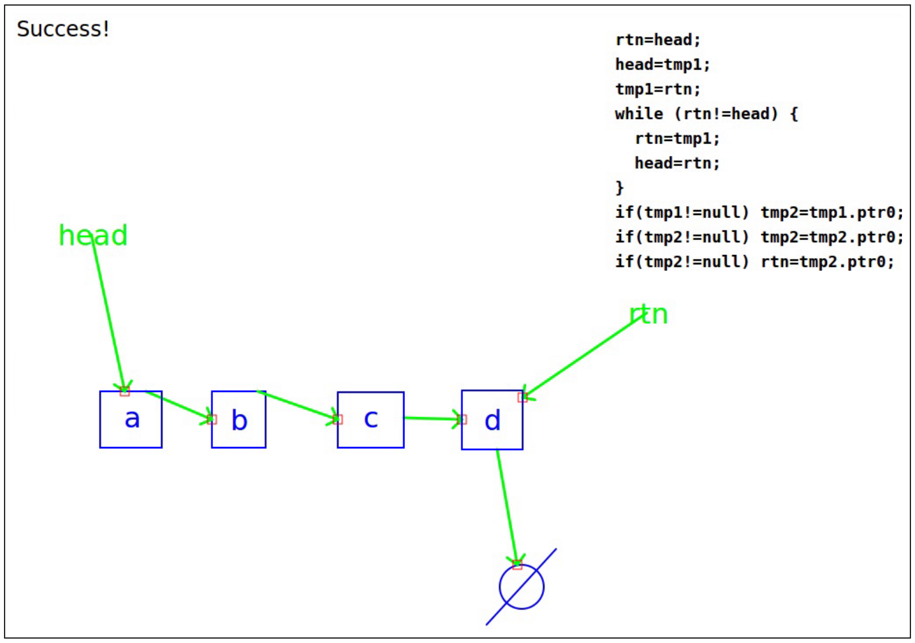
CoMo animates code by:

1. Interpreting the code with a simple parser and interpreter, capable of handling

the simple kinds of statements, conditionals, and logic constructs that appear in SPT's output.

2. Relating the variables that appear in the statements to the drawn elements in the sketch (first producing those variables if they do not already appear in the sketch).

3. Animating a smooth transition for each pointer from its current position to the target specified by each line of code.

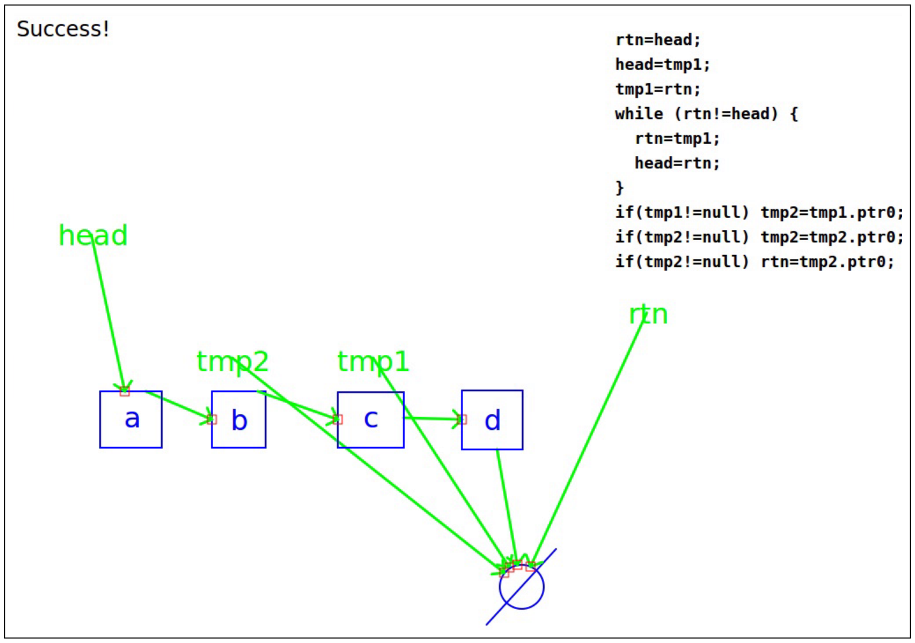


```
rtn=head;
head=tmp1;
tmp1=rtn;
while (rtn!=head) {
  rtn=tmp1;
  head=rtn;
}
if(tmp1!=null) tmp2=tmp1.ptr0;
if(tmp2!=null) tmp2=tmp2.ptr0;
if(tmp2!=null) rtn=tmp2.ptr0;
```

Figure 5-18: This sketch is missing `tmp1` and `tmp2`.

Fig. 5-18 shows the first example—the list with four nodes—after synthesis completes and CoMo displays the code next to the example. Before CoMo begins animating the synthesized code in Fig. 5-17b, it notices that the two temporary pointers "`tmp1`" and "`tmp2`" are not present in any of the example sketches. It creates these pointers and initializes them to point at null[8] (Fig. 5-19). Once these pointers are added, CoMo begins the animation process. At each line of code, CoMo finds the referenced pointer and its new target by name and, in a smooth animation, transitions

[8]The `tmp1` and `tmp2` pointers are initialized to null because SPT assumes that any variable not mentioned in a scenario was initialized to null.

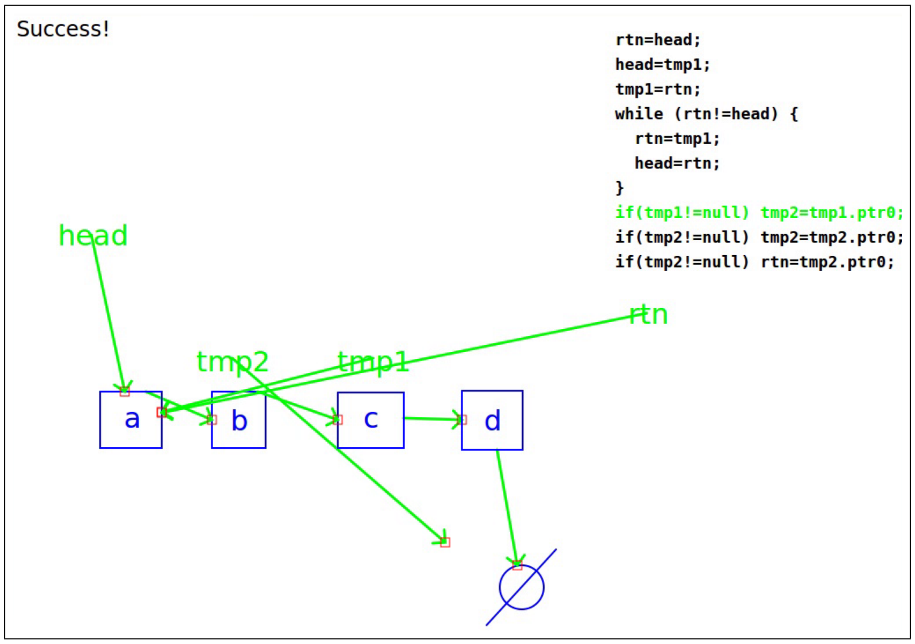


```
rtn=head;
head=tmp1;
tmp1=rtn;
while (rtn!=head) {
  rtn=tmp1;
  head=rtn;
}
if(tmp1!=null) tmp2=tmp1.ptr0;
if(tmp2!=null) tmp2=tmp2.ptr0;
if(tmp2!=null) rtn=tmp2.ptr0;
```

Figure 5-19: CoMo adds `tmp1` and `tmp2`, initializes them to null, and resets the structure to its starting state in preparation for code animation.

the pointer from its current target to its new target.

Fig. 5-20 shows CoMo in the process of animating the line "`if(tmp1!=null) tmp2=tmp1.ptr0`." Evaluating this line consists of two phases: determining what the conditional evaluates to (`true` or `false`), and executing the line if the conditional evaluates to `true`. To evaluate the conditional `tmp1!=null`, CoMo finds the `tmp1` pointer and determines what its target is. In this case, its target is the `a` node, which is not null, so the conditional evaluates to true. Next, CoMo finds the pointers and targets needed to execute the line of code. It finds that `tmp1` is pointing to `a`, and follows `a`'s `ptr0` to find its target: `b`. It finds the `tmp2` pointer and computes the trajectory from where the pointer currently points (null) to the center of the `b` node. It animates the transition from the null symbol to the `b` node by moving the pointer's head 10px every frame of animation at 15Hz; Fig. 5-20 shows `tmp2` midway through the transition between pointing to null and pointing to `b`. Once the pointer has reached its new target, CoMo assigns the pointer to it (resulting in the pointer snapping to its new target), and moves on to the next line of code.


Success!

```
rtn=head;
head=tmp1;
tmp1=rtn;
while (rtn!=head) {
  rtn=tmp1;
  head=rtn;
}
if(tmp1!=null) tmp2=tmp1.ptr0;
if(tmp2!=null) tmp2=tmp2.ptr0;
if(tmp2!=null) rtn=tmp2.ptr0;
```

Figure 5-20: CoMo animates the code on the initial example with four nodes.

```
scenario scenario2
input head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      rtn -> null
output head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      rtn -> d ;

scenario scenario6
input rtn -> null, head -> a, a.ptr0 -> null
output rtn -> a, head -> a, a.ptr0 -> null ;

scenario scenario7
input rtn -> null, head -> null
output rtn -> null, head -> null ;

scenario scenario8
input head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d,
      d.ptr0 -> e, e.ptr0 -> null, rtn -> null
output head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d,
      d.ptr0 -> e, e.ptr0 -> null, rtn -> d ;

main(Node head, Node rtn) {
  Node tmp1, tmp2;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 5-21: The augmented input CoMo produces for SPT.

As described in Chapter 4, the user has CoMo animate the code on the lists with four and five nodes and discovers that the code does not work properly. After correcting the example with five nodes (see Fig. 4-14), they initiate code synthesis as before, by saying, “Generate code.” As seen in Fig. 5-21, CoMo adds the new scenario, `scenario8`, to the existing SPT input. Notice that the input is identical to the previous input, except a new scenario has been added (`scenario8`) representing the example with five nodes.

```
void main() {
  rtn=head;
  rtn=tmp2;
  tmp1=head;
  while(head!=null){
    rtn=tmp1;
    tmp2=head;
    head=head;
    tmp1=rtn;
    head=head.ptr0;
  }
    if(false) tmp2=head;
    if(rtn!=null) rtn=tmp2;
    if(tmp2!=null) head=tmp1;
}
```

(a) Raw output.

```
rtn=tmp2;
tmp1=head;
while(head!=null){
  rtn=tmp1;
  tmp2=head;
  tmp1=rtn;
  head=head.ptr0;
}
if(rtn!=null) rtn=tmp2;
if(tmp2!=null) head=tmp1;
```

(b) Cleaned output.

Figure 5-22: SPT’s output before and after CoMo removes dead code.

With the new scenario, synthesis takes approximately one minute, and produces the code seen in Fig. 5-22a, which is cleaned and presented to the user as the code seen in Fig. 5-22b.

5.3.6 Labeling the Interaction

Finally, CoMo labels the interaction by parsing the user’s utterance, “This is a linked list find-last.” CoMo knows to label the structure as “linked list” and the manipulation as “find-last” because it has a list of these words, labeled as either data structure names or manipulation names, in its speech recognition grammar (discussed below). As described in Chapter 4, the user verifies this newly synthesized code by testing it on lists with three, four, and five nodes, and the interaction is complete.

5.4 System Overview

This section describes how CoMo is implemented. It breaks CoMo down into the different subparts responsible for speech and sketch recognition and generation, interaction management, and question generation. Further, it demonstrates the flexibility of the mixed-initiative code-generation framework by including a state transition diagram showing how the user and CoMo structure the interaction together.

5.4.1 System Architecture

Fig. 5-23 shows CoMo's functional architecture—how all its parts work together to implement a fluid symmetric multimodal interaction with the user.

The user draws strokes on the canvas (at the top-left of the figure) that are interpreted by the sketch recognition module. As described fully in §5.4.3, strokes are first classified as either gestures or non-gestures. Non-gestures are interpreted as parts of the data structure being drawn by the sketch recognition pipeline. These non-gesture strokes collectively form the structure begin discussed. Gestures, on the other hand, are passed directly to this underlying drawn data structure to update its configuration. The only kind of gesture CoMo recognizes are pointer manipulations

Speech (seen at the top-right of the figure) is interpreted by the Sphinx speech recognizer [37, 32]. As described fully in §5.4.4, Sphinx uses a speech grammar to determine which utterances are to be recognized. The utterance and the drawn data structures are fed directly into the Interaction Manager—the module responsible for implementing the MICGF.

Section 5.4.2 describes how the Interaction Manager reacts to user input. Depending on the context of the interaction, the Interaction Manager issues four kinds of commands: a sketch generation command, a speech generation command, a code generation command, and a code playback command.

Sketch generation (as mentioned above) comes in three flavors: generating an example to ask the user to perform a manipulation on, generating a canvas to ask how the repetition symbol expands, and generating an empty canvas. The Interaction

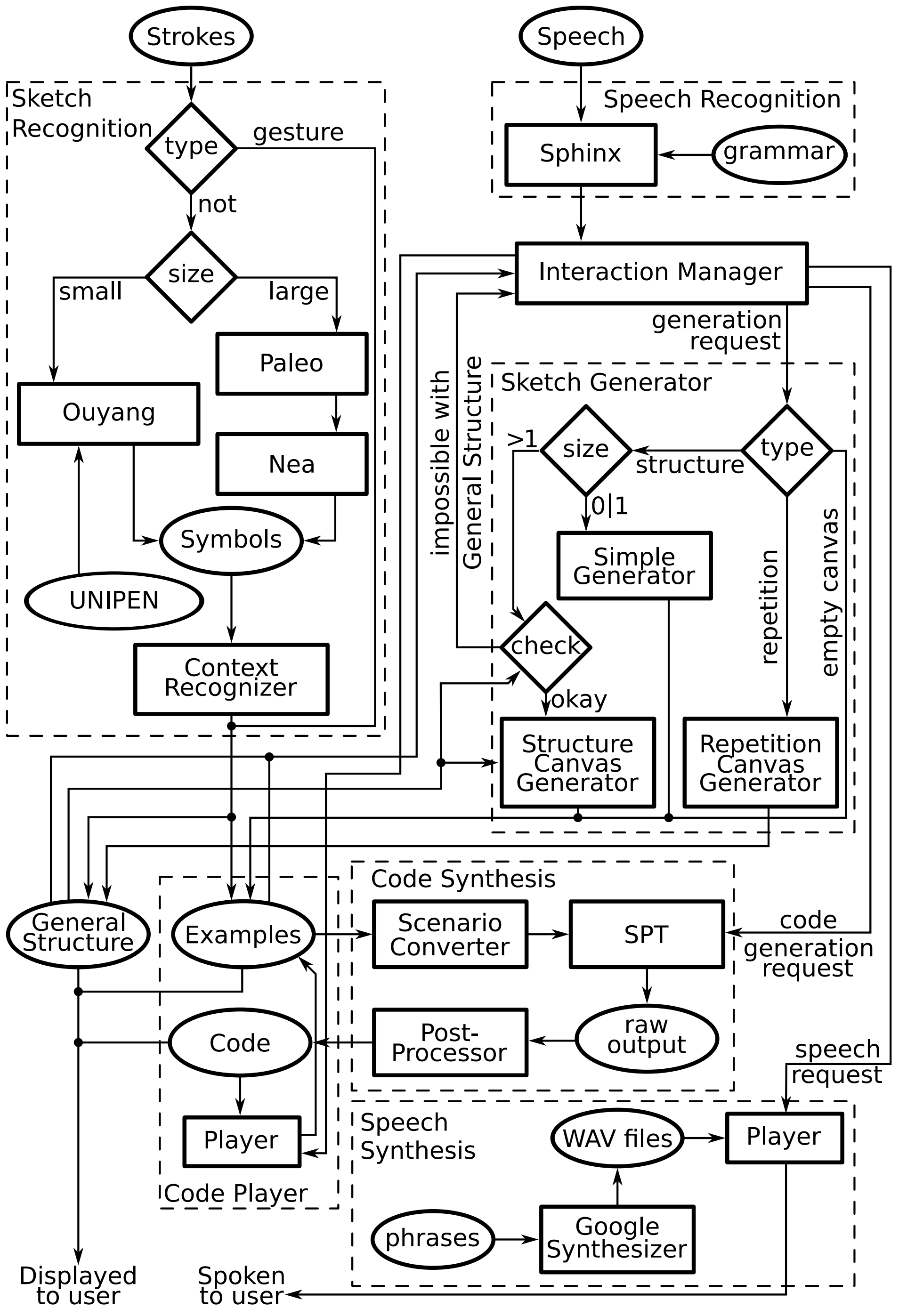


Figure 5-23: An overview of CoMo's architecture. Input is displayed at the top and output at the bottom.

Manager sends these kinds of commands to the Sketch Generator. The repetition canvas is generated by copying a repetition symbol (an ellipsis or a triangle) onto the top-center of a new canvas. As described above, there are two distinct methods of generating an example structure. The simpler of the two (used for generating structures with 0 or 1 nodes) involves copying pointers, nodes, and null symbols from other canvases while the more complex (for generating larger structures) involves using the inductive general structure definition. If the larger general structure definition does not allow creating the structure with the specified number of nodes, the Sketch Generator informs the Interaction Manager. Otherwise, the new canvas is generated and added to the existing examples.

Speech commands generated by the Interaction Manager are passed to the Speech Synthesis module. As explained in §5.4.5, this module plays pre-made WAV files generated with Google's speech synthesizer. This module associates pre-determined string labels with audio files that contain voice data and plays the files on command.

The final module in the system diagram is the Code Synthesis module, responsible for interfacing with SPT [28]. As described above, this module converts the drawn examples into SPT's scenario-based input format, selects a CFG, and runs SPT. Once SPT completes, this module removes dead code and displays it to the user. The Code Player can then play through the code when commanded to do so by the Interaction Manager.

5.4.2 Interface Transition

This section describes how the interaction manager from Fig. 5-23 guides the interaction while remaining responsive to the user's input.

Fig. 5-24 shows a state-transition diagram describing how the Interaction Manager reacts to input. The state diagram starts at the right; CoMo starts in an idle mode, awaiting user input to initiate the interaction. The three ways that the user can initiate the interaction are:

- asking CoMo if it knows what a specific data structure or manipulation is;

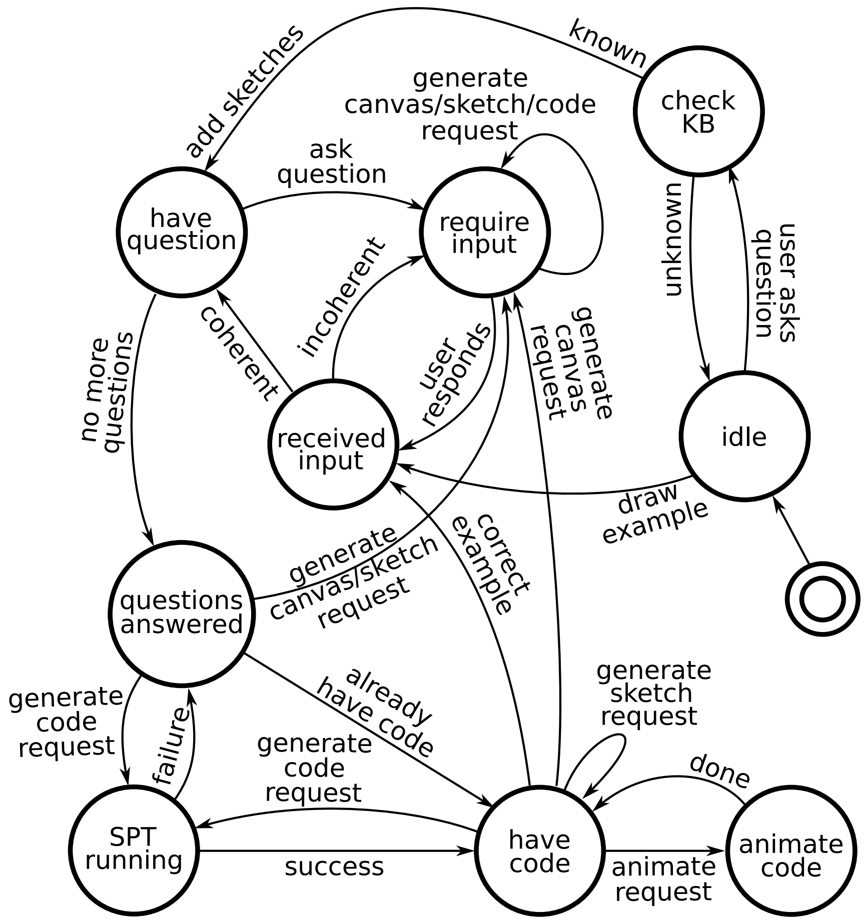


Figure 5-24: The state transition diagram describing how CoMo reacts to user input.

- demonstrating a concrete example on a drawn structure; or
- defining the general data structure to talk about.

If the user simply starts drawing a sketch—i.e., defining the general structure or demonstrating a concrete example on a drawn structure—then, after a three second pause (this heuristic is discussed in §5.5.3), CoMo parses the input and determines whether the user has given a coherent diagram or not. Coherency is determined by two things: whether all the nodes in the diagram are of the same type and whether all the pointers in an example exist in all examples (for the general definition, only the main structure's pointers must be present). If the diagram is incoherent, CoMo requires more input before it can try parsing the sketch again. If the input is coherent, then the question loop can begin in the "have question" state.

The other way to start a conversation with CoMo (seen in the example above), is by asking CoMo whether it knows about a specific data structure or manipulation. When asked, it queries its knowledge base—abbreviated in the diagram as "KB"—the set of data structures and manipulations, referenced by name, that CoMo has previously encountered[9]. If the KB does not contain the data structure or manipulation, CoMo returns to its idle state. If the KB does contain the data structure or manipulation, it adds the sketches associated with the data structure or manipulation and begins the question loop.

The question loop starts in the state labeled "have question." CoMo selects a question based on a question order heuristic (discussed below) and asks it. Having asked a question, it enters the state where it requires input in order to continue. Once the user gives input, CoMo waits three seconds before attempting to parse it. The parsing process proceeds identically as when coming from the idle state. One of the inputs that the user may provide in the "require input" state is a command to generate a new canvas, a sketch, or code. If the user tries to initiate code generation from this state, CoMo will respond "I have another question first." If the user says,

[9]Note that mention of this knowledge base is omitted from the system architecture diagram (Fig. 5-23) as it adds clutter without adding meaningful information. If it were in the diagram, it would be a sub-part of the Interaction Manager module.

"Generate a structure with __ nodes" (or "Make a new canvas."), CoMo creates the sketch (or canvas) and waits for input for both the sketch (or canvas) and the original question.

Once all of CoMo's questions have been answered, it transitions to the "questions answered" state. At this point in the interaction, the user generate more canvases and sketches to demonstrate additional examples to CoMo, in which case CoMo reenters the input parsing loop. If, however, the user is satisfied with the discussed examples, they can say, "Generate code" and CoMo will run SPT. While SPT runs, CoMo does not allow the user to update any of the current examples or provide new ones. If SPT complete successfully, then CoMo transitions to "have code," if it fails, then CoMo returns to "questions answered[10]."

Once SPT completes successfully, CoMo transitions to the "have code" state. In this state, CoMo can accept all three generation requests from the user. For example, it can generate a new sketch or a new canvas to demonstrate a new example on. The user can initiate code animation by saying, "Animate the code." During code animation (as with code synthesis), CoMo does not allow user input. Code animation must complete before the user can interact with CoMo again. If the code does not animate properly (e.g., as above when the code failed to find the last element of a five-element list), the user can correct the example. This will cause the user's correction to be parsed in the "received input" state, fall through the "have question" state (as there are no more questions), and fall through the "questions answered" state (as CoMo already has code) to return to the "has code" state. It is left to the user to determine that code synthesis must be retried. The user can engage in this loop until they are satisfied with the code, at which point the interaction is complete.

[10]SPT fails when there is no code that performs the manipulations on the provided examples. At this point, it is left to the user to determine what is inconsistent in the discussed sketches, and reattempt code synthesis.

5.4.3 Sketch Recognition

As demonstrated above, sketching is used both to compose and to manipulate data structures. As the primary mode of input, sketches are recognized as drawn data structures and gestures are used to edit the structure by clicking and dragging pointers. CoMo displays pointers with small red boxes at their tips. These boxes are handles that can be clicked and dragged to reassign pointers, thereby either updating the process of the manipulation or simply reassigning pointers to configure the data structure in its final state.

As discussed in the user study (Chapter 3), different users conceptualized manipulations either as input and output state pairs or as a sequence of steps by working through the details of memory management with temporary pointers. CoMo allows both of these styles. Each intermediate step is recorded along with the beginning and end states. This leaves the possibility of providing SPT with all of these states—potentially improving synthesis time—despite it currently requiring only the beginning and end states (see Chapter 7 for a discussion of this possible improvement).

Sketch recognition proceeds in two parts: shape recognition and symbol recognition. Shape recognition is the process of classifying a sequential set of strokes as lines, boxes, circles, arrows, and characters. CoMo employs two distinct shape recognizers in parallel. If a stroke is sufficiently small[11] it is processed by a shape recognizer developed by Ouyang and Davis [25]. Ouyang's recognizer was trained with a subset of the UNIPEN [35, 15] dataset, a publicly available dataset of hand-drawn symbols. After training, Ouyang's recognizer is used to recognize all upper- and lower-case letters, all numbers, and the > and < symbols. If a stroke's bounding box is sufficiently large, CoMo performs shape recognition with a limited reimplementation of PaleoSketch [7] to classify a stroke as a single-stroke shape, and then compose the shapes with its own shape-composition module, called NeaSketch[12]. At the end of this first stage of sketch recognition, strokes are recognized by either Ouyang's recognizer or by a

[11] "Sufficiently small" entails a stroke's bounding box being smaller than 40 pixels wide by 60 pixels high.

[12] I chose the name "NeaSketch" because "nea" means "new" like "paleo" means "old." "NeaSketch" seemed a fitting name for PaleoSketch's helper.

combination of PaleoSketch and NeaSketch.

Name		**Description**
Label	head	one or more letters
Constraint	a < b < c	text including the > or < symbol
Null	∅	a circle with a line through it
Concrete Node	$\boxed{a}$ ⓐ	a box or circle (optionally with a label)
Summary Node	. . . △	dots or a triangle
Pointer	⟶	an open-headed arrow

Table 5.1: CoMo's visual vocabulary.

Shapes must then be compared and composed to form the symbols that make up the data structure (Table 5.1). This process consists of comparing a shape's context—all of the elements on the surrounding canvas—to interpret the shape as a symbol. For example, a single character can be recognized as either a pointer or a node's label. The difference is whether the label is in a box (or circle) or not. When symbols are composed to form a data structure, the sketch recognition task is complete.

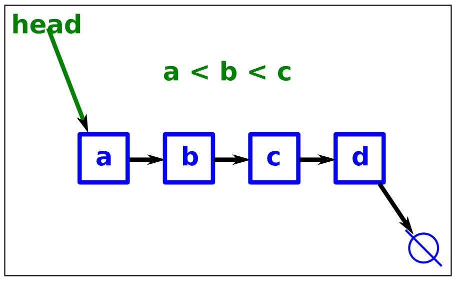


Figure 5-25: A list with four nodes and a constraint showing it is ordered.

Constraints are special types of symbols that serve as restrictions the user places on the space of valid data structures. In CoMo, they are used to order a data structure. The constraints that CoMo handles are the simple binary scalar comparators `<` and `>`. The linked list seen in Fig. 5-25 has the constraint: `a < b < c`, marking it as ordered from smallest to largest, starting at `head`. To interpret the constraint, CoMo breaks it down into two sub-constraints, `a < b` and `b < c`, and processes each one, one at a time. To process `a < b`, it determines that `a` and `b` are connected via `next` and so assigns all pointers named `next` the relationship that, when following the pointer, the target node must be greater than the parent node. More concretely, since `a.next` points to `b`, `a` must be less than `b`. Processing the next sub-constraint yields the same result. Had this been a doubly linked list, the same process would be applied to `prev`; `b` is connected to `a` via `prev`, so `prev` would be assigned the reverse relationship as `next`.

SPT is able to take simple constraints of this kind, however its input format requires that the nodes be given a concrete number instead of an abstract constraint. CoMo finds concrete values for the nodes (e.g., $\texttt{a} = 5$, $\texttt{b} = 6$, $\texttt{c} = 7$, $\texttt{d} = 8$; the number generation algorithm is detailed fully in §5.4.6) to use in code synthesis. Constraints allow CoMo to generate the scenarios SPT needs to synthesize code for manipulations like ordered linked list insertions.

5.4.4 Speech Recognition

CoMo uses CMU's Sphinx speech recognizer to recognize a simple set of statements [37, 32]. Speech is used as a secondary mode of input. Unlike previous work done by Adler [1], CoMo is constrained to understand specific commands. For example, the user says, "Generate code" to initiate code synthesis and, "Do nothing" to answer questions like, "How do you handle this case?" when CoMo presents them with an empty list.

Do you understand?
Like this.
Do nothing.
That does not make sense.
To this (or this . . .).
This is a ______ .
Do you know what a ______ is?
Generate code.
Make a new canvas.
Generate a structure with ___ node(s).

Table 5.2: List of phrases CoMo understands.

Table 5.2 lists the phrases that CoMo understands. The table is divided into four sections that represent the different circumstances under which CoMo expects speech. The user can ask the first utterance, "Do you understand," at any time. CoMo will respond either by asking a question or saying, "I think I understand" if all its questions have already been answered.

The second section contains phrases expected as responses to the questions CoMo poses about specific examples. The first three are responses to the question where CoMo presents a new example and asks, "How do you handle *this* case?" "Like this" is expected after the user demonstrates how to handle the case, "Do nothing" is used to inform CoMo that the case has identical input and output states, and "That does not make sense" informs CoMo that the example should not be used in code synthesis. For example, the user says "Do nothing" when asked how to reverse a singly linked list with one or zero nodes. The final statement, "Like this (or this . . .)" is used as a response to the ellipsis expansion, and can include as many instances of "or this" as desired[13].

```
<label>      = This is a <dsm_clause> ;
<recall>     = Do you know what a <dsm_clause> is ;
<dsm_clause> = <ds_name> [<dsm_name>] | <dsm_name> ;
<ds_name>    = [ordered] ([singly] | doubly) linked list |
               binary [search] tree | stack | queue ;
<dsm_name>   = insert | delete | reverse | swap | traverse |
               search | find (max | min | last) | (left | right) rotation ;
```

Figure 5-26: The subtrees of CoMo's grammar corresponding to labeling and recalling data structures and manipulations by name.

The next section contains phrases that allow the user to label and recall data structures and manipulations by name. CoMo has a list of the names of all the structures and manipulations it can handle. After Sphinx successfully recognizes an utterance, CoMo parses the returned string first to determine whether it is a labeling utterance or a recalling utterance, and then which data structure or manipulation is mentioned. For example, when Sphinx recognizes that the user said, "This is a binary search tree left rotation," CoMo first parses the string to see that it starts with "This

[13]CoMo does not pay attention to how many times "or this" appears in the user's utterance; it only parses the diagram. It allows "or this" to appear arbitrarily many times in an attempt to be more natural. §5.5.1 describes how this visual parsing process proceeds.

is" and then finds and recognizes "binary search tree" as a data structure and "left rotation" as a manipulation.

The final section consists of the three phrases the user can utter to generate one of the three things that CoMo can generate: code, canvases, and structures. Once CoMo has a general structure definition (described above), it can generate example structures with a user-specified number of nodes. As with recalling and labeling data structures, CoMo parses the simple string that Sphinx returns, looking for the number of nodes to generate.

5.4.5 Speech Generation

CoMo uses Google's speech synthesizer to generate audio. A list of utterances was generated and can be played on demand.

Can you draw me a general structure?
What does this expand to?
How do you handle this case?
Okay.
I think I understand.
I will try.
Yes.
No.
I have another question first.
I can't do that.

Table 5.3: Things CoMo is capable of saying.

Table 5.3 lists the phrases CoMo utters in response to, or in order to ask, questions. As with the table on recognized speech, this is broken up into logical sections. The first section contains the phrases relating to asking questions. If the user does not lead with the structure definition (as in the examples above) CoMo can ask them to draw it by saying, "Can you draw me a general structure?" As demonstrated above, "What does this expand to" is used to ask what an ellipsis represents. CoMo says, "How do you handle this case" when presenting the user with a new generated example.

The second section contains the phrases CoMo uses to acknowledge user responses. Though not explicitly stated after each answer in the examples, CoMo responds "Okay" to each answer the user gives. CoMo says, "I think I understand" when all its questions have been asked, "I will try" in response to a code synthesis request, and "Yes" or "No" in response to a question about whether CoMo knows what a specific data structure or manipulation is.

The final section contains the phrases used to inform the user that a requested task could not be performed. If the user requests that code synthesis begin before all of CoMo's questions were answered, it responds by saying, "I have another question first." Similarly, if the user requests that CoMo generate a structure with a number of nodes it is not able to[14], CoMo will say, "I can't do that."

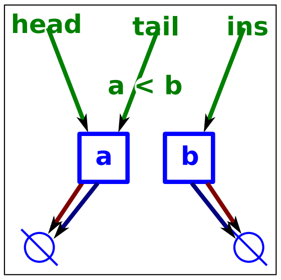
5.4.6 Question Generation

CoMo generates questions based on comparing the kind of information SPT requires with the kind of information the user has thus far given. As discussed in §5.2, SPT needs to know:

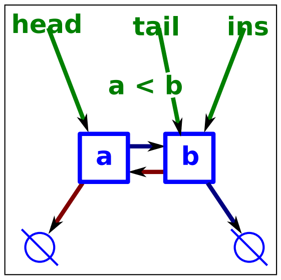
- An inductive definition of the general structure.
- What to do with a structure with four, one, and zero nodes; or—if there is more than one structure—what to do with comparisons between the main structure of sizes four, one, and zero, and any auxiliary structures of size one or zero.
- What to do with pairs of structures with differently valued nodes.

The example above demonstrated how all of these questions are asked except those asking about comparisons between ordered structures. This section describes how these questions are asked for a doubly linked list insertion, then it describes how the constraints are used to generate input for SPT.

[14]Recall from above that one reason for this is that the drawn description of a general list would not allow lists with two nodes to be generated. Alternatively, if the user has not yet drawn the general structure, CoMo will respond to a request to generate structures with more than one node with this phrase.

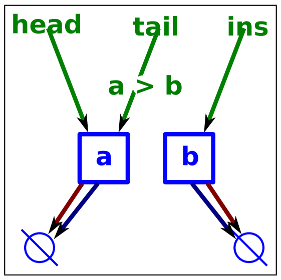


(a) CoMo: "How do you handle *this* case?"

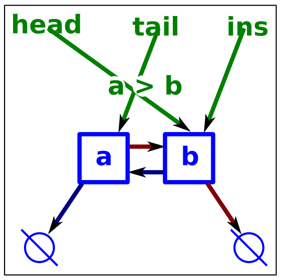


(b) User: (after demonstrating) "Like this."

Figure 5-27: Question about the list being less than the node to insert.



(a) CoMo: "How do you handle *this* case?"



(b) User: (after demonstrating) "Like this."

Figure 5-28: Question about the list being greater than the node to insert.

An ordered doubly linked insertion consists of a two structures: the doubly linked list (i.e., the "main" structure) and the node to insert (i.e., the "auxiliary" structure). To maintain the order of the list, the node is inserted in a location in the list based on its value. To correctly handle constraints, CoMo must go through the possible configurations of relationships between manipulations involving both structures when one is greater than the other and vice versa. Fig. 5-27 and Fig. 5-28 show the constraint-based questions CoMo asks about the doubly linked list insertion. These questions compare simple, differently valued, one-node structures with one another. In these examples, the user would like the ordered list to have the smallest element at the front and the largest element at the back.

SPT's input format requires two pieces of information to synthesize constraint-aware code: concrete values and an explicit label for the variable containing a numeric value. Fig. 5-29 shows these additions to SPT's input. CoMo generates concrete values based on the constraints. It starts at a node accessible by a pointer (in this case it starts with `a`, accessible via `head`) and assigns it a value of 5. It then performs

```
...

scenario scenario4
input head -> a, a.next -> null, tail -> a, a.prev -> null,
      ins -> b, b.next -> null, b.prev -> null,
      a.val -> 5, b.val -> 6
output head -> a, a.next -> b, b.next -> null,
       tail -> b, b.prev -> a, a.prev -> null, ins -> b ;

scenario scenario5
input head -> a, a.next -> null, tail -> a, a.prev -> null,
      ins -> b, b.next -> null, b.prev -> null,
      a.val -> 6, b.val -> 5
output head -> b, b.next -> a, a.next -> null,
       tail -> a, a.prev -> b, b.prev -> null, ins -> b ;

...

data_selector val;
```

Figure 5-29: Subset of input to SPT illustrating constraint specification.

a depth-first search on the node's pointers and assigns values to nodes appropriate to the constraint. In the case of `scenario4`, `a` = 5 and `a < b`, so `b` gets assigned 6. After assigning values, CoMo adds them to the scenario: "`a.val -> 5`" and "`b.val -> 6`." SPT is informed that CoMo has selected "`val`" to contain a numeric value with the directive "`data_selector val`;" This directive informs SPT that it should include value comparisons involving `val` in its search for conditions.

5.5 Assumptions, Algorithms, and Heuristics

This section describes the assumptions CoMo makes, and separates the algorithms it employs from the heuristics it uses. CoMo makes two important simplifying assumptions about the data structures and manipulations that it handles. First, it assumes that every node drawn in a description is of the same type—so, if one node has two pointers, then all nodes should have two pointers. Second, it assumes that only one node can be added, removed, or returned. As mentioned above, the node being removed added or returned is called the *auxiliary* structure, while the structure being added to, returned from, or removed from is called the *main* structure[15]. Pointers

[15]This assumption causes CoMo to ask nonsensical question. The `rtn` pointer in the above example was classified as the auxiliary structure. In a manipulation like a list append, this distinction becomes

other than those that are part of the main structure are assumed to be part of the auxiliary structure. For example, the user used the `rtn` pointer to delimit the node to be returned in the above example, and the `ins` pointer to denote the node to be inserted in an insertion operation.

5.5.1 Parsing a Data Structure

The first algorithm guides the process of "parsing" a data structure—the process of modeling it as a graph and partitioning it into structures. There are two places parsing is used: on inductive ellipsis (or, in the case of binary trees, triangle) definitions and on drawn examples. For the ellipsis (or triangle) drawing, the sketch is parsed to separate each of the options from one another. For an example data structure manipulation, it is used to separate the diagrams into separate structures. This section demonstrates how this simple algorithm functions by going through several examples of its use.

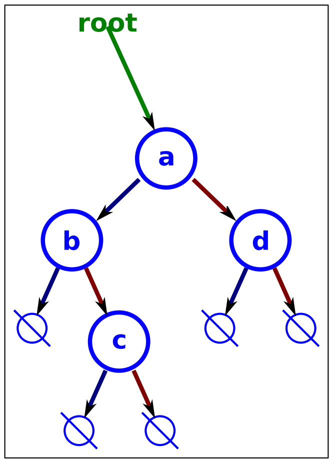


Figure 5-30: A binary tree.

To parse the simple binary tree structure seen in Fig. 5-30, CoMo performs an depth-first search [10] beginning at a randomly-selected pointer, marking nodes as visited as the tree is explored. In this diagram, the only present pointer is `root`, so CoMo starts the depth-first search with `a`. `a` is marked as visited and its children, `b` and `d`, are queued to be explored. Next, `b` is marked and its only child, `c`, is queued.

more clear. CoMo classifies the list as the primary structure and the node to append as an auxiliary structure.

Next, `d` is marked, but since it has no children, none are queued. The `c` node is marked and processed the same way. After this search, all the nodes in the tree have been marked, and CoMo has recognized that this is a single structure.

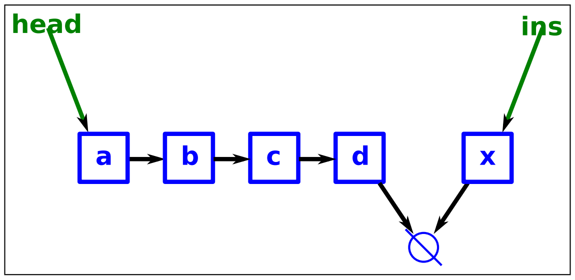


Figure 5-31: A doubly linked list and extra node. The list is classified as the "main" structure, and the extra node as an auxiliary structure.

Fig. 5-31 shows an example manipulation involving two structures: a linked list insert. In this example, there are two pointers pointing at two separate structures; `head` points at a list with four nodes, and `ins` points at a node to insert. As with the tree, CoMo starts by randomly selecting a pointer from the sketch. In this case, it selects `head` and marks all the nodes in the list starting at `a` and ending at `d`. When it reaches `d`, it sees that its `next` pointer points at null, and so the search terminates. At this point, `a`–`d` have been marked as part of the structure starting with `head`. CoMo then selects the only remaining pointer from the drawing, `ins`, which points to the `x` node. After this part of the sketch is searched through, `x` is marked as belonging to the structure `ins` points to. Since the algorithm does not include null symbols in its search, nodes from two separate structures using the same null symbol are not confused as being a single structure.

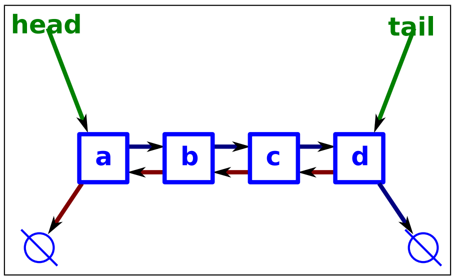


Figure 5-32: A doubly linked list.

Fig. 5-32 shows one final example of a data structure: a doubly linked list. This structure contains two pointers, `head` and `tail`, and two node pointers, `prev` and `next`. CoMo starts out parsing this structure the same way it did in the previous example, with the `a` node that `head` points to. As before, `a`–`d` are marked as belonging to the structure that `head` points to. This example differs in the next step; CoMo explores the structure that `tail` is pointing to (starting at the `d` node) and finds that `d`–`a` belong to the structure that `tail` points to, as well. Since all of these nodes are connected and both `head` and `tail` were found to be pointing at the structure they form, this collection of four nodes is classified as a single structure with two pointers.

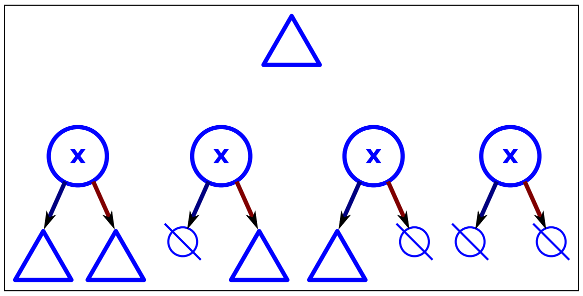


Figure 5-33: A binary tree repetition definition. CoMo parses four options.

Fig. 5-33 shows the triangle expansion for a binary tree[16]. As with an ellipsis expansion, CoMo asks this question by displaying a copy of the repetition symbol (in this case a triangle) at the top of a new canvas and asking, “What does this expand to?” The user answers by drawing the four options seen in the figure. Parsing this sketch proceeds slightly differently than with the above examples, because there are no pointers to start with. Instead, every element in the graph—this time we need to include null symbols also—is iterated over and, if it is a node, its pointers are explored. CoMo starts with a randomly selected element (excluding the cloned triangle at the top)—in this case, `b` is selected—and as with the examples above, `b` is marked and its pointers are explored. The null `b`’s left pointer is pointing at and the triangle `b`’s right pointer is pointing at are marked as visited. As the process continues, some elements will be visited twice. When this happens, CoMo ignores them (they have already been marked) and continues. When all the elements have been visited and marked, the parsing process is complete.

[16]Triangles are commonly used as the repetition symbol in tree drawings.

5.5.2 Abstracting the CFG

As mentioned above, CoMo does not require the user to select a control flow graph (CFG); CoMo selects one for them. The next algorithm that CoMo uses dictates how it selects the CFG for SPT to use in code synthesis, and adds variables if necessary.

```
main(Node head, Node tail, Node ins) {
  Node tmp1, tmp2;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

(a) The default CFG.

```
main(Node head, Node tail, Node ins) {
  Node tmp1, tmp2;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(8)
}
```

(b) The CFG used on the second try.

Figure 5-34: The CFGs used to synthesize code for an ordered, doubly linked list insertion.

The CFG that CoMo selects is seen in Fig. 5-34a. This CFG was selected because it is general enough to work with all the manipulations except one: the doubly linked list insertion. To successfully synthesize code for the doubly linked list insertion, CoMo follows an augmentation rule. If synthesis fails or if it takes longer than an hour (the hour heuristic is discussed below), CoMo adds another five lines to the bottom of the CFG and reruns SPT. The resulting CFG that is used to synthesize code for the ordered, doubly linked list insertion is seen in Fig. 5-34b.

Fig. 5-35 shows the raw and cleaned code synthesized for the ordered, doubly linked list insertion with SPT. This case illustrates one shortcoming of how CoMo uses SPT. SPT determines that the first CFG is insufficient to synthesize the specified manipulation by exhausting all code options in the CFG. This process takes several hours to run. Once the second CFG is selected and run, synthesis then takes about 25 minutes.

5.5.3 Heuristics

This section discusses the five heuristics that CoMo employs. First, recall that CoMo asks about examples with the main structure containing four, one, and zero nodes.

```
void main(){
  tmp2=head;
  head=ins;
  ins=tmp1;
  while(head!= null && tmp2!= null && head.val < tmp2.val){
    tmp2=head;
    ins=tail;
    head=tail;
    tail=tmp2;
    tmp1=head;
  }
    if(tail!=null) tail.next=head;
    if(tail!= null && head!= null && tail.val < head.val) ins=tmp2;
    if(head!=null) head.prev=tail;
    if(tmp1==null) ins=head;
    if(head!=null) tail=head;
    if(tmp2!=null) head=tmp2;
}
```

(a) Raw synthesized code.

```
tmp2=head;
head=ins;
ins=tmp1;
while(head!=null && tmp2!=null && head.val < tmp2.val){
  tmp2=head;
  ins=tail;
  head=tail;
  tail=tmp2;
  tmp1=head;
}
if(tail!=null) tail.next=head;
if(tail!=null && head!=null && tail.val < head.val) ins=tmp2;
if(head!=null) head.prev=tail;
if(tmp1==null) ins=head;
if(head!=null) tail=head;
if(tmp2!=null) head=tmp2;
```

(b) Cleaned synthesized code.

Figure 5-35: The code synthesized for an ordered, doubly linked list insertion.

The number four is a heuristic. SPT needs an example with some number of nodes larger than zero and one, and examples with four nodes were found to be effective in synthesizing correct code.

Next, the order that CoMo asks its questions in was also selected because it seemed natural. The order is:

1. Ask about the general structure.
2. Ask about the repetition definition.
3. Ask about concrete structures.

The concrete structure questions differ when there is just a main structure, a main structure and an auxiliary structure, and when there is an ordered main structure and an auxiliary structure. When there is only a main structure, these questions' order is:

1. What do you do with a structure with four nodes?
2. What do you do with a structure with one node?
3. What do you do with a structure with no nodes?

When there is a main and an auxiliary structure, the questions' order becomes:

1. What do you do when the *main* structure has four nodes and the *auxiliary* structure has one node?
2. What do you do when the *main* structure has one node and the *auxiliary* structure has one node?
3. What do you do when the *main* structure has no nodes and the *auxiliary* structure has one node?
4. What do you do when the *main* structure has four nodes and the *auxiliary* structure has no nodes?

5. What do you do when the *main* structure has one node and the *auxiliary* structure has no nodes?

6. What do you do when the *main* structure has no nodes and the *auxiliary* structure has no nodes?

When there are multiple structures and the main structure is ordered, the questions' order is the same as above[17], however, instead of asking about the situation where both structures have one node, CoMo the two questions listed below, in the order shown:

1. What do you do when the *main* structure has one node and the *auxiliary* structure has on node, and the main structure's node has a value *smaller than* the auxiliary node's value?

2. What do you do when the *main* structure has one node and the *auxiliary* structure has on node, and the main structure's node has a value *greater than* the auxiliary node's value?

The next heuristic CoMo employs was borrowed from Adler's MIDOS [1]. MIDOS waits three seconds before deciding that the user has completed their input. Similarly, when giving an example that does not require a verbal acknowledgment (e.g., where CoMo expects the user to respond, "Like this"), CoMo waits three seconds before determining that the user has completed demonstrating the manipulation and moving on to a new question.

If SPT takes more than one hour to run, CoMo assumes that the input it was given is over-constrained, and that it will therefore fail to produce code. When this happens, CoMo stops SPT and reruns it with the second, larger CFG. The value of one hour was empirically chosen because the longest SPT takes to successfully synthesize code with input generated by CoMo is about 45 minutes. The extra 15 minutes were added to give SPT some leeway.

[17]Note that CoMo does not assume anything about the values of the nodes. The user must input a constraint to inform CoMo of the nodes' values.

Finally, several small heuristics exist within the sketch recognition module. One example of such a heuristic is how CoMo determines whether to use Ouyang's classifier or PaleoSketch to recognize a stroke. Small strokes—strokes that have bounding boxes smaller than 40 pixels wide by 60 pixels high—are classified by Ouyang's recognizer, and all others are classified by PaleoSketch. This heuristic places a constraint on the user that simplifies the sketch recognition task. Most heuristics, however, come from PaleoSketch [7], which has 26 heuristics used throughout the single-stroke recognition process. Additionally, the remainder of the sketch recognition pipeline—NeaSketch and the context recognizer—have some small number of heuristics based on Ehrenfels' Gestalt principles [36], and some that were empirically determined to work. These heuristics aide CoMo in determining things like when lines are parallel or diagonal, and also when lines' endpoints are close enough to be considered connecting.

5.5.4 Abstractions

During discussions, CoMo continually updates abstract representations of the node, structure, and manipulation definition. It keeps track of the following things about nodes:

- The types of pointers a node has (referred to as *pointer types*—these encapsulate the difference between "`next`" and "`prev`" pointers).
- The name of each pointer type (initially "`ptr`*n*").
- The constraints (if any) associated with each pointer type.

the following things about structures:

- The number of nodes in a structure.
- The name of the nodes (possibly the default: a single-letter name starting at "`a`" based on the number of nodes in the sketch).
- The number and names of pointers.

- Where each pointer is pointing (i.e., the configuration of the structure).

and the following things about manipulations:

- The sequence of states in each example's data structure manipulation.
- The general structure definition (like those seen in Fig. 5-7 and Fig. 5-8).

CoMo uses these abstractions to analyze the examples given and determine which questions remain unanswered. Some examples of how these abstractions are used include:

- To properly generate a structure with a single node, CoMo must know that a node has two pointers.
- To generate SPT's input, CoMo must know the names of the pointers and nodes, and whether constraints exist and what they are.
- To understand what questions have been answered (and therefore which remain to be asked), CoMo must know the configurations of the examples on the canvas.

Future work discusses how this information may be further used.

5.6 Limitations

This section discusses some of the interesting limitations preventing CoMo from holding a conversation with a user about more complex data structures—data structures that violate one of the assumptions mentioned above. The following sections describe what changes to CoMo would allow it to interact about an AVL tree insertion and a min-heap pop. Brief descriptions of the manipulations are given in each section. For a more complete description, see Cormen et al.'s *Introduction to Algorithms*, 3$^{\text{rd}}$ edition [10]. Note that each of the manipulations presented here would also require a more sophisticated code synthesis system, however since that is not the topic of this thesis, this section will assume that the availability of a synthesis system that works identically to SPT for these manipulations.

To be able to converse about new manipulations, one or more observational user studies would have to be performed (similar to the one presented in Chapter 3) to determine how people describe them. As before, CoMo's interaction could then be modeled after the findings of these user studies. Since these user studies have not yet been performed, the following sections conjecture plausible findings, and model CoMo's augmentation after them.

5.6.1 AVL Tree Insertion

An AVL tree is a self-balancing binary tree with the property that the heights of any node's two subtrees will never differ by more than one. This is called the AVL tree property.

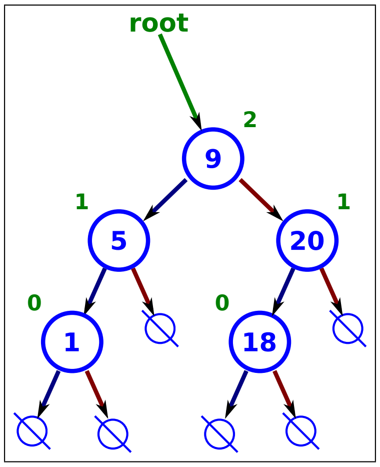
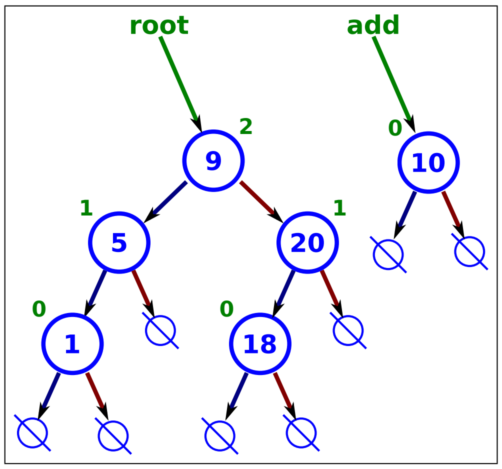
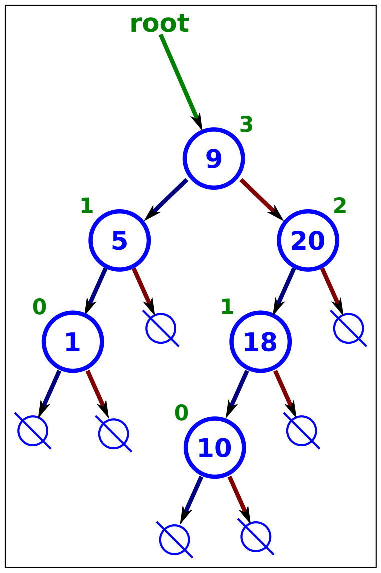


Figure 5-36: An example AVL tree.

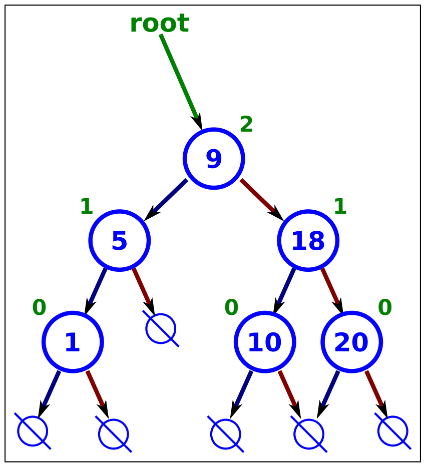
Fig. 5-36 shows an example AVL tree. Notice that each node also stores its height—the length of its largest subtree—typically drawn above and to the side of each node. These heights can be used to compute whether a rotation needs to be performed to balance the tree by comparing each node's left child's height u against its right child's height v and determining whether $|u - v| \geq 2$. If this inequality is true, one of the six possible tree rotations must be performed to balance the tree. Treating null symbols as though they have a height of -1 makes this inequality work for all cases in an AVL tree.



(a) The AVL tree and a node to add.



(b) The AVL tree after the node was added with a non-balancing BST insertion, violating the AVL property.



(c) The AVL tree after being rebalanced.

Figure 5-37: An AVL tree insertion consisting of a BST insertion followed by a rotation.

Fig. 5-37 shows an example AVL tree insertion. The insertion starts with the AVL tree and node to insert seen in Fig. 5-37a. A binary search tree insertion is performed and the heights of the nodes are updated (Fig. 5-37b). The node labeled "20" violates the AVL tree property—its left child has a height of 1, its right child has a height of -1, and $|(-1) - 1| \geq 2$—so a rotation must be performed. The tree is balanced with a right rotation about the node labeled "20," the heights are updated, and the tree is again a valid AVL tree (Fig. 5-37c).

CoMo needs three additional abilities to be able to converse about this manipulation. First, CoMo needs the ability to store multiple values in nodes (currently, only a single numerical value can be stored) An additional value is required to store the height of each subtree. This addition is a two-part task: first augmenting the context recognizer in the sketch recognition module to recognize that a number above and to the side of a node signifies a second value, and second augmenting CoMo's internal representation of data structures to lift the "one value" constraint. Adding to the internal representation would involve giving each value a default name—perhaps "`val1`" and "`val2`"—and allowing one or both to appear in constraints.

The next augmentation CoMo needs is allowing the user to specify a complex constraint like the AVL tree property, perhaps by drawing "$|\texttt{ptr1.val2} - \texttt{ptr0.val2}| \leq 2$" on the canvas. This would require augmenting the constraint-handling code that currently only works on simple constraints like "$a < b < c$."

The third and final augmentation required is the ability to verbally compose manipulations. The AVL tree insertion involves five manipulations: a BST insertion and four tree rotations. As shown above, the first step in each AVL insertion is a BST insertion. This can result in violating the AVL property, which requires one of the four rotations to be performed. Determining which of the rotations must be performed is a matter of following four simple rules.

Fig. 5-38 shows a snapshot of how a user might describe an AVL tree insertion to a user. Describing it would consist of two phases. It would start with the user saying, "Perform a binary search tree insertion," which CoMo would do, resulting in Fig. 5-38. Next, the user would say, "Since *this* is true," while drawing the condition

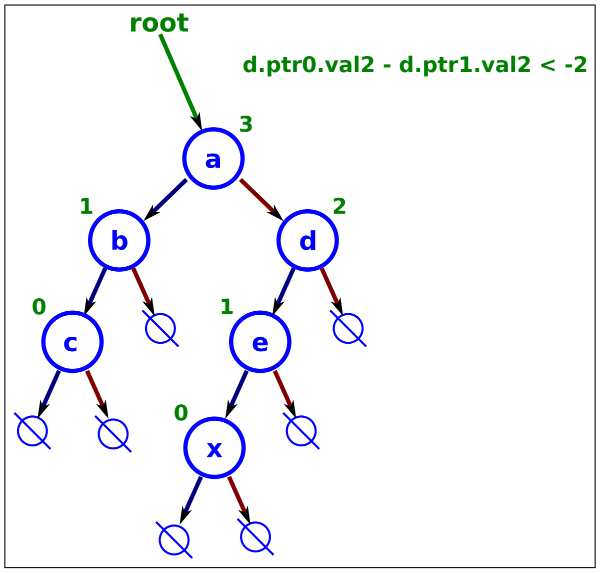


Figure 5-38: A snapshot of a possible interaction with CoMo about an AVL tree insert. CoMo has just performed a BST insertion, and the user is explaining why a rotation is required.

seen on the canvas, "then do a right rotation about *this* node," while pointing at the `d` node. Giving an example for each different rotation would result in CoMo stitching together previously learned manipulations to form a new, more complex one, like an AVL tree insertion.

5.6.2 Min-Heap Pop

Min-heaps are structures that always have the minimum element at the top. Pop operations on min-heaps return the smallest element—the one at the top of the heap. Heaps implemented as binary trees are always complete, and each node has a larger value than its parent. These two constraints on heaps are together called the heap property.

Fig. 5-39 shows an image of one possible min-heap. Popping the top element off of this heap proceeds in three steps: removing the top element, putting the "last" element in its place, and *re-heapifying* the heap—a recursive operation that ensures the heap property is met.

Fig. 5-40 demonstrates how a min-heap pop proceeds. Fig. 5-40a shows the heap

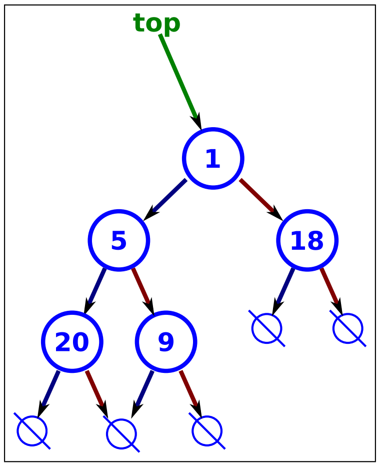
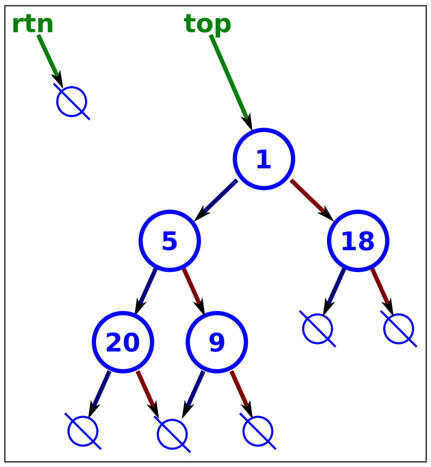
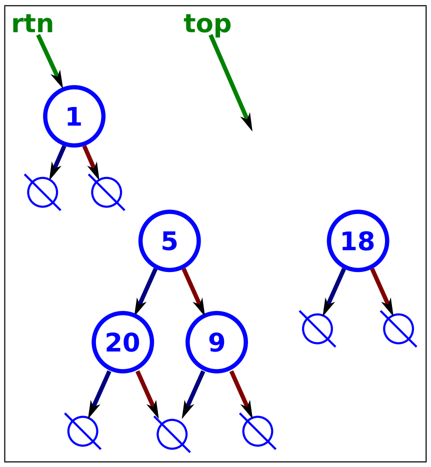


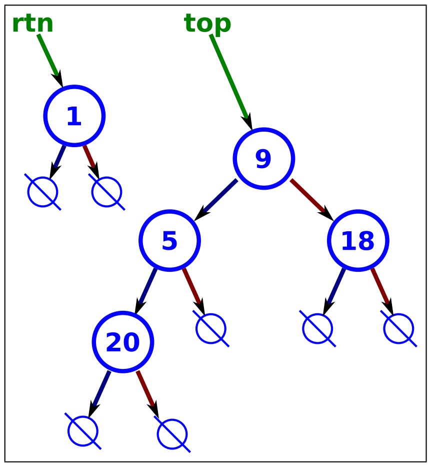
Figure 5-39: An example min-heap.



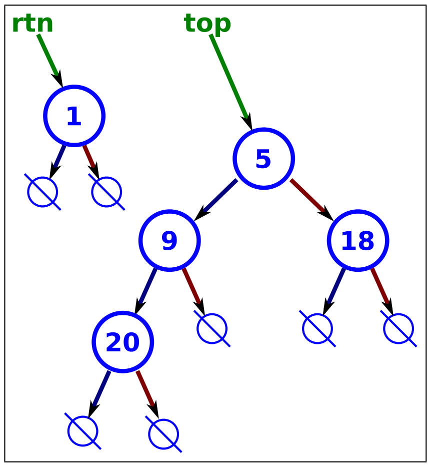
(a) A heap.



(b) The top element is removed.



(c) The last element is placed at the top.

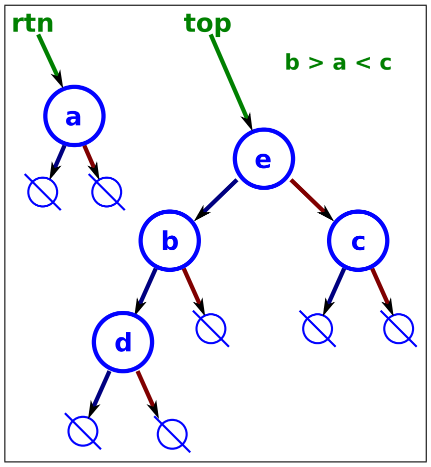


(d) The heap is (recursively) re-heapified.

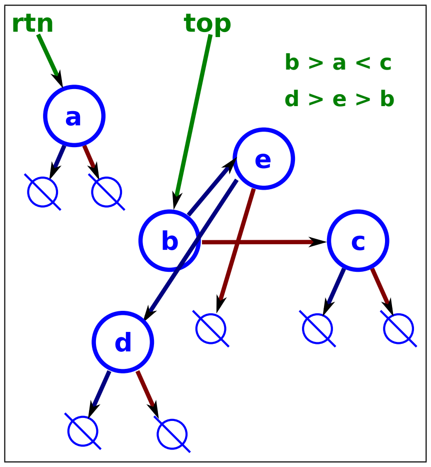
Figure 5-40: The process of popping an element from a heap; the minimum element is returned, and the next smallest is placed at the top of the heap, ready for the next pop operation.

from the example diagram above next to a `rtn` pointer, meant to denote the node to return. First, the top element of the heap is removed and returned[18] (Fig. 5-40b). The heap is now incomplete—it is missing a root element—so the bottom-most element in the heap is removed from its position and placed at the root of the tree (Fig. 5-40c). The heap is complete, but it violates the heap property. To fix this, an operation, usually called "re-heapify," must be performed. This is the process by which, starting at the root node, each node is recursively swapped with its smallest-valued child if that child's value is less than it. Once this operation is complete, the heap looks like the diagram in Fig. 5-40d and the min-heap pop is complete.

Assume that a user study was performed that determined people describe heap operations in terms of the recursive re-heapify manipulation.[19] Let us also assume that CoMo has been taught to re-heapify a list.



(a) The user has clicked-and-dragged . . .



(b) The user draws a constraint . . .

Figure 5-41: A snapshot of a user explaining a min-heap pop manipulation to CoMo. Here, the user explains why a swap must happen.

In principle, CoMo's current abilities should suffice in handling this manipulation; what is missing is a sufficient code synthesis system. Fig. 5-41 shows a snapshot of the interaction with the user about the min heap operation. This figure has analogous

[18]Note that this description is conceptual; it ignores the order of operations. An actual implementation would proceed differently, however the purpose of this explanation is to communicate the idea of what steps go into a min-heap pop, not to mirror an implementation.

[19]This may in fact be the case, since adding and removing elements from a heap require it to be re-heapified as a final step.

diagrams to Fig. 5-40c and Fig. 5-40d; after describing the heap (complete with the constraint "`b > a < c`"), simple pop operation, and placing the last element at the top of the heap (Fig. 5-41a), the user describes why the `e` node must be swapped down one level by writing, "`d > e > b`" and clicking and dragging the heads of pointers to show the heap in its re-heapified state.

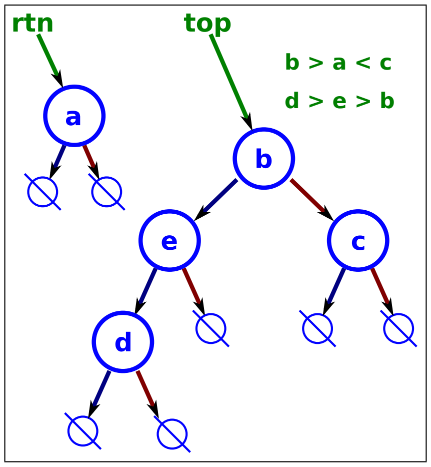


Figure 5-42: The ability to move nodes' positions allows the user to clean up the sketch. This figure differs from Fig. 5-41b only in the positions of the nodes.

One addition that would improve CoMo's interaction would be the ability to move the positions of nodes. As is apparent from the diagram, the inability to move nodes can quickly result in messy drawings. The conceptually simple change of swapping two nodes (by updating five pointers) results in many overlapping nodes. Swapping node positions could be achieved by clicking and dragging nodes. After swapping the pointers, clicking and dragging nodes would allow the user to clean up the sketch, resulting in a figure like that seen in Fig. 5-42.

5.7 Conclusions

This chapter discussed how CoMo guides the interaction with the user to ask and answer questions, interpret answers and sketches, and communicate multimodally. It described how the mixed-initiative code-generation framework's simple set of rules for generating example questions—asking about cases involving four, one, and zero nodes—and simple rule for guessing an appropriate CFG—starting with a default one, augmenting it, and retrying—result in a powerful system able to hold a limited conversation about 50 manipulations on 8 structures, synthesize code, and help the user verify the code's correctness with animations. Further, it discussed some of CoMo's limitations by describing the augmentations needed to enable CoMo to discuss an AVL tree insert, a min-heap pop, and a red-black tree search.

Chapter 6

Domain

CoMo's domain (i.e., the set of data structures and manipulations for which it can successfully synthesize code) is a set of common pointer- and node-based data structure manipulations that conform to the following constraints:

1. All nodes in the structure are of the same type.
2. Nodes contain at most one value and some number of pointers.
3. The manipulation does not contain any recursive or subroutine calls.
4. At most one node can be added, removed, or returned from a structure.

This chapter lists the data structures and manipulations in CoMo's domain, then it describes the interaction for a representative subset of manipulations.

6.1 Summary of Structures

CoMo can handle the following data structure manipulations:

1. Stack
    (a) push
    (b) pop
    (c) poll
2. Queue
    (a) enqueue

   (b) dequeue
   (c) peek
3. Singly Linked List
   (a) insert
   (b) delete
   (c) reverse
   (d) append
   (e) prepend
   (f) remove first
   (g) remove last
   (h) find first
   (i) find last
4. Doubly Linked List
   (a) insert
   (b) delete
   (c) reverse
   (d) append
   (e) prepend
   (f) remove first
   (g) remove last
   (h) find first
   (i) find last
5. Ordered, Singly Linked List
   (a) insert
   (b) reverse
   (c) remove minimum
   (d) remove maximum
   (e) find minimum
   (f) find maximum
6. Ordered, Doubly Linked List:
   (a) insert
   (b) reverse
   (c) remove minimum
   (d) remove maximum
   (e) find minimum
   (f) find maximum
7. Binary Tree
   (a) left rotation
   (b) right rotation
   (c) left-right rotation
   (d) left-left rotation
   (e) right-left rotation
   (f) right-right rotation
8. Binary Search Tree
   (a) left rotation
   (b) right rotation
   (c) left-right rotation
   (d) left-left rotation
   (e) right-left rotation

(f) right-right rotation

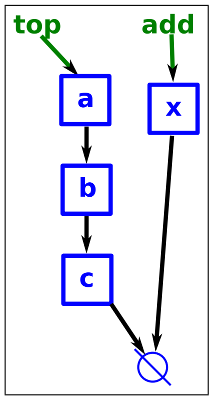
(g) find minimum

(h) find maximum

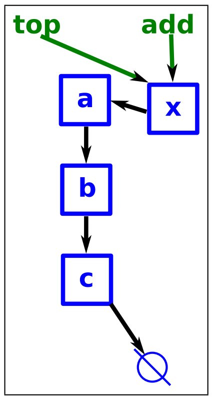
This sums up to a total of 50 manipulations on 8 data structures.

6.2 Similarity Classes

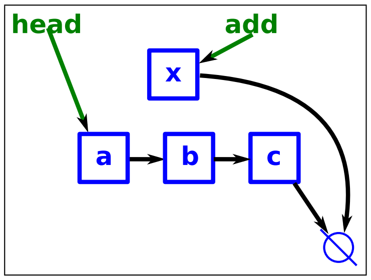
There is quite a bit of similarity across the different kinds of data structure manipulations that CoMo can handle. For example, a stack push and a linked list prepend look virtually identical in code and thus virtually identical when drawn. As seen in Fig. 6-1, the differences between the two manipulations are just the labels chosen for their pointers ("head" or "top") and the direction they are drawn in (vertical or horizontal).



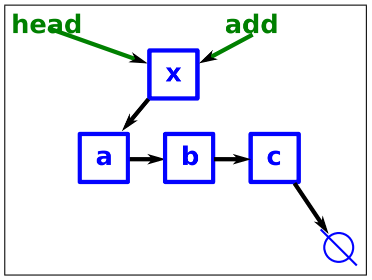
(a) A stack before the push manipulation.



(b) The same stack after the manipulation.



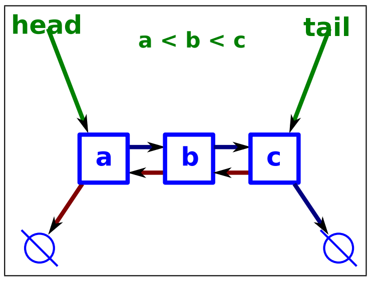
(c) A list before the prepend manipulation.



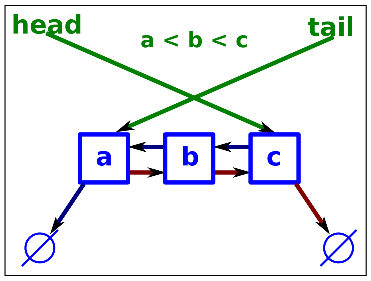
(d) The same list after the manipulation.

Figure 6-1: A stack push and a linked list prepend. Note that these two manipulations are nearly identical.

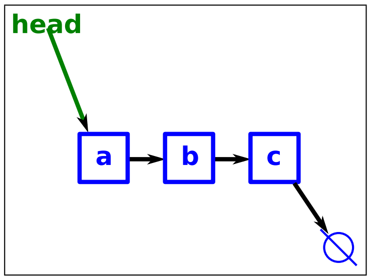
Manipulations are also considered similar when the set of operations in one manipulation is a strict subset of the set of operations in the other. Fig. 6-2 demonstrates this by comparing an ordered, doubly linked list reversal to an unordered, singly linked list reversal. The singly linked list reversal consists of a strict subset of the operations required to perform the ordered, doubly linked list reversal; it requires half the pointer reassignments and no constraint. Demonstrating that CoMo can discuss and successfully generate code for the more complex of the two effectively demonstrates that it can successfully discuss and generate code for the less complex of the two.



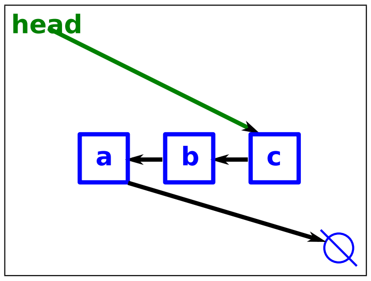
(a) An ordered, doubly linked list before the reverse manipulation.



(b) The same list after the manipulation.



(c) An unordered, singly linked list before the reverse manipulation.



(d) The same list after the manipulation.

Figure 6-2: List reversals for an ordered, doubly linked list and unordered singly linked list. Note that the operations required for the singly linked list reversal are a strict subset of the operations required for ordered, doubly linked list reversal.

Table 6.1 groups the list-based manipulations into similarity classes. If CoMo can successfully discuss the most complicated example in a similarity class and generate code for it, then it can handle the remaining, less-complex examples. This chapter gives seven representative examples of these manipulations. Manipulations with a section number below them are demonstrated in the referenced section. Most of the demonstrated manipulations are on singly linked list-based data structures because manipulating the extra pointer in a doubly linked list-based data structure adds visual complexity without adding any useful information. Two manipulations on doubly linked list-based structures are included (the queue dequeue in §6.4 and the ordered doubly linked list insertion in §6.8) to demonstrate this complexity.

Stack	LL	Ordered LL	Queue	DLL	Ordered DLL
push	prepend		enqueue	prepend	
(§6.3)				append	
pop	remove-first	remove-min	dequeue	remove-first	remove-min
			(§6.4)	remove-last	remove-max
poll	find-first	find-min	peek	find-first	find-min
				find-last	find-max
	reverse	reverse		reverse	reverse
	(§6.5)				
	insert			insert	
	delete			delete	
	(§6.7)				
		insert			insert
					(§6.8)
	append				
	remove-last	remove-max			
	find-last	find-max			
	(§6.6)				

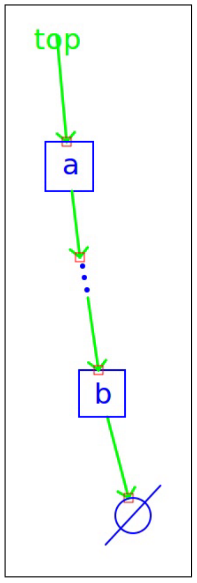
Table 6.1: The list-like data structure manipulations about which CoMo can interact, separated by similarity. "LL" and "DLL" stand for "Linked List" and "Doubly Linked List" respectively.

Table 6.1 only sorts the list-based manipulations. There are an additional six kinds of rotations that can be performed on binary trees and binary search trees (see §6.9) and a find-min and find-max operation on a binary search tree. The find-min and find-max operations are in the same similarity class as the linked list find-last[1]; see §6.6 for a very similar manipulation.

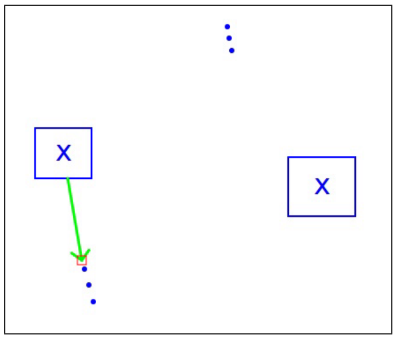
[1]Indeed, in the worst case of a binary search tree being perfectly unbalanced, it very nearly resembles a singly linked list list. The only difference is the additional null-pointer that each tree node has.

6.3 Stack Push

Stacks are sequential lists of objects that ensure the first element added to the stack will be the last element out of it (FILO). Stack operations consist primarily of "push" operations that add an element to the top of the stack, and "pop" operations that remove an element from the top.



(a) The user draws a general stack.



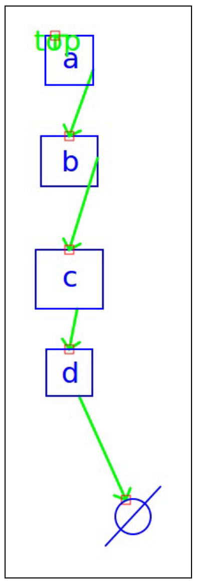
(b) The user defines the ellipsis expansion.

Figure 6-3: The general stack and ellipsis definition

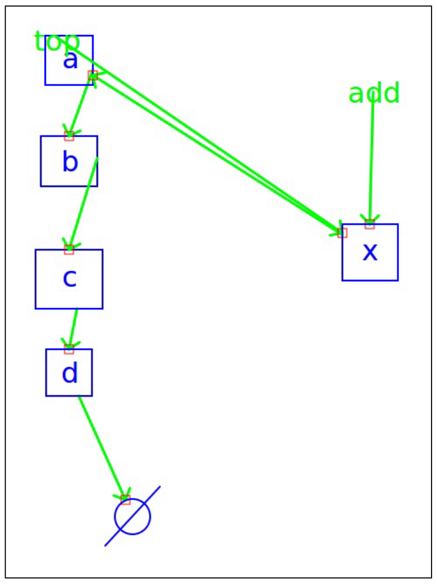
The user starts the interaction by asking CoMo, "Do you know what a stack is?" CoMo replies "No," so the user draws the general ellipsis seen in Fig. 6-3a. CoMo asks, "What does this expand to?" while displaying a copy of the ellipsis from the user's general stack drawing. The user answers the question by drawing the two possible expansions seen in Fig. 6-3b.

Next, CoMo asks what to do with a stack containing four elements (Fig. 6-4a). The user responds by adding an `add` pointer pointing at an `x` node, and updating the sketch by clicking and dragging `x`'s pointer to point at the top of the stack, and the `top` pointer to point at the `x` node. The final state of the canvas can bee seen in Fig. 6-4b.

The next three questions are about when the stack has four, one, and zero nodes



(a) CoMo: "How do you handle this case?"



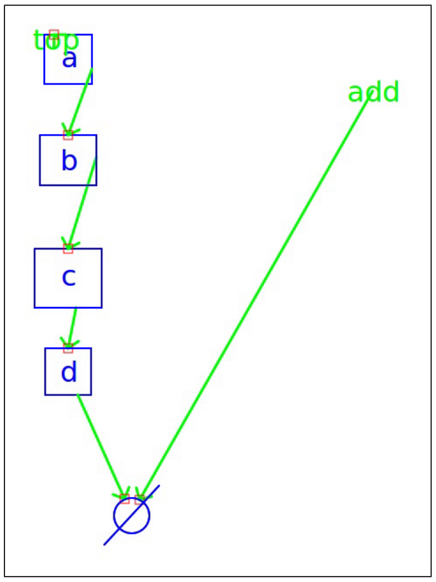
(b) User (after adding `add` and performing the manipulation): "Like this."

Figure 6-4: A question and answer about pushing onto a stack with four nodes.

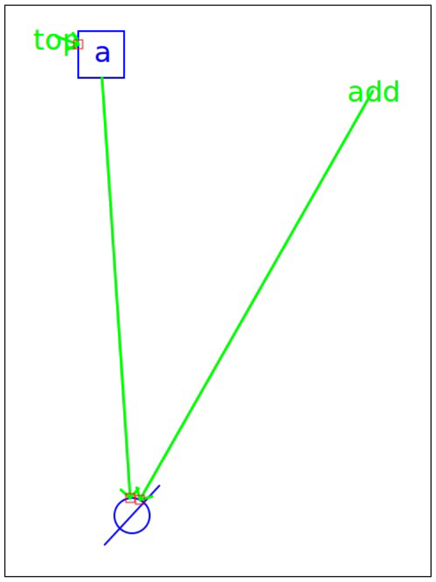
and the `add` pointer is pointing at null. Fig. 6-5 shows the structures that CoMo generates while asking "How do you handle this case?" The user answers "Do nothing" to each of these.

The final two questions that CoMo asks are how to push a node onto a stack with one (Fig. 6-6) and zero (Fig. 6-7) nodes in it. These questions proceed identically: CoMo generates the sketches and asks "How do you handle this case?" The user answers by updating the sketch by clicking and dragging the pointers' heads and saying, "Like this." After the user answers the final question, CoMo says, "I think I understand."

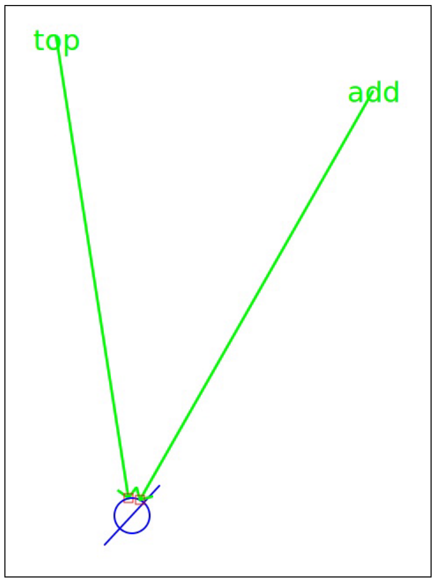
The user cannot think of anything to add and says, "Generate code." CoMo generates the input to SPT seen in Fig. 6-8 and after about five seconds, SPT generates the code seen in Fig. 6-9a. CoMo cleans the code and presents it to the user. Fig. 6-9 shows the cleaned code. The user tests the code out on stack with four elements and a new stack with five elements. They are satisfied that the code is correct, so the interaction is complete.



(a) CoMo: "How do you handle this case?"
User: "Do nothing."

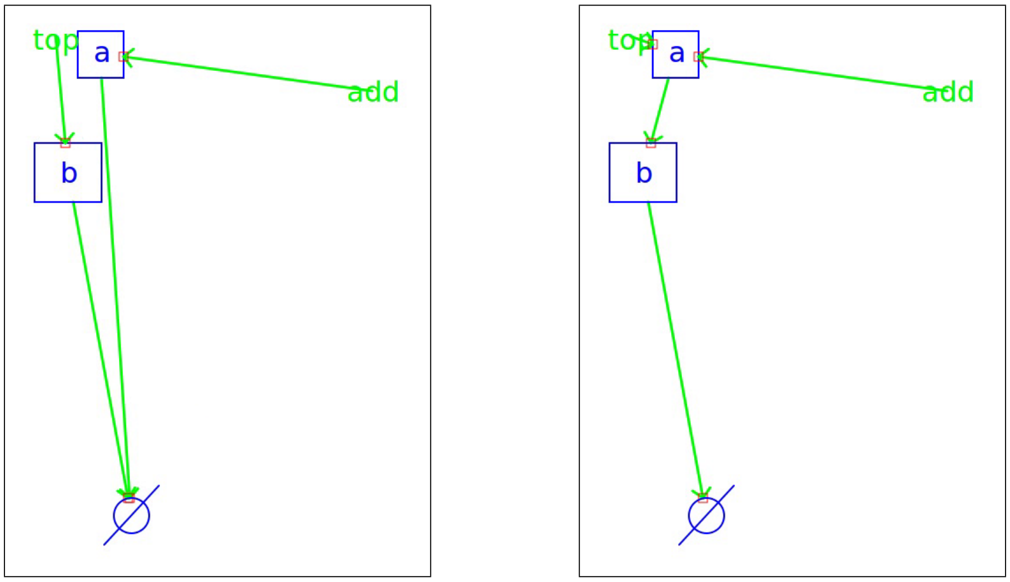


(b) CoMo: "How do you handle this case?"
User: "Do nothing."



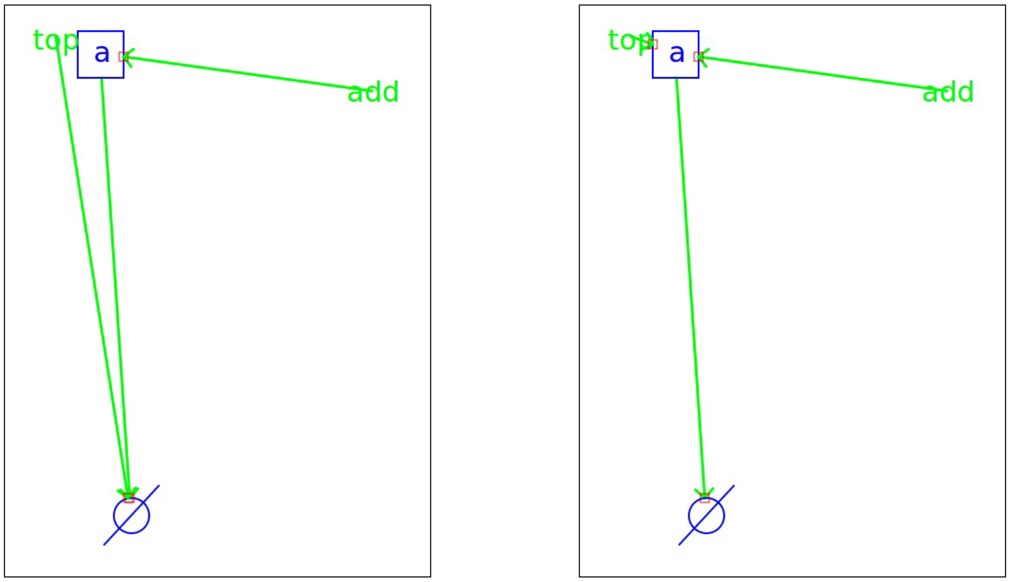
(c) CoMo: "How do you handle this case?"
User: "Do nothing."

Figure 6-5: The questions and corresponding answers about stacks with nothing to add. These questions proceed identically to one another.



(a) CoMo: "How do you handle this case?" (b) User (after updating the sketch): "Like this."

Figure 6-6: Question and answer about what to do with a stack containing one node and an `add` pointer pointing at a separate node.



(a) CoMo: "How do you handle this case?" (b) User (after updating the sketch): "Like this."

Figure 6-7: Question and answer about what to do with an empty stack and an `add` pointer pointing at a node.

```
scenario scenario2
input top -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      add -> x, x.ptr0 -> null
output top -> x, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
       add -> x, x.ptr0 -> a ;

scenario scenario3
input top -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      add -> null
output top -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
       add -> null ;

scenario scenario4
input add -> a, top -> b, a.ptr0 -> null, b.ptr0 -> null
output add -> a, top -> a, a.ptr0 -> b, b.ptr0 -> null ;

scenario scenario5
input add -> null, top -> a, a.ptr0 -> null
output add -> null, top -> a, a.ptr0 -> null ;

scenario scenario6
input add -> a, top -> null, a.ptr0 -> null
output add -> a, top -> a, a.ptr0 -> null ;

scenario scenario7
input add -> null, top -> null
output add -> null, top -> null ;

main(Node add, Node  top) {
  Node tmp2, tmp1;
  ?=(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 6-8: The input CoMo generates for SPT.

```
void main(){
  add=add;
  top=top;
  add=add;
  while(false){
    add=top.next;
    top.next=add;
    add=top;
    add=top.next;
    top=add;
  }
    if(false) top=add.next;
    if(add!=null) add.next=top;
    if(add!=null) top=add;
}
```

(a) The raw code.

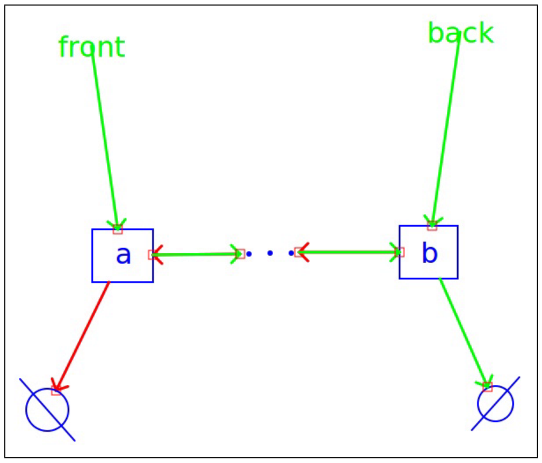
```
if(add!=null) add.next=top;
if(add!=null) top=add;
```

(b) The cleaned code.

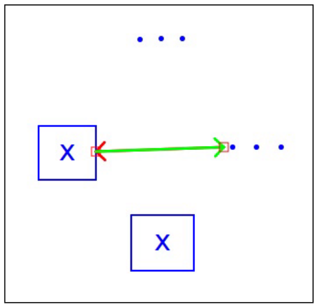
Figure 6-9: The raw and cleaned synthesized code for the stack push manipulation.

6.4 Queue Dequeue

A queue, like a stack, is a structure that keeps the order of a set of elements. Unlike a stack, the first in is guaranteed to be the first out (FIFO). One way to implement a stack (the way presented here) is with a doubly linked list.



(a) The user draws a general queue.



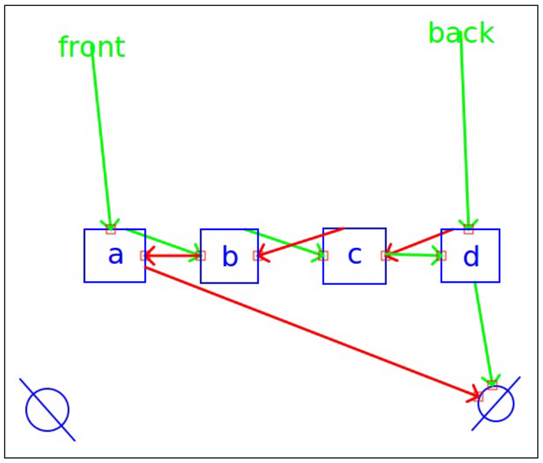
(b) The user defines the ellipsis expansion.

Figure 6-10: The definition of a general queue.

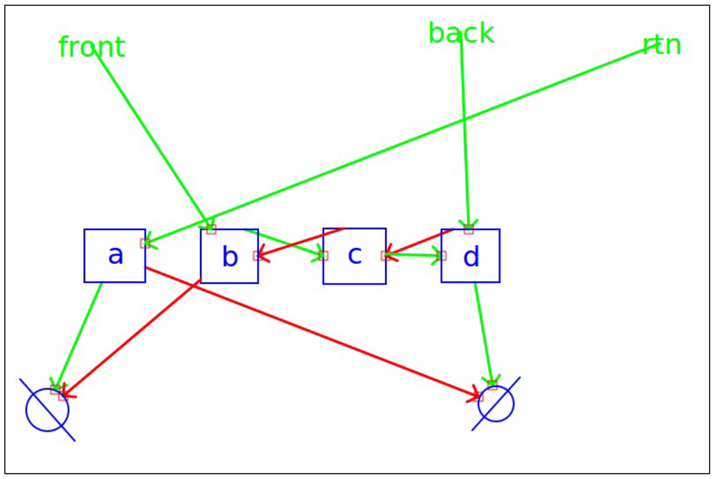
The user starts by asking CoMo, "Do you know what a queue is?" CoMo responds, "No." The user then draws a general queue, seen in Fig. 6-10a. CoMo recognizes the ellipsis in the general queue to be a form of repetition, but does not know what for the repetition takes, so it asks, "What does this expand to?" and displays a new canvas with a copy of the ellipse at the top. The user answers by drawing the inductive definition of the ellipsis, seen in Fig. 6-10b, and saying, "To this or this."

Next, CoMo asks the user what to do with a queue containing four elements. CoMo generates the sketch seen in Fig. 6-11a and asks, "How do you handle this case?" The user responds by adding a `rtn` pointer pointing at null, moving it to point at the front of the queue, updating the pointers to remove the front element from the queue, and saying, "Like this."

The next three questions are seen in Fig. 6-12, Fig. 6-13, and Fig. 6-14. In each case, CoMo brings up the figure shown and asks the user, "How do you handle this case?" The user considers these questions to be nonsensical, as they involve a `rtn`



(a) CoMo: “How do you handle this case?”



(b) User (after adding `rtn` and performing the manipulation): “Like this.”

Figure 6-11: A question and corresponding answer about a queue with four nodes.

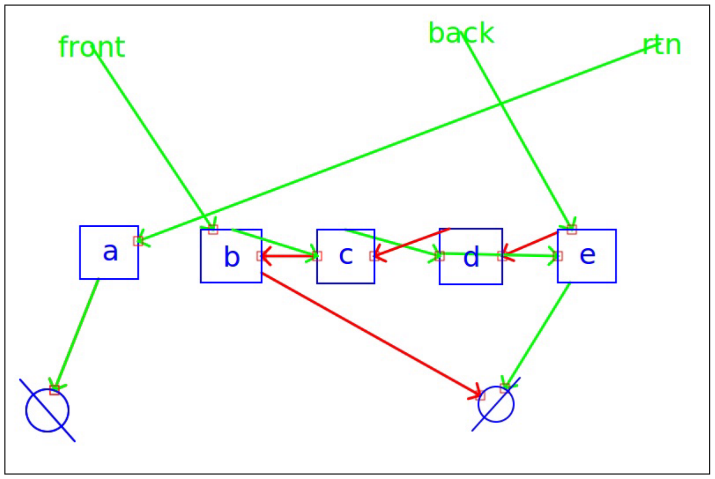


Figure 6-12: CoMo: “How do you handle this case?” User: “That does not make sense.”

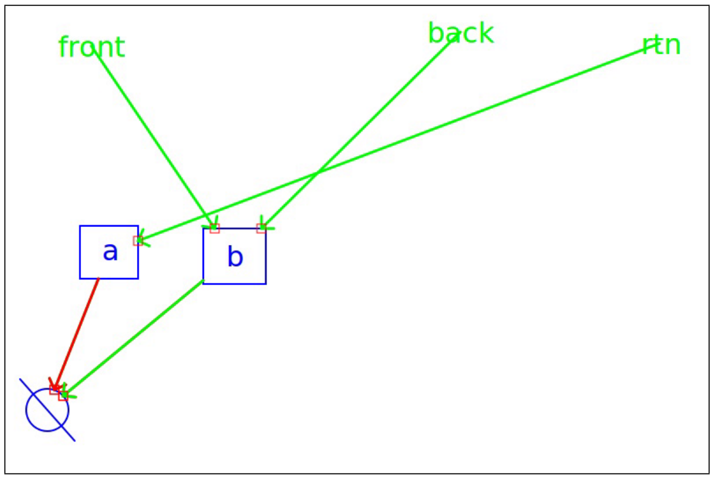


Figure 6-13: CoMo: “How do you handle this case?” User: “That does not make sense.”

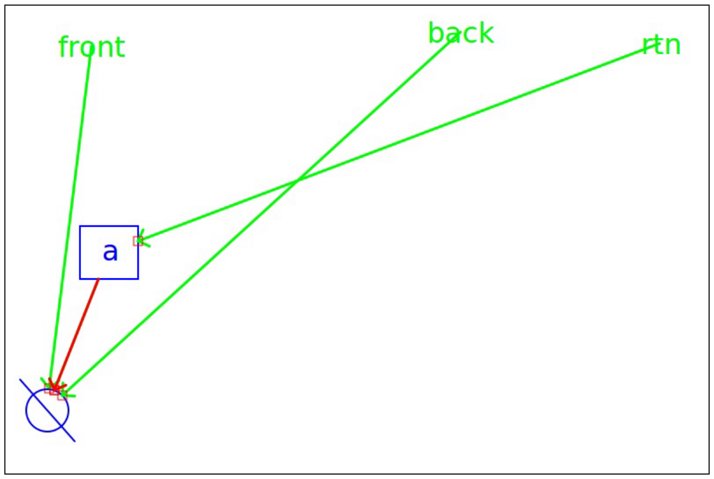
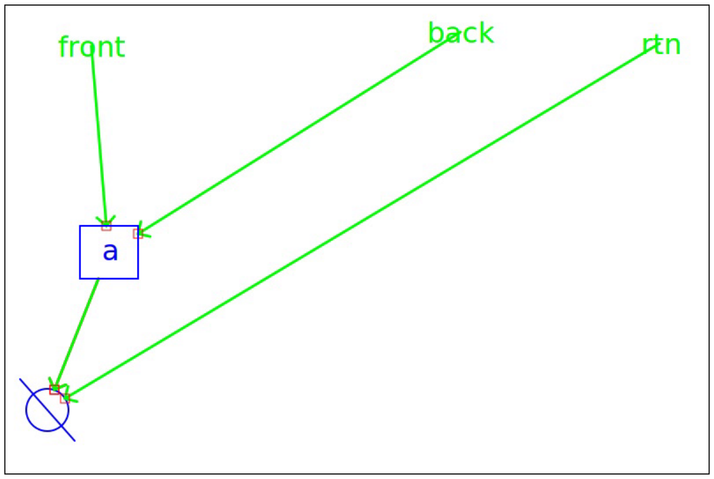
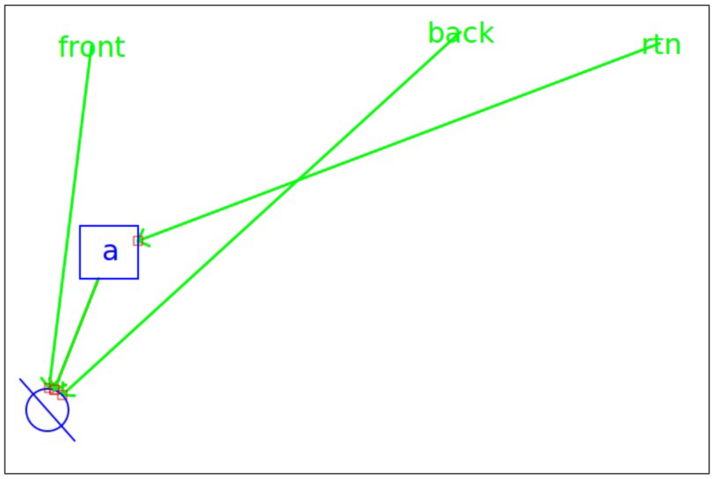


Figure 6-14: CoMo: “How do you handle this case?” User: “That does not make sense.”

pointer starting out pointing at a node, so in each case the user answers with the phrase, “That does not make sense.”



(a) CoMo: “How do you handle this case?”



(b) User: “Like this.”

Figure 6-15: CoMo asks about a queue with one element and an `rtn` pointer pointing at null.

The final two questions are seen in Fig. 6-15a–6-16. Each involves `rtn` pointing at null, so the user answers them by updating the sketch to reflect the state after the manipulation, and saying, “Like this.” After the user answers its final question, CoMo says, “I think I understand.”

The user cannot think of anything to add, so they say “Generate code.” CoMo generates the input to SPT seen in Fig. 6-17 and, after about 30 seconds, SPT generates the code seen in Fig. 6-18a, which CoMo cleans and presents to the user.

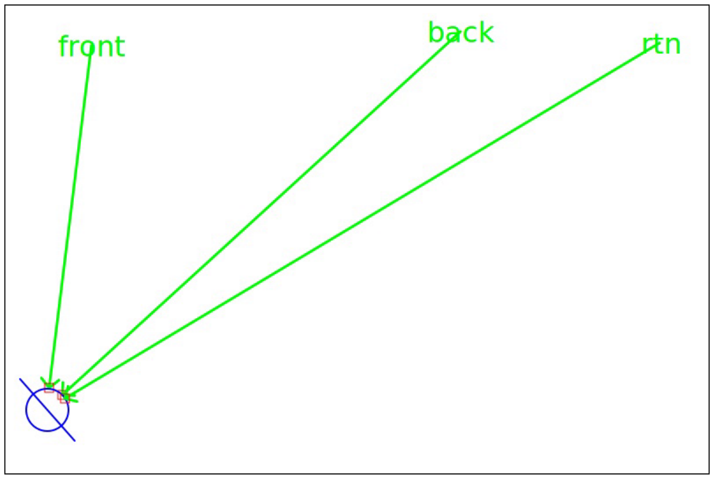


Figure 6-16: The question and answer for an empty queue. CoMo asks, "How do you handle this case?" and the user responds, "Do nothing."

```
scenario scenario2
input front -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      back -> d, d.ptr1 -> c, c.ptr1 -> b, b.ptr1 -> a, a.ptr1 -> null,
      rtn -> null
output front -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
       back -> d, d.ptr1 -> c, c.ptr1 -> b, b.ptr1 -> a null,
       rtn -> a, a.ptr0 -> null, a.ptr1 -> null ;

scenario scenario5
input back -> a, front -> a, rtn -> null, a.ptr0 -> null, a.ptr1 -> null
output back -> null, front -> null, rtn -> a, a.ptr0 -> null, a.ptr1 -> null ;

scenario scenario7
input back -> null, front -> null, rtn -> null
output back -> null, front -> null, rtn -> null ;

main(Node rtn, Node  back, Node  front) {
  Node tmp2, tmp1;
  ?=(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 6-17: The input CoMo generates for SPT.

```
void main(){
  tmp2=back;
  back=tmp1;
  rtn=tmp2;
  while(false){
    tmp2.ptr1=back;
    rtn.ptr1=front;
    tmp2=back;
    front=back;
    tmp1.ptr0=front;
  }
    if(rtn==front) front=back;
    if(front!=null) back=tmp2.ptr1;
    if(tmp1!=front) back.ptr0=tmp1;
}
```

(a) The raw code.

```
tmp2=back;
back=tmp1;
rtn=tmp2;
if(rtn==front) front=back;
if(front!=null) back=tmp2.ptr1;
if(tmp1!=front) back.ptr0=tmp1;
```

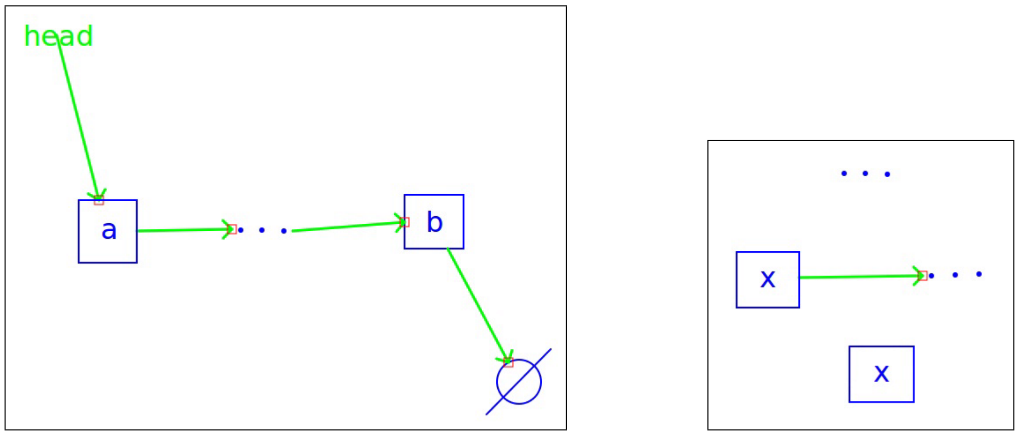
(b) The cleaned code.

Figure 6-18: The raw and cleaned synthesized code for the dequeue manipulation.

The cleaned code is seen in Fig. 6-18b. The user then tests the code on a queue with four elements, and a new queue with five elements. They are satisfied that the code works, and so the interaction is complete.

6.5 Singly Linked List Reversal

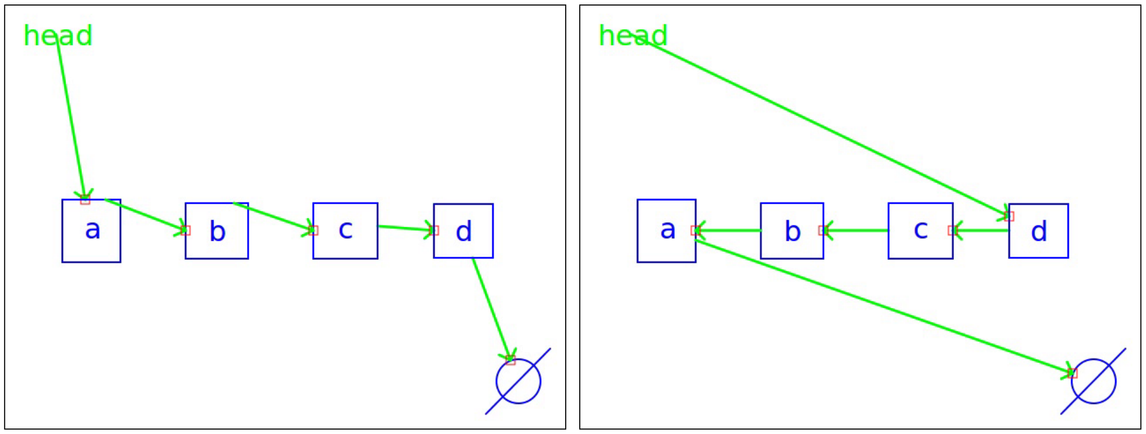
The singly linked list reversal was demonstrated in the Introduction. An abbreviated version is presented here for completeness.



(a) The user draws a general linked list. (b) The user defines the ellipsis expansion.

Figure 6-19: The inductive definition of a general list.

The user starts by asking CoMo, “Do you know what a linked list is?” CoMo responds, “No.” The user then draws a general linked list, seen in Fig. 6-19a. CoMo recognizes the ellipsis in the general list as a form of repetition, but it does not know what form the repetition takes, so it asks, “What does this expand to?” and displays a new canvas with a copy of the ellipse at the top. The user answers by drawing the inductive definition of the ellipsis expansion, seen in Fig. 6-19b, and saying, “To this or this.”



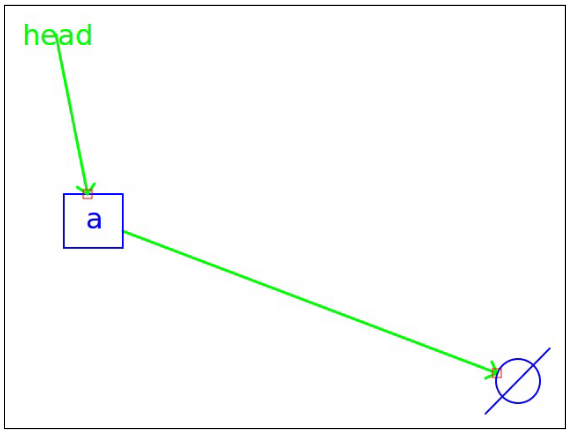
(a) CoMo: “How do you handle this case?” (b) User (after reversing the list): “Like this.”

Figure 6-20: A question about a list with four nodes.

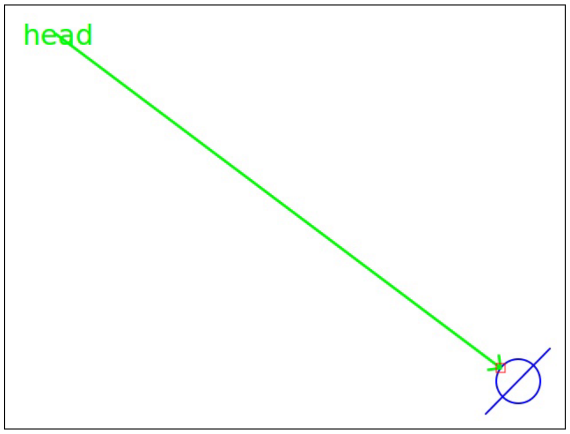
Next, CoMo asks the user what to do with a list containing four elements. CoMo generates the sketch seen in Fig. 6-20a and asks, “How do you handle this case?” The user responds by clicking and dragging the heads of pointers to reflect the state of the list after being reversed and saying, “Like this.” The result of the manipulation can bee seen in Fig. 6-20b.

Next CoMo asks about the two examples seen in Fig. 6-21. In each case, CoMo displays the example and asks, “What does this expand to?” Both of these lists look the same reversed, so the user responds only verbally, by saying, “Do nothing.” Once both questions are answered, CoMo says, “I think I understand.”

The user cannot think of any examples to add, so they say, “Generate code.” CoMo runs SPT with the input shown in Fig. 6-22, and after about 20 seconds, SPT synthesizes the code seen in Fig. 6-23a. CoMo cleans the code (seen in Fig. 6-23b) and displays it to the user.



(a) CoMo: “How do you handle this case?” User: “Do nothing.”



(b) CoMo: “How do you handle this case?” User: “Do nothing.”

Figure 6-21: Two questions about lists that look the same reversed.

```
scenario scenario2
input head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null
output head -> d, a.ptr0 -> null, b.ptr0 -> a, c.ptr0 -> b, d.ptr0 -> c ;

scenario scenario3
input head -> a, a.ptr0 -> null
output head -> a, a.ptr0 -> null ;

scenario scenario4
input head -> null
output head -> null ;

main(Node head) {
  Node tmp2, tmp1;
  ?=(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 6-22: The input CoMo generates for SPT.

```
void main(){
  tmp1=head;
  head=tmp1;
  head=tmp2;
  while(tmp1!=null){
    tmp2=head;
    tmp1=tmp1;
    head=tmp1;
    tmp1=head.next;
    head.next=tmp2;
  }
    if(false) head=tmp1;
    if(false) head=tmp1;
    if(false) head=tmp1.next;
}
```

(a) The raw code.

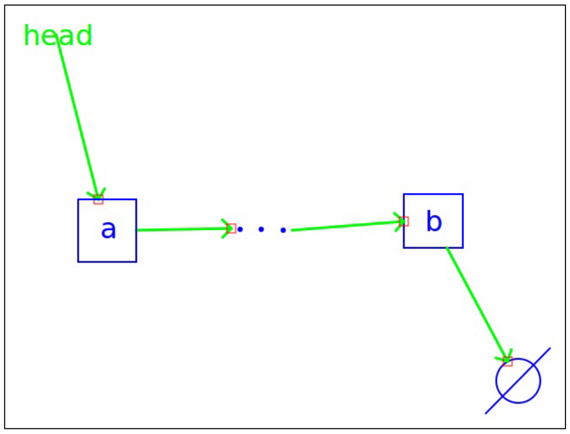
```
tmp1=head;
head=tmp2;
while(tmp1!=null){
  tmp2=head;
  head=tmp1;
  tmp1=head.next;
  head.next=tmp2;
}
```

(b) The cleaned code.

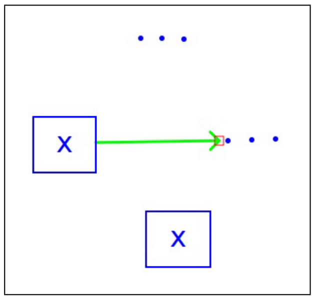
Figure 6-23: The raw and cleaned synthesized code for the reversal.

6.6 Singly Linked List Find Last

The singly linked list find last operation was fully described in chapters 4 and 5. It will differ here slightly, because CoMo just had a simple conversation about a linked list, so it will reuse this definition.



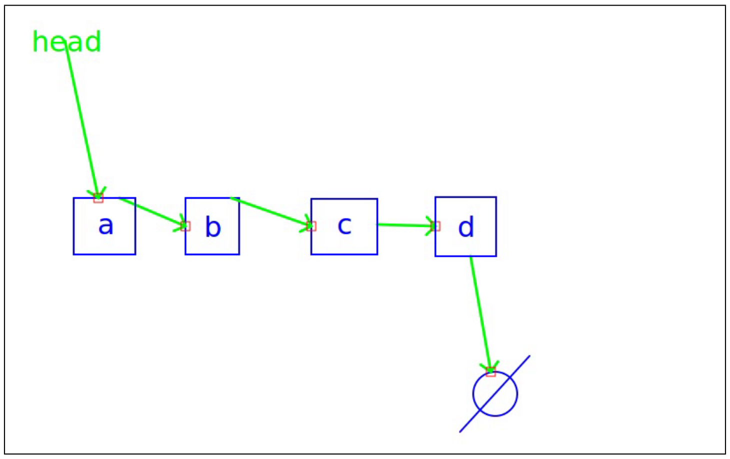
(a) The general list that CoMo remembers.



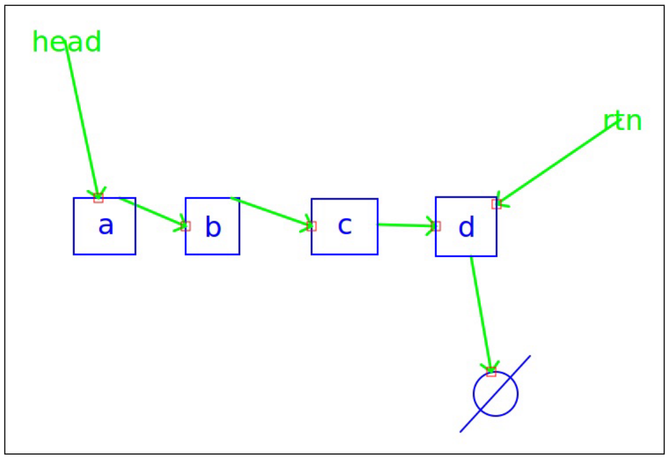
(b) The ellipsis expansion that CoMo remembers.

Figure 6-24: CoMo recalls what a linked list is when asked.

The user begins by asking CoMo, “Do you know what a linked list is?” CoMo says, “Yes.” and brings up the figures from the previous interaction, seen in Fig. 6-24. The user then asks, “Do you know what a linked list reversal is?” CoMo responds, “No.”



(a) CoMo: “How do you handle this case?”



(b) User (after adding the `rtn` pointer and updating it): “Like this.”

Figure 6-25: The first question and corresponding answer that CoMo asks.

After waiting three seconds for more input (and receiving none), CoMo asks its first question. It displays the list seen in Fig. 6-25a and asks, "How do you handle this case?" The user adds a `rtn` pointer pointing at null and moves it to point at the end of the list, seen in Fig. 6-25b. The user then says, "Like this."

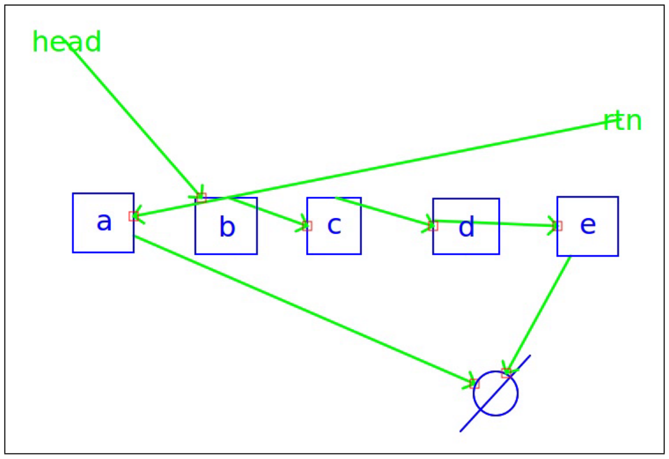


Figure 6-26: CoMo: "How do you handle this case?" User: "That does not make sense."

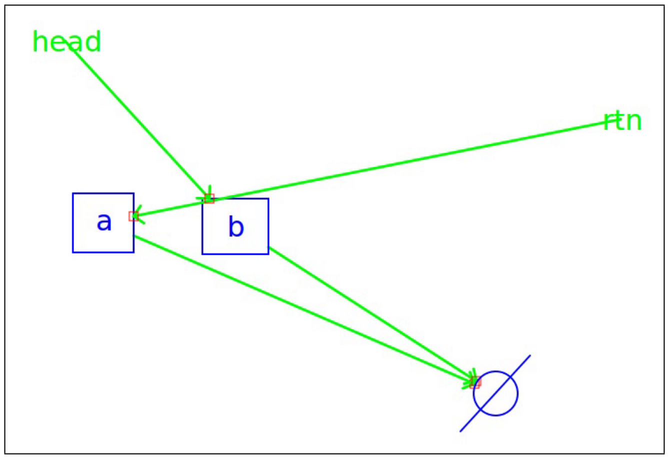


Figure 6-27: CoMo: "How do you handle this case?" User: "That does not make sense."

Next, CoMo asks about the three examples seen in Fig. 6-26, Fig. 6-27, and Fig. 6-28. As with the analogous questions in the queue dequeue manipulation (seen in Fig. 6-12, Fig. 6-13, and Fig. 6-14), the `rtn` pointer points at an element separate from the main structure. The user decides this does not make sense, since the `rtn`

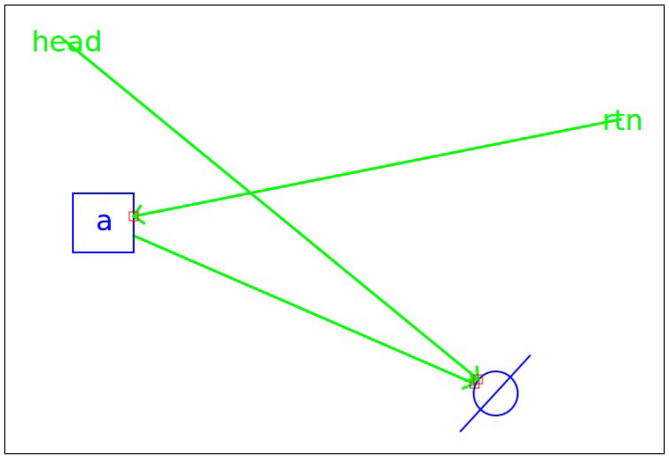


Figure 6-28: CoMo: “How do you handle this case?” User: “That does not make sense.”

pointer is meant as a return statement, and they respond to each question by saying, “That does not make sense.”

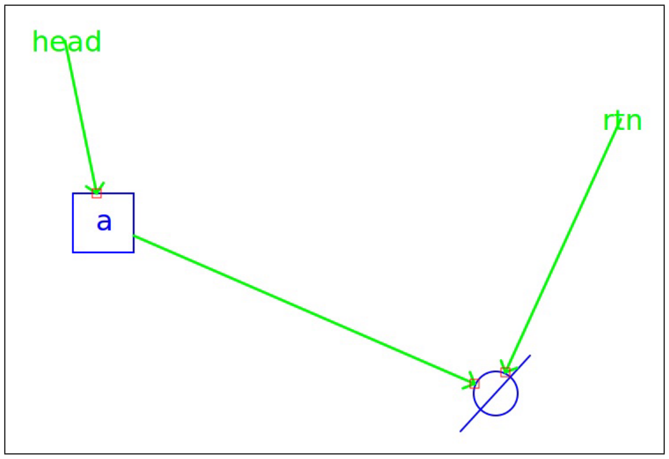
Next CoMo asks two questions about examples where `rtn` points at null, the first with a list with one node (Fig. 6-29) and the second with an empty list (Fig. 6-30). The user answers the first question by moving the `rtn` pointer to the only element in the list (Fig. 6-29b) and saying, “Like this,” and the second question by saying, “Do nothing” (Fig. 6-30). After the user answers this final question, CoMo says, “I think I understand.”

The user cannot think of any other examples to give CoMo, so they say, “Generate code.” After about 30 seconds, the code seen in Fig. 6-32a is synthesized. CoMo cleans the code to what is seen in Fig. 6-32b and displays it to the user. After correcting the simulation with five nodes (as described fully above), SPT is rerun with the input seen in Fig. 6-34, and

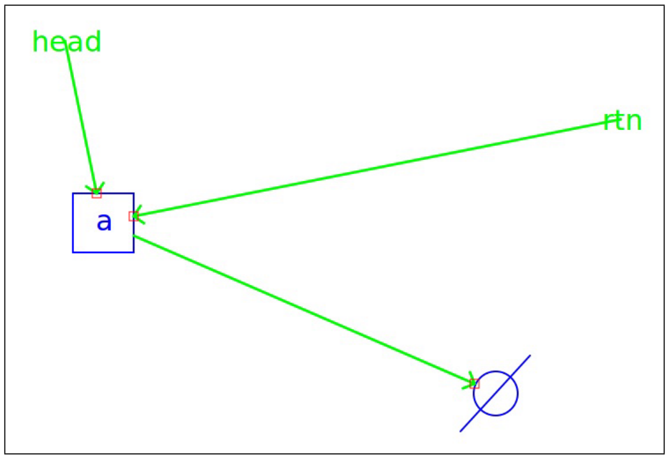
6.7 Singly Linked List Delete

This is the final demonstration of a singly linked list manipulation in this thesis. Since the singly linked list in this manipulation is not ordered, a `del` pointer is added to denote the node to be removed.

The user starts by asking CoMo, “Do you know what a singly linked list is?”



(a) CoMo: “How do you handle this case?”



(b) User (after demonstrating): “Like this.”

Figure 6-29: CoMo asks about a list with one node and `rtn` pointer pointing at null.

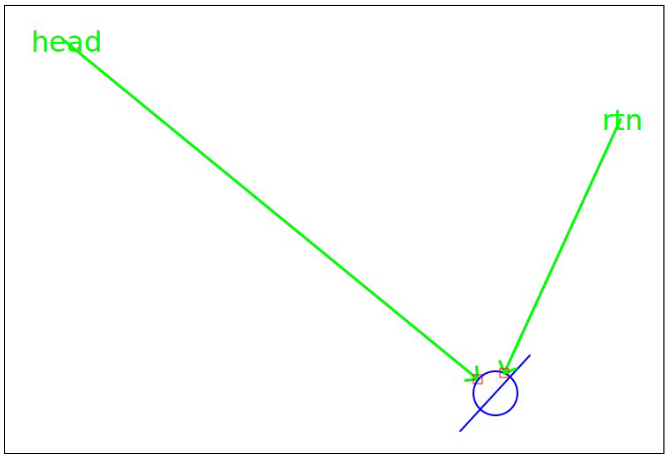


Figure 6-30: CoMo: “How do you handle this case?” User: “Do nothing.”

```
scenario scenario2
input head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      rtn -> null
output head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      rtn -> d ;

scenario scenario6
input rtn -> null, head -> a, a.ptr0 -> null
output rtn -> a, head -> a, a.ptr0 -> null ;

scenario scenario7
input rtn -> null, head -> null
output rtn -> null, head -> null ;

main(Node head, Node rtn) {
  Node tmp1, tmp2;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 6-31: The first input to SPT generated from the find-last interaction.

```
void main() {
  rtn=head;
  head=tmp1;
  tmp1=rtn;
  while(rtn!=head){
    rtn=head;
    rtn=rtn;
    rtn=head;
    rtn=tmp1;
    head=rtn;
  }
    if(tmp1!=null) tmp2=tmp1.ptr0;
    if(tmp2!=null) tmp2=tmp2.ptr0;
    if(tmp2!=null) rtn=tmp2.ptr0;
}
```

(a) The raw code.

```
rtn=head;
head=tmp1;
tmp1=rtn;
while(rtn!=head){
  rtn=tmp1;
  head=rtn;
}
if(tmp1!=null) tmp2=tmp1.ptr0;
if(tmp2!=null) tmp2=tmp2.ptr0;
if(tmp2!=null) rtn=tmp2.ptr0;
```

(b) The cleaned code.

Figure 6-32: The raw and cleaned code synthesized for the first attempt.

```
scenario scenario2
input head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      rtn -> null
output head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      rtn -> d ;

scenario scenario6
input rtn -> null, head -> a, a.ptr0 -> null
output rtn -> a, head -> a, a.ptr0 -> null ;

scenario scenario7
input rtn -> null, head -> null
output rtn -> null, head -> null ;

scenario scenario8
input head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d,
      d.ptr0 -> e, e.ptr0 -> null, rtn -> null
output head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d,
      d.ptr0 -> e, e.ptr0 -> null, rtn -> d ;

main(Node head, Node rtn) {
  Node tmp1, tmp2;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 6-33: The second input to SPT generated from the find-last interaction.

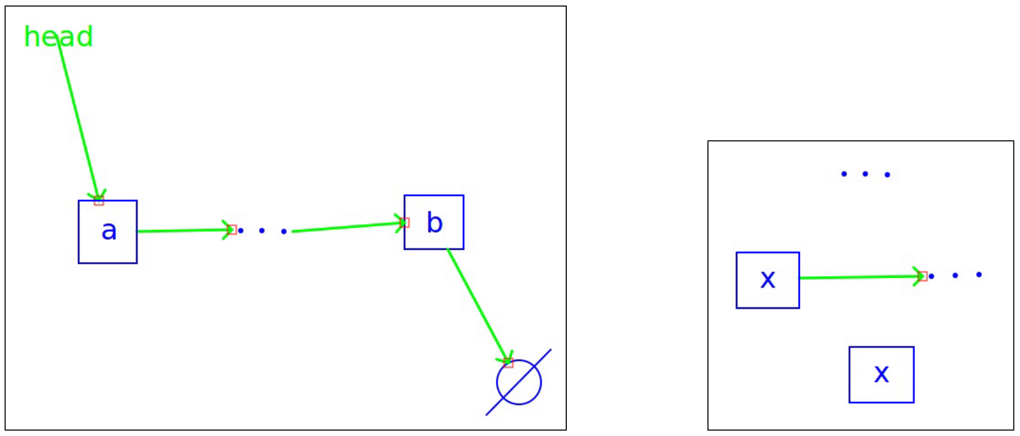
```
void main() {
  rtn=head;
  rtn=tmp2;
  tmp1=head;
  while(head!=null){
    rtn=tmp1;
    tmp2=head;
    head=head;
    tmp1=rtn;
    head=head.ptr0;
  }
    if(false) tmp2=head;
    if(rtn!=null) rtn=tmp2;
    if(tmp2!=null) head=tmp1;
}
```

(a) The raw code.

```
rtn=tmp2;
tmp1=head;
while(head!=null){
  rtn=tmp1;
  tmp2=head;
  tmp1=rtn;
  head=head.ptr0;
}
if(rtn!=null) rtn=tmp2;
if(tmp2!=null) head=tmp1;
```

(b) The cleaned code.

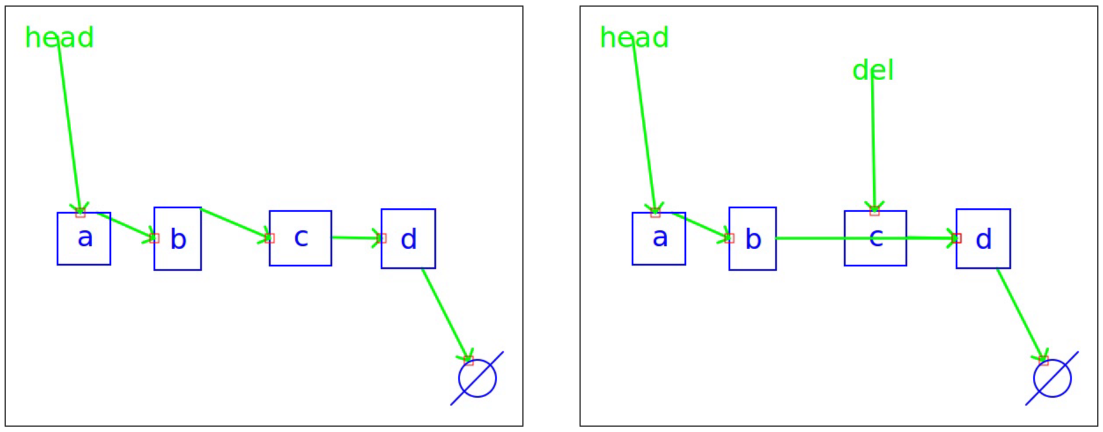
Figure 6-34: The raw and cleaned code synthesized for the second attempt.



(a) The general list that CoMo remembers. (b) The ellipsis expansion that CoMo remembers.

Figure 6-35: CoMo recalls what a linked list is when asked.

CoMo responds, "Yes" and brings up the inductive singly linked list definition from the previous examples, seen in Fig. 6-35.

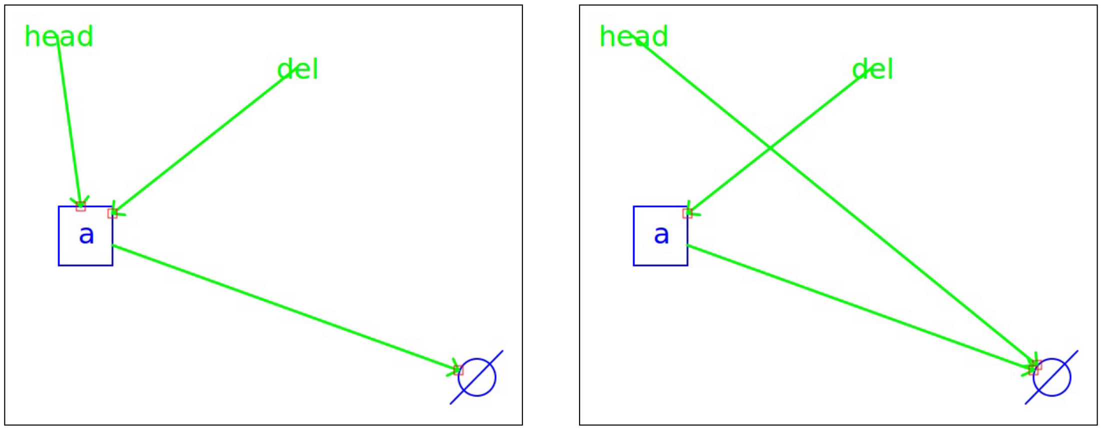


(a) CoMo: "How do you handle this case?" (b) User (after adding `del` and performing the manipulation): "Like this."

Figure 6-36: CoMo asks a question about a singly-linked list with four nodes.

The user does not say or draw anything, so after waiting three seconds, CoMo asks its first question about a list with four nodes, seen in Fig. 6-36, by generating a list and saying, "How do you handle this case?" The user responds by drawing a `del` pointer pointing at the `c` node, removing it from the list by reassigning `b`'s pointer to point at `d`, and saying, "Like this."

Next, CoMo asks about a list with one node. Since it saw a `del` pointer in the



(a) CoMo: “How do you handle this case?” (b) User (after performing the manipulation): “Like this.”

Figure 6-37: CoMo asks a question about a singly-linked list with one node.

previous example, it adds it to point at the only node in the sketch it generates. As before, CoMo asks, “How do you handle this case?” The user responds by moving the `head` pointer to point at the null symbol and saying, “Like this.”

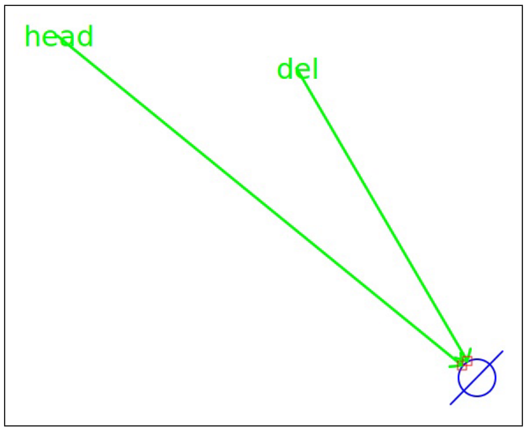
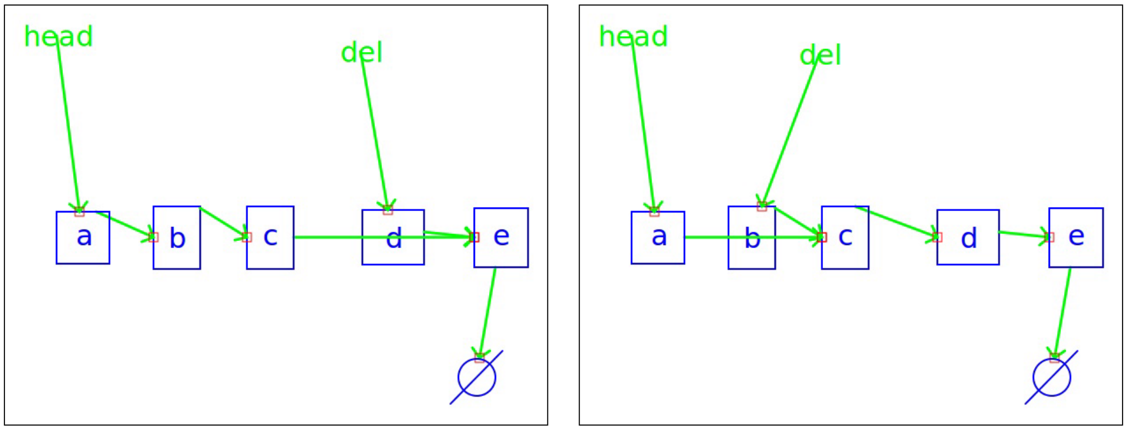


Figure 6-38: CoMo: “How do you handle this case?” User: “Do nothing.”

CoMo asks its final question about an empty list. It generates the sketch and says, “How do you handle this case?” The user responds only verbally, by saying, “Do nothing.” CoMo has finished asking all of its questions, so it says, “I think I understand.”

At this point, the user is unconvinced that CoMo has received enough examples to generate code correctly. Instead of initiating code synthesis by saying “Generate code,” the user says, “Generate a structure with five nodes.” CoMo does, and the



(a) The user removes the next-to-last node. (b) The user removed the next-to-first node.

Figure 6-39: The user decides to give two more demonstrations on structures with five nodes.

user proceeds to demonstrate how to remove the next to last element of a five-element list (Fig. 6-39a). Once done with that, the user says again, "Generate a list with five nodes." After CoMo does, the user demonstrates how to remove the second element from this new five-element list (Fig. 6-39b). Now the user is satisfied that enough examples have been given, and so they say, "Generate code." CoMo responds, "I will try."

CoMo generates the input seen in Fig. 6-40 from the examples above. After about ten seconds, SPT produces the code seen in Fig. 6-41a. CoMo cleans the code (Fig. 6-41b) and displays it to the user. After testing this out on a few examples, the user is satisfied that the code works, and the interaction is complete.

6.8 Ordered Doubly Linked List Insertion

The ordered, doubly linked list insertion is the most complicated manipulation that CoMo can handle. Ordered insertions proceed based on the values of the nodes involved. In the following example, no concrete values are used in the interaction with the user. As mentioned in the previous chapter, CoMo is able to handle constraints abstractly; specifying the list is ordered consists of drawing a line of the form, "`a < b < c`."

```
scenario scenario2
input del -> c,
      head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null
output del -> c,
       head -> a, a.ptr0 -> b, b.ptr0 -> d, c.ptr0 -> d, d.ptr0 -> null ;

scenario scenario3
input del -> a, head -> a, a.ptr0 -> null
output del -> a, head -> null, a.ptr0 -> null ;

scenario scenario4
input del -> null, head -> null
output del -> null, head -> null ;

scenario scenario5
input del -> d, head -> a,
      a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> e, e.ptr0 -> null
output del -> d, head -> a,
       a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> e, d.ptr0 -> e, e.ptr0 -> null ;

scenario scenario6
input del -> b, head -> a,
      a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> e, e.ptr0 -> null
output del -> b, head -> a,
       a.ptr0 -> c, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> e, e.ptr0 -> null ;

main(Node del, Node  head) {
  Node tmp2, tmp1;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 6-40: The input to SPT produced by CoMo.

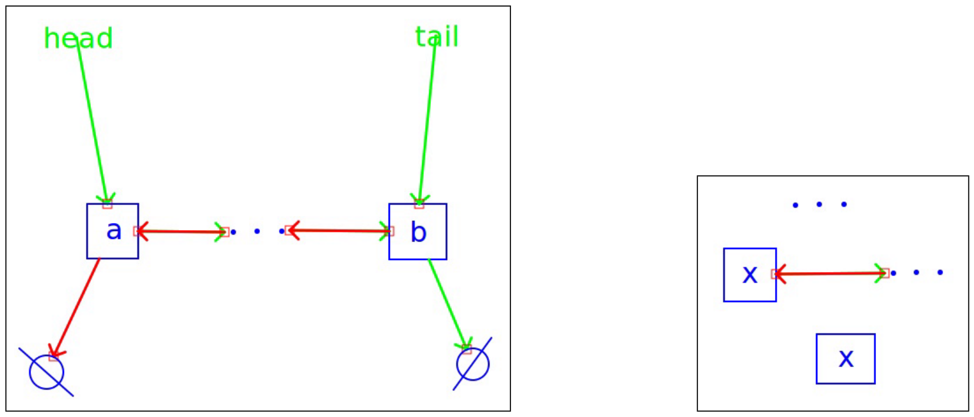
```
void main(){
  tmp1=head;
  head=del;
  del=tmp1;
  while(head!=del){
    tmp2=del;
    del=del.ptr0;
    del=del;
    del=del;
    del=del;
  }
    if(tmp2==null) tmp1=tmp2;
    if(true) head=tmp1;
    if(tmp2!=null) tmp2.ptr0=del.ptr0;
}
```

(a) The raw code.

```
tmp1=head;
head=del;
del=tmp1;
while(head!=del){
  tmp2=del;
  del=del.ptr0;
}
if(tmp2==null) tmp1=tmp2;
head=tmp1;
if(tmp2!=null) tmp2.ptr0=del.ptr0;
```

(b) The cleaned code.

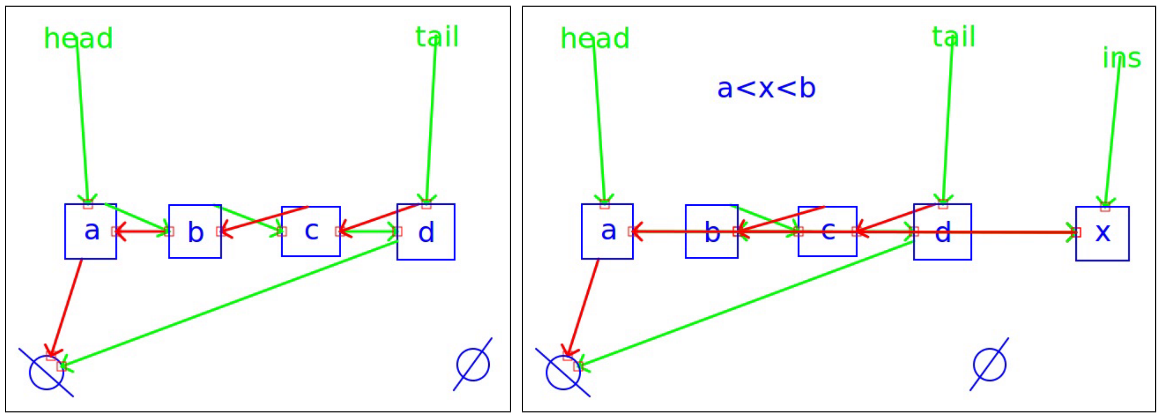
Figure 6-41: The raw and cleaned synthesized code for the delete manipulation.



(a) A general ordered doubly linked list. (b) The inductive ellipsis expansion.

Figure 6-42: The inductive definition of a general linked list.

The user starts by asking, "Do you know what a doubly linked list is?" CoMo responds, "No," so the user draws a general doubly linked list (Fig. 6-42a) and, when asked as in the previous examples to do so by CoMo, the inductive ellipsis expansion (Fig. 6-42b).[2]



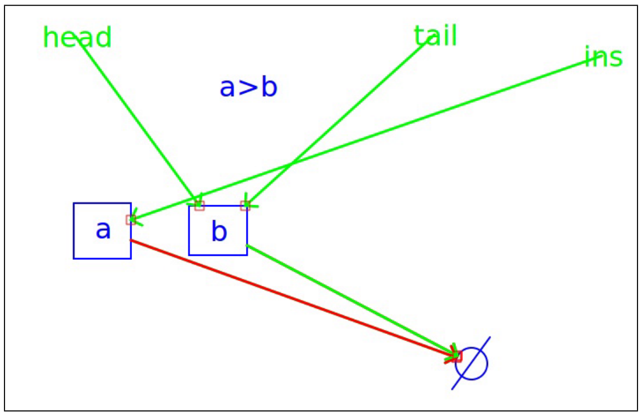
(a) CoMo: "How do you handle this case?" (b) User (after adding `ins` and performing the manipulation): "Like this."

Figure 6-43: CoMo's first question (and the corresponding answer) about a list with four nodes.

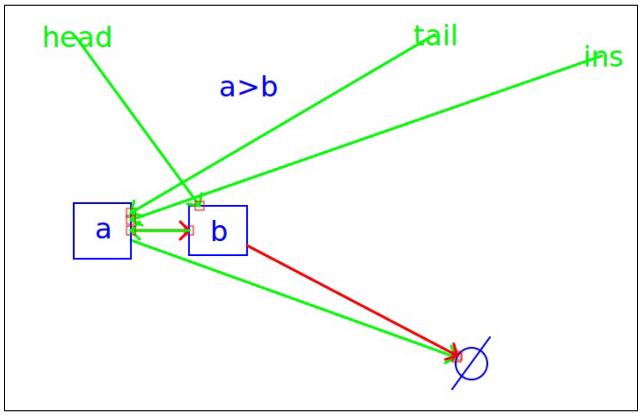
CoMo's first question is about what to do with a list of length four. Fig. 6-43 shows the question CoMo asks and the answer the user gives. CoMo does not yet know that the list is ordered, and it does not know that the manipulation requires

[2]Note that, due to the geometry heuristic mentioned in the user study findings in Chapter 3, CoMo recognizes the pointers as their correct types on the first try. This classification can be seen in the colors CoMo displays the pointers with.

an auxiliary structure, so the user answers by adding the auxiliary structure (an `ins` pointer pointing at a separate `x` node) and a constraint (seen in Fig. 6-43b: "`a<x<b`"), and updating the list to contain `x` where the constraint dictates it should be: between the `a` and `b` nodes. To signal they are done answering the question, the user says, "Like this."



(a) CoMo: "How do you handle this case?"

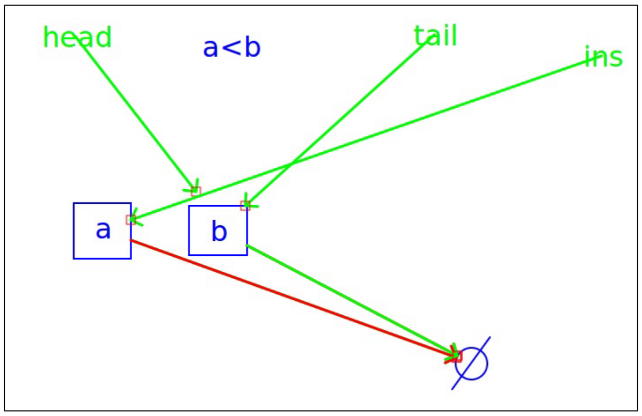


(b) User (after demonstrating): "Like this."

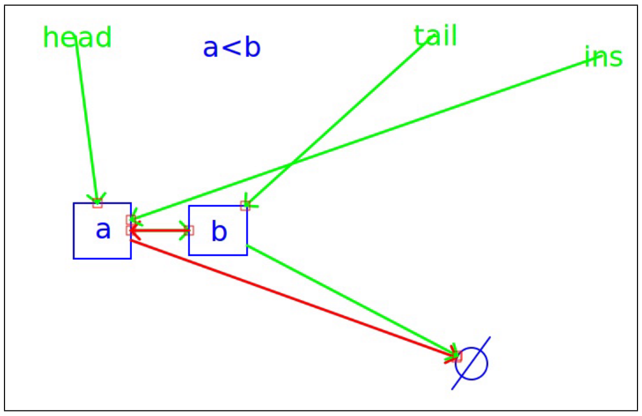
Figure 6-44: CoMo asks about how to insert a node into a list with one node when the node to insert is less than the node in the list.

Now CoMo is aware of the constraint and the fact that multiple structures are needed. The next question it asks is about a list with one node and a single node to insert where the list's node is less than the node to insert. Fig. 6-44a shows the structure presented to the user as CoMo asks, "How do you handle this case?" The user has decided that the list is ordered smallest to largest (as indicated by the previous constraint "`a<x<b`"), so the user updates the sketch to reflect this by putting

`b` in front of `a`. When done, they say, "Like this."



(a) CoMo: "How do you handle this case?"



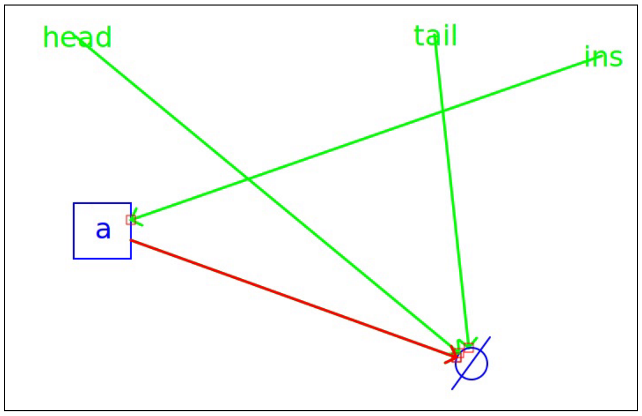
(b) User (after demonstrating): "Like this."

Figure 6-45: CoMo asks about how to insert a node into a list with one node when the node to insert is greater than the node in the list.

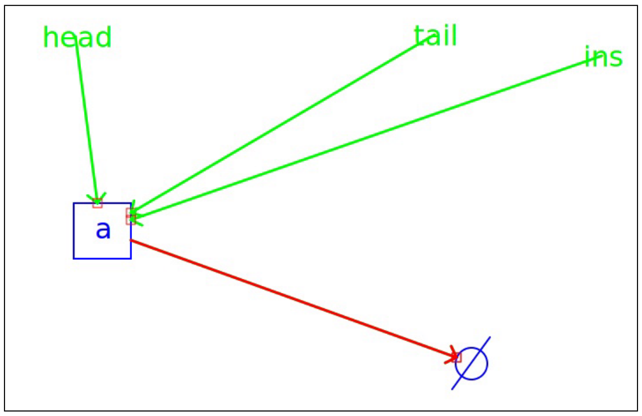
The next question CoMo asks, seen in Fig. 6-45, is similar. The only difference is that it asks about what to do when the node to add is *larger* than the node in the list. As seen in Fig. 6-45b, the user places the node to add behind the node in the list, and says, "Like this."

Next, CoMo asks about how to add a node to an empty list. As seen in Fig. 6-46, the user answers by placing the node in the list and saying, "Like this."

The final three questions that CoMo asks are about how to add zero nodes to lists of length four, one, and zero. For each question, CoMo generates the sketch, presents it to the user, and says, "How do you handle this case?" and the user responds by saying, "Do nothing." Once these are answered, CoMo says, "I think I understand."



(a) CoMo: "How do you handle this case?"



(b) User (after demonstrating): "Like this."

Figure 6-46: CoMo asks how to insert a node into an empty list, and the user responds.

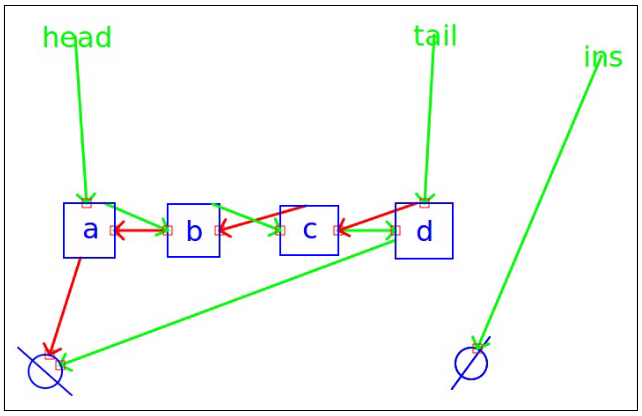


Figure 6-47: CoMo: "How do you handle this case?" User: "Do nothing."

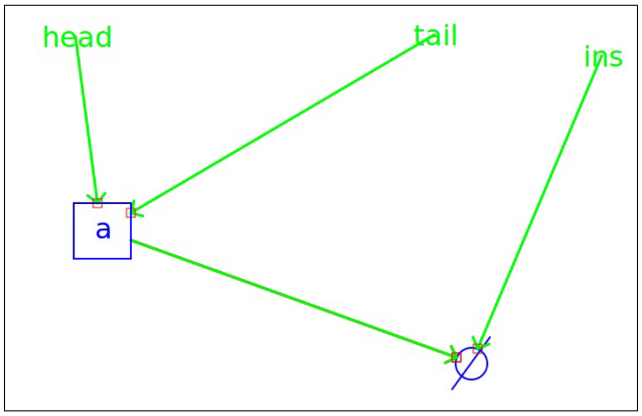


Figure 6-48: CoMo: “How do you handle this case?” User: “Do nothing.”

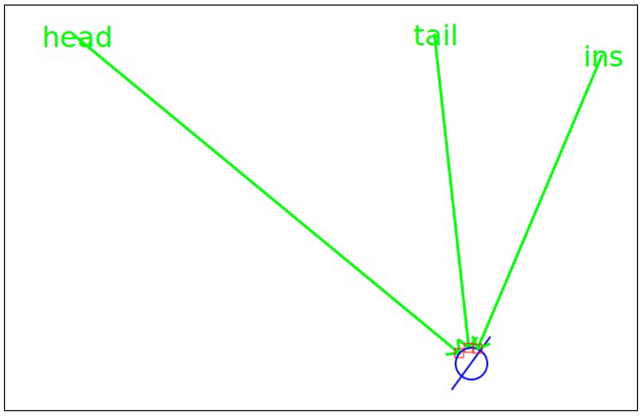


Figure 6-49: CoMo: “How do you handle this case?” User: “Do nothing.”

```
scenario scenario2
input ins -> x, x.ptr0 -> null, x.ptr1 -> null,
      head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      tail -> d, d.ptr1 -> c, c.ptr1 -> b, b.ptr1 -> a, a.ptr1 -> null,
      a.val -> 5, x.val -> 6, b.val -> 7, c.val -> 8, d.val -> 9
output ins -> x, x.ptr0 -> b, x.ptr1 -> a,
       head -> a, a.ptr0 -> x, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
       tail -> d, d.ptr1 -> c, c.ptr1 -> b, b.ptr1 -> x, a.ptr1 -> null ;

scenario scenario3
input ins -> b, b.ptr0 -> null, b.ptr1 -> null,
      head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null,
      a.val -> 5, b.val -> 6
output head -> a, a.ptr0 -> b, b.ptr0 -> null,
       tail -> b, b.ptr1 -> a, a.ptr1 -> null, ins -> b ;

scenario scenario4
input ins -> b, b.ptr0 -> null, b.ptr1 -> null,
      head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null,
      a.val -> 6, b.val -> 5
output head -> b, b.ptr0 -> a, a.ptr0 -> null,
       tail -> a, a.ptr1 -> b, b.ptr1 -> null, ins -> b ;

scenario scenario5
input ins -> a, a.ptr0 -> null, a.ptr1 -> null,
      head -> null, tail -> null, a.val -> 5
output ins -> a,
       head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null ;

scenario scenario6
input head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null,
      ins -> null, a.val -> 5
output head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null,
       ins -> null ;

scenario scenario7
input ins -> null, head -> null, tail -> null
output ins -> null, head -> null, tail -> null ;

data_selector val;

main(Node head, Node tail, Node ins) {
  Node tmp2, tmp1;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 6-50: CoMo's first try at producing SPT input. After one hour, CoMo halts SPT and retries.

```
scenario scenario2
input ins -> x, x.ptr0 -> null, x.ptr1 -> null,
      head -> a, a.ptr0 -> b, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
      tail -> d, d.ptr1 -> c, c.ptr1 -> b, b.ptr1 -> a, a.ptr1 -> null,
      a.val -> 5, x.val -> 6, b.val -> 7, c.val -> 8, d.val -> 9
output ins -> x, x.ptr0 -> b, x.ptr1 -> a,
       head -> a, a.ptr0 -> x, b.ptr0 -> c, c.ptr0 -> d, d.ptr0 -> null,
       tail -> d, d.ptr1 -> c, c.ptr1 -> b, b.ptr1 -> x, a.ptr1 -> null ;

scenario scenario3
input ins -> b, b.ptr0 -> null, b.ptr1 -> null,
      head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null,
      a.val -> 5, b.val -> 6
output head -> a, a.ptr0 -> b, b.ptr0 -> null,
       tail -> b, b.ptr1 -> a, a.ptr1 -> null, ins -> b ;

scenario scenario4
input ins -> b, b.ptr0 -> null, b.ptr1 -> null,
      head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null,
      a.val -> 6, b.val -> 5
output head -> b, b.ptr0 -> a, a.ptr0 -> null,
       tail -> a, a.ptr1 -> b, b.ptr1 -> null, ins -> b ;

scenario scenario5
input ins -> a, a.ptr0 -> null, a.ptr1 -> null,
      head -> null, tail -> null, a.val -> 5
output ins -> a,
       head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null ;

scenario scenario6
input head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null,
      ins -> null, a.val -> 5
output head -> a, a.ptr0 -> null, tail -> a, a.ptr1 -> null,
       ins -> null ;

scenario scenario7
input ins -> null, head -> null, tail -> null
output ins -> null, head -> null, tail -> null ;

data_selector val;

main(Node head, Node tail, Node ins) {
  Node tmp2, tmp1;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(8)
}
```

Figure 6-51: CoMo's second try at running SPT. This produces code after anywhere from 15 minutes to 2 hours.

```
void main(){
  tmp1=tail;
  tmp2=tmp2;
  tmp1=tail;
  while(tmp1!= null && tmp2!= null && tmp1.val < tmp2.val){
    head.ptr0=tail;
    ins.ptr0=tmp1.ptr1;
    tail.ptr1=head.ptr1;
    ins.ptr1=tmp2.ptr1;
    tail=tmp2;
  }
    if(ins!= null && head!= null && ins.val < head.val) head=head.ptr0;
    if(head==tmp2) head=ins;
    if(head!= null && ins!= null && head.val < ins.val) tmp1=head.ptr0;
    if(head!= null && ins!= null && head.val < ins.val) head.ptr0=ins;
    if(head!= null && tmp1!= null && head.val < tmp1.val) tmp1.ptr1=ins;
    if(tmp1==null) tail=ins;
    if(head!= null && ins!= null && head.val < ins.val) ins.ptr1=head;
    if(ins!= null && tail!= null && ins.val < tail.val) ins.ptr0=tmp1;
}
```

(a) The code SPT produces after running SPT the second time.

```
tmp1=tail;
while(tmp1!= null && tmp2!= null && tmp1.val < tmp2.val){
  head.ptr0=tail;
  ins.ptr0=tmp1.ptr1;
  tail.ptr1=head.ptr1;
  ins.ptr1=tmp2.ptr1;
  tail=tmp2;
}
if(ins!= null && head!= null && ins.val < head.val) head=head.ptr0;
if(head==tmp2) head=ins;
if(head!= null && ins!= null && head.val < ins.val) tmp1=head.ptr0;
if(head!= null && ins!= null && head.val < ins.val) head.ptr0=ins;
if(head!= null && tmp1!= null && head.val < tmp1.val) tmp1.ptr1=ins;
if(tmp1==null) tail=ins;
if(head!= null && ins!= null && head.val < ins.val) ins.ptr1=head;
if(ins!= null && tail!= null && ins.val < tail.val) ins.ptr0=tmp1;
```

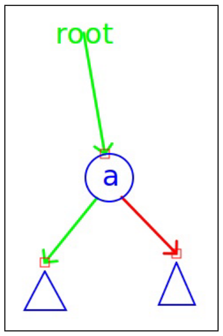
(b) The code after CoMo cleans it.

Figure 6-52: The code produced after the second try.

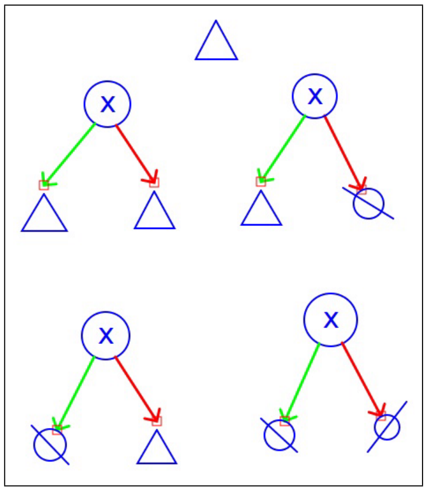
The user is satisfied with the range of examples that CoMo has, so they say, “Generate code.” CoMo responds “I will try.” CoMo generates the input seen in Fig. 6-50 and begins synthesis. After an hour[3], CoMo halts SPT and tries synthesis again with the input seen in Fig. 6-51. After another hour, SPT produces the code seen in Fig. 6-52a and CoMo cleans it to be the code seen in Fig. 6-52b. After running this code on the examples given and on a new example containing a list with five nodes, the user is satisfied that the code is correct, and the interaction is complete.

6.9 Binary Tree Left Rotation

The final manipulation shown in this chapter is the binary tree left rotation. The kind of left rotation that CoMo talks about it one that rotates about the root.



(a) A general binary tree.



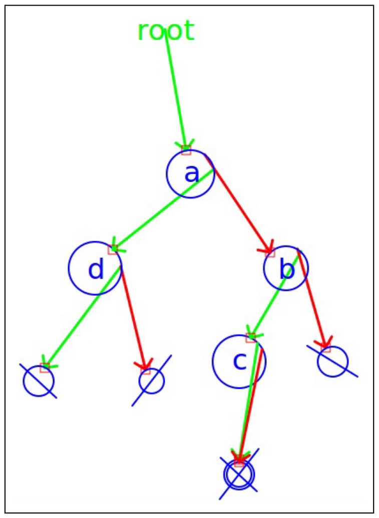
(b) An inductive definition of the triangle.

Figure 6-53: The inductive definition of a general binary tree.

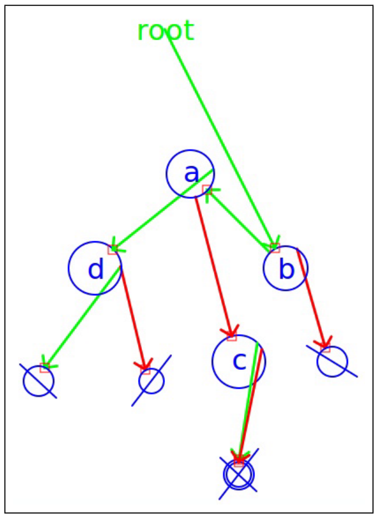
The user starts the conversation by asking, “Do you know what a binary tree is?” CoMo responds, “No,” so the user draws the general binary tree seen in Fig. 6-53a. CoMo recognizes the triangle symbol as a form of repetition but does not know the form the repetition takes, so it displays a new canvas with a copy of the triangle

[3]Recall from the previous section that CoMo gives SPT an hour to run on the first try. If SPT has not completed synthesizing code in that time, it retries with a broader CFG.

at the top and asks, “What does this expand to?” The user answers with the four examples seen in Fig. 6-53b.



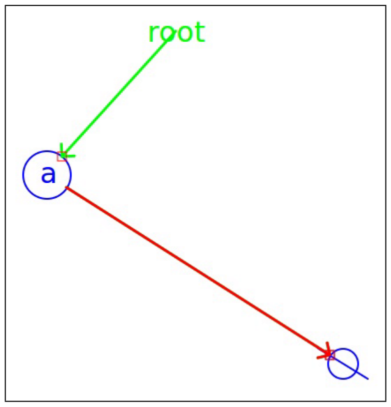
(a) CoMo: “How do you handle this case?”



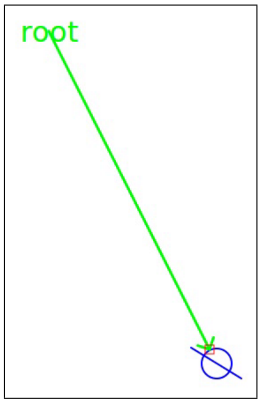
(b) User (after demonstrating): “Like this.”

Figure 6-54: CoMo asks a question about a tree with four nodes.

CoMo asks its first question about a tree with four nodes. It displays the sketch in Fig. 6-54a and says, “How do you handle this case?” The user responds by reassigning the pointers to configure the tree to be in its rotated state and says, “Like this.”



(a) CoMo: “How do you handle this case?” User: “Do nothing.”



(b) CoMo: “How do you handle this case?” User: “Do nothing.”

Figure 6-55: CoMo asks two questions about trees that do not have enough nodes to be rotated.

Next, CoMo asks questions about the two trees seen in Fig. 6-55 that do not have enough nodes to be rotated. For each question, CoMo asks, “How do you handle this case?" and the user responds "Do nothing." Once these questions are answered, CoMo says, "I think I understand."

```
scenario scenario2
input root -> a, a.ptr0 -> d, a.ptr1 -> b, b.ptr0 -> c, b.ptr1 -> null,
      c.ptr0 -> null, c.ptr1 -> null, d.ptr0 -> null, d.ptr1 -> null
output root -> b, a.ptr0 -> d, a.ptr1 -> c, b.ptr0 -> a, b.ptr1 -> null,
       c.ptr0 -> null, c.ptr1 -> null, d.ptr0 -> null, d.ptr1 -> null ;

scenario scenario3
input root -> a, a.ptr0 -> null, a.ptr1 -> null
output root -> a, a.ptr0 -> null, a.ptr1 -> null ;

scenario scenario4
input root -> null
output root -> null ;

main(Node root) {
  Node tmp2, tmp1;
  ??(3)
  while (**) {
    ??(5)
  }
  ?=(3)
}
```

Figure 6-56: The input to SPT produced by CoMo.

The user is satisfied that enough examples have been given and says, "Generate code." CoMo responds, "I will try" and generates the input to SPT seen in Fig. 6-56. After about ten seconds, SPT synthesizes the code seen in Fig. 6-57a and CoMo cleans it to be the code seen in Fig. 6-57b and displays it to the user. After testing it out on a hand-drawn tree with five nodes, the user is satisfied that the code is correct, and the interaction is complete.

```
void main(){
  tmp2=root;
  tmp1=tmp1;
  tmp1=tmp1;
  while(tmp2!=null){
    tmp2=tmp1;
    tmp2=tmp2;
    tmp2=tmp1;
    tmp1=root.ptr1;
    root=root;
  }
    if(tmp1!=null) root.ptr1=tmp1.ptr0;
    if(tmp1!=null) tmp1.ptr0=root;
    if(tmp1!=null) root=tmp1;
}
```

(a) The raw code.

```
tmp2=root;
while (tmp2!=null) {
  tmp2=tmp1;
  tmp1=root.ptr1;
}
if(tmp1!=null) root.ptr1=tmp1.ptr0;
if(tmp1!=null) tmp1.ptr0=root;
if(tmp1!=null) root=tmp1;
```

(b) The cleaned code.

Figure 6-57: The raw and cleaned synthesized code for the left rotation.

Chapter 7

Contributions & Future Work

The contributions presented fall in three categories. The first contribution includes the findings from a user study (Chapter 3). In this study, I determined that:

1. Teachers tend to ask students what they know before starting to explain something.
2. Teachers tend to describe manipulations in terms of before and after states, but sometimes explain the process of the manipulation in detail.
3. Visual vocabularies are easily learned, understood, and used by people.
4. Teachers sometimes used an unannounced, unexplained drawn shorthand that was immediately understood by students.
5. Interactions were mixed-initiative in the sense that teachers generally initiated interactions by explaining a manipulation, but students also initiated interactions by asking questions.
6. One subject *drew* a question when verbal communication repeatedly failed.

The second contribution is CoMo, the first conversation-to-code smart whiteboard system capable of holding a mixed-initiative symmetric-multimodal interaction about 50 manipulations on 8 structures and correctly synthesizing functioning C code. The structures and manipulations CoMo can handle are enumerated here:

1. Stack
    (a) push
    (b) pop
    (c) poll
2. Queue
    (a) enqueue
    (b) dequeue
    (c) peek
3. Singly Linked List
    (a) insert
    (b) delete
    (c) reverse
    (d) append
    (e) prepend
    (f) remove first
    (g) remove last
    (h) find first
    (i) find last
4. Doubly Linked List
    (a) insert
    (b) delete
    (c) reverse
    (d) append
    (e) prepend
    (f) remove first
    (g) remove last
    (h) find first
    (i) find last
5. Ordered, Singly Linked List
    (a) insert
    (b) reverse
    (c) remove minimum
    (d) remove maximum
    (e) find minimum
    (f) find maximum
6. Ordered, Doubly Linked List:
    (a) insert
    (b) reverse
    (c) remove minimum
    (d) remove maximum
    (e) find minimum
    (f) find maximum
7. Binary Tree
    (a) left rotation
    (b) right rotation
    (c) left-right rotation
    (d) left-left rotation

    (e) right-left rotation
    (f) right-right rotation

8. Binary Search Tree
    (a) left rotation
    (b) right rotation
    (c) left-right rotation
    (d) left-left rotation
    (e) right-left rotation
    (f) right-right rotation
    (g) find minimum
    (h) find maximum

The third and final contribution is the novel mixed-initiative code-generation framework (MICGF) that CoMo implements to engage in interactions about data structure manipulations (Chapter 5), which proceeds in five basic steps:

1. The user initiates the interaction by:
    - asking CoMo if it knows what a specific data structure or manipulation is; or
    - demonstrating a concrete example on a drawn structure; or
    - defining the general data structure to talk about.
2. CoMo asks questions (and interprets answers) about:
    - an inductive general structure definition;
    - a structure with four, one, and zero nodes; or—if the manipulation consists of two structures: a *main* and an *auxiliary* one—the power set of comparisons between the main structure with four, one, and zero nodes and the auxiliary structure with one and zero nodes; and
    - if the manipulation is on an ordered structure that (as above) involves multiple structures (e.g., an ordered linked list insertion)—the power set of constraint comparisons between single-node structures where the main structure's node is less than and greater than the auxiliary structure's node.
3. CoMo uses SPT [28] to synthesize code by:

    (a) generating input based on examples the user drew;
    (b) selecting a control flow graph (CFG) to use;
    (c) running SPT; and
    (d) cleaning code and displaying it to the user.

4. CoMo animates the synthesized code on examples until the user is satisfied it is correct.

5. If the code is incorrect, CoMo reruns synthesis (step 3) with new corrected examples.

It improves upon using a text-based code synthesis system alone by:

1. Removing rudimentary dead code.

2. Engaging in a mixed-initiative, symmetric-multimodal interaction and converting data structure manipulation diagrams into the input and output states that the code synthesis system requires.

3. Providing feedback about what kinds of scenarios to provide, namely that the following examples suffice for SPT [28] to synthesize correct code for many simple data structure manipulations:
    (a) Examples containing structures with four, one, and zero nodes.
    (b) Examples enumerating all size comparisons between the *main* structure when it has 4, 1, and 0 nodes and any *auxiliary* structures with one or no nodes.
    (c) Examples comparing single-node structures with different values in multiple structure scenarios where the structure is ordered.

4. Removing the need to provide a code outline (called a *control flow graph*) to the code synthesis system by automatically selecting one.

5. Aiding the user in verifying synthesized code's correctness by animating it, thereby facilitating the user's own checking of the code.

7.1 Future Work

Several features would greatly increase CoMo's abilities. This remainder of this chapter discusses these extensions and lays out plans to create them.

7.1.1 Recursion

The first feature to add would be the ability to handle recursive manipulations. Adding this functionality requires two additions:

1. SPT (or another code synthesis system) must be able to support specifying and synthesizing recursive functions.

2. CoMo must accept user input for recursive functions.

For example, to describe a binary search tree insertion, CoMo could accept a verbal command from the user like, "This is a recursive manipulation." Then, the user could demonstrate the base case—inserting a node into an empty tree—followed by an insertion on a more complex tree, saying at each level of the tree, "Recurse here."

7.1.2 Composing Manipulations

CoMo should be able to compose previously explained manipulations. Take an AVL tree insertion as an example. An AVL tree is a form of self-balancing binary search tree (BST). Insertions in AVL trees consist of three parts:

1. Inserting the node as one would with a standard (i.e., non-self-balancing) BST.

2. Bookkeeping to determine whether the tree needs to be balanced.

3. Balancing the tree (if needed) with tree rotations.

Tree rotations are manipulations that change the configuration of a binary tree without altering its in-order property. In AVL trees, the balancing step is one of four manipulations. Assuming CoMo had already been taught these manipulations—the

BST insertion and the four BST rotations—describing the AVL tree insertion would be a matter of describing:

1. How to compose previously-learned manipulations.

2. Which manipulations to compose—a process that is potentially far simpler than re-describing each of the sub-manipulations.

The user could describe each manipulation, one at a time, and after they are convinced CoMo understands them sufficiently, can verbally label them by saying "This is a binary search tree insertion." When the time comes to compose the manipulations into the AVL tree insertion, the user could start the interaction by saying, "Perform a binary search tree insertion, then possibly a left rotation, right rotation . . . " and list the rotations that AVL tree insertions may require. CoMo would ask for the conditions under which to perform one of the manipulations. The user would first go through the process of demonstrating the bookkeeping involved in an AVL tree insertion—drawing the tree heights next to nodes. They would then draw the condition under which the left rotation should happen while saying, "Perform a left rotation if this is true." For example, they may write the line:

```
a.left.key > a.right.key + 1
```

to describe when to perform a right rotation. The user would then do the same for the remaining rotations.

7.1.3 Specifying Algorithms

Another interesting piece of functionality that CoMo should have is the ability to understand explanations about algorithms. Currently, CoMo allows the user to specify only the manipulation, not the method used to achieve it. Take the example of describing list sorting to CoMo. Currently, one would give several examples of unsorted lists, then update each list by clicking and dragging pointers until it is sorted. CoMo would generate input for SPT, which would find *some* code that works. There would be no way to specify whether the quicksort algorithm or the bubble-sort algorithm should be generated; depending on the examples given, both could be equally likely. The user should be able to use temporary pointers to show the process of the manipulation, then communicate that to SPT—possibly based on intermediate state specification or some new feature in SPT's description format. From a user perspective, the algorithm description process will be a pointer-by-pointer run-through of the desired algorithm.

7.1.4 Handling Multiple Large Structures

One of the assumptions CoMo makes about its manipulations is that at most one node can be added, removed, or returned. To lift this assumption, CoMo would need two augmentations. First, CoMo would require an alternate list of questions. In addition to asking about examples where the main structure has four, one, and zero nodes crossed with the auxiliary structure having one or zero nodes, CoMo would also have to ask about three examples were the main structure has four, one, and zero nodes and the auxiliary structure has four nodes. This small change alone would allow CoMo to handle manipulations like the linked list append seen in Fig. 7-1.

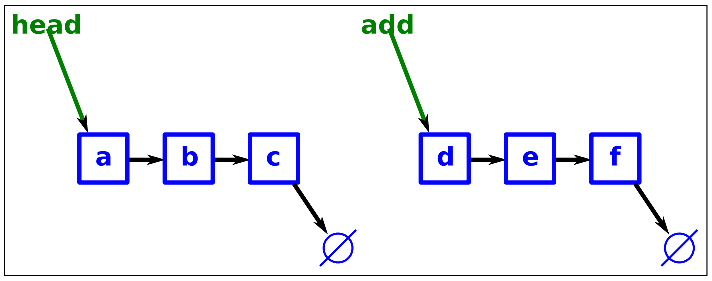
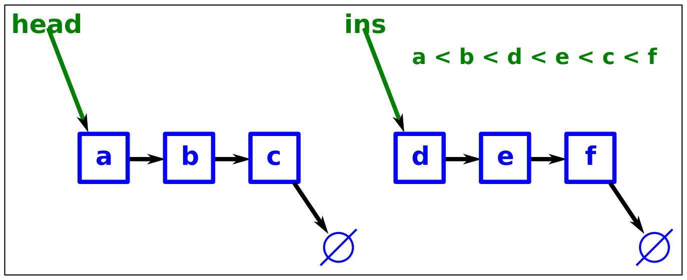
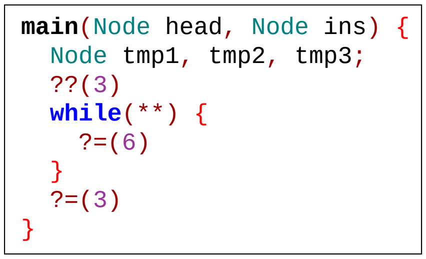


Figure 7-1: A singly linked list append. Notice that the auxiliary structure (beginning with `add`) has more than one node.

The next augmentation involves the CFG rule, and can be demonstrated with Fig. 7-2. In this figure, two ordered, singly linked lists are to be combined—the one starting with `ins` is to be inserted into the one starting with `head`. A node order is specified with the constraint, "`a < b < d < e < c < f`," indicating the final order the nodes should have in the `head` list. Determining where to insert each element requires iterating over the `ins` list and `head` list in tandem to determine where to insert each node. Fig. 7-2b shows a possible CFG for this manipulation. This CFG



(a) An ordered singly linked list append.



```
main(Node head, Node ins) {
  Node tmp1, tmp2, tmp3;
  ??(3)
  while(**) {
    ?=(6)
  }
  ?=(3)
}
```

(b) A possibly adequate CFG.

Figure 7-2: An ordered, singly linked list insertion and an altered CFG that may work for it. Neither of CoMo's CFGs are sufficient to perform this manipulation, however using this CFG may result in SPT synthesizing correct code.

adds an additional temporary pointer and more lines to the center of the while loop, and it allows those lines to use conditionals.

7.1.5 Joint Synthesis Work

A final interesting direction would be integrating more tightly with the code synthesis system. Currently, communication between the code generation portion of CoMo and the user interface portion is limited to the simple serial interaction of state specification. That is, after receiving input from the user, CoMo generates input for SPT and waits for it to finish synthesizing code. How could the internals of SPT be exposed to aid CoMo's understanding of the manipulation? Three possible paths could be explored, namely:

1. Intermediate state specification.

2. Over-constrained specification question generation.

3. Under-constrained specification question generation.

To explain what can be achieved with intermediate state specification, take the example of a linked list reversal. When describing the manipulation, the user clicks and drags pointers to reassign them from the in-order to the reverse-order configuration. As this happens, CoMo keeps track of each intermediate state. These states are not given to SPT, but the information is there. If SPT accepted these intermediate states, including them would give SPT hints about how functioning code might look and may result in quicker synthesis time.

Next, imagine that the input to SPT is under-constrained. This could result in SPT finding two distinct pieces of code that satisfies the input's constraints. CoMo could ask a disambiguating question about an example structure that is affected differently by the different pieces of code. It could then ask, "How do you handle this case" (as with other example questions) and integrate the user's response into an updated input to SPT. On the other hand, if the input has conflicting constraints (i.e., if it is over-constrained), SPT will not be able to synthesize any code that works on the input. CoMo would have to determine which of the scenarios in the description is over-constraining SPT, or perhaps resort to asking the user what is conflicting.

With these features, CoMo has the potential to evolve into a much more capable system. It could evolve into a software engineering tool—a tool able to engage in more complex discussions about data structures and manipulations. Its abilities could expand to discussing manipulations consisting of multiple manipulations, recursive manipulations, and specifying algorithms. It could tie into the inner workings of the code synthesis system to generate questions, and possibly speed up code synthesis. It could be an active software engineering tool, building its ever-growing knowledge base by interacting with users and learning from them, potentially helping to create and visualize new algorithms, data structures, and manipulations.

Bibliography

[1] Aaron Adler. *MIDOS: Multimodal Interactive DialOgue System.* PhD thesis, Massachusetts Institute of Technology, June 2009.

[2] Aaron Adler and Randall Davis. Speech and sketching: An empirical study of multimodal interaction. In *SBIM '07: Proceedings of the 4th Eurographics workshop on Sketch-based interfaces and modeling*, pages 83–90, New York, NY, August 2–3 2007. ACM.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques and Tools.* Addison Wesley Publishing Company, 1986.

[4] Christine Alvarado, Andy Kearney, Alexa Keizur, Calvin Loncaric, Miranda Parker, Jessica Peck, Kiley Sobel, and Fiona Tay. Logisketch: A free sketch digital circuit design and simulationsystem. In *Proceedings of the Workshop on the Impact of Pen and Touch Technologies in Education*, WIPTTE '13, 2013.

[5] Mathias Bauer and Dietmar Dengler. Trias: Trainable information assistants for cooperative problem solving. In *1999 International Conference on Autonomous Agents*, Agents '99, 1999.

[6] David Bischel, Thomas Stahovich, Eric Peterson, Randall Davis, and Aaron Adler. Combining speech and sketch to interpret unconstrained descriptions of mechanical devices. In *Proceedings of the 2009 International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1401–1406, Pasadena, California, July 2009.

[7] Brandon Paulson Brandon and Tracy Hammond. Paleosketch: Accurate primitive sketch recognition and beautification. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, IUI '08, pages 1–10, New York, NY, USA, 2008. ACM.

[8] Sarah Buchanan, Brandon Ochs, and Joseph J. LaViola Jr. Cstutor: a pen-based tutor for data structure visualization. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 565–570, New York, NY, USA, 2012. ACM.

[9] Qi Chen, John Grundy, and John Hosking. An e-whiteboard application to support early design-stage sketching of uml diagrams. In *Proceedings of 2003*

IEEE Human-Centric Computing Conference, HCC '01, Piscataway, NJ, USA, October 2003. IEEE.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[11] Travis J. Cossairt and Joseph J. LaViola Jr. Setpad: A sketch-based tool for exploring discrete math set problems. In *Proceedings of the Ninth Eurographics/ACM Symposium on Sketch-Based Interfaces and Modeling 2012*, pages 47–56, June 2012.

[12] Judith Good, Katherine Howland, and Keiron Nicholson. Young people's descriptions of computational rules in role-playing games: an empirical study. In *IEEE Symposium on Visual Languages and Human-Centric Computing 2010*, pages 67–74, September 2010.

[13] Thomas R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:191–174, 1996.

[14] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, PLDI 2011, 2011.

[15] Isabelle Guyon, Lambert Schomaker, Réjean Plamondon, Mark Liberman, and Stan Janet. UNIPEN project of on-line data exchange and recognizer benchmarks. In *Pattern Recognition, 1994. Vol. 2—Conference B: Computer Vision amp; Image Processing., Proceedings of the 12th IAPR International. Conference on*, volume 2, pages 29–33, 1994.

[16] Ken Hinckley. Input technologies and techniques. In Andrew Sears and Julie A. Jacko, editors, *Handbook of Human-Computer Interaction*. Lawrence Erlbaum and Associates, 1998.

[17] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, OOPSLA, 2010, 2010.

[18] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *Programming Language Design and Implementation*, PLDI, pages 304–314, 2002.

[19] Kenneth F. Kahn. Toontalk—steps toward ideal computer-based learning environments. In Mario Tokoro and Luc Steels, editors, *A Learning Zone of One's Own: Sharing Representations and Flow in Collaborative Learning Environments*. IOS Press, illustrated edition, 2004.

[20] Edward C. Kaiser. Multimodal new vocabulary recognition through speech and handwriting in a whiteboard scheduling application. In *In Proceedings of the International Conference on Intelligent User Interfaces*, pages 51–58. ACM Press, 2005.

[21] Henry Lieberman. *Your Wish is My Command: Giving Users the Power to Instruct their Software.* Interactive Technologies. Morgan Kaufmann Publishers, 1st edition, March 2001.

[22] Henry Lieberman, Elizabeth Rosenzweig, and Christopher Fry. Steptorials: Mixed-initiative learning of high-functionality applications. In *ACM Conference on Intelligent User Interfaces*, number 19 in IUI '14, pages 359–364, February 2014.

[23] Nicolas Mangano and André van der Hoek. The design and evaluation of a tool to support software designers at the whiteboard. *Automated Software Engineering*, 19(4):381–421, December 2012.

[24] F. Modugno, A. T. Corbett, and B. A. Myers. Evaluating program representations in demonstrational visual shell. In *Empirical Studies of Programmers: Sixth Workshop*, pages 131–146, Norwood, NJ, 1996. Ablex Publishing Corporation.

[25] Tom Y. Ouyang and Randall Davis. A visual approach to sketched symbol recognition. In *Proceedings of the 2009 International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1463–1468, 2009.

[26] Beryl Plimmer and Mark Apperley. Interacting with sketched interface designs: an evaluation study. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '04, pages 1337–1340, New York, NY, USA, 2004. ACM.

[27] Subhajit Roy. From concrete examples to heap manipulating programs. In *SAS*, pages 126–149, 2013.

[28] Rishabh Singh and Armando Solar-Lezama. Spt: Storyboard programming tool. In *Proceedings of Computer Aided Verification*, number 24 in CAV '12, pages 738–743, July 2012.

[29] Armando Solar-Lezama. *Program Synthesis By Sketching.* PhD thesis, EECS, UC Berkeley, 2008.

[30] Armando Solar-Lezama. Program sketching. *International Journal on Softgware Tools for Technology Transfer*, 15(5-6):475–495, October 2013.

[31] Armando Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 404–415, 2006.

[32] List of publications for the cmu sphinx speech recognizer. `http://cmusphinx.sourceforge.net/wiki/research`. Accessed: 2013-12-09.

[33] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, PLDI, 2011, 2011.

[34] Carol Traynor and Marian G. Williams. A study of end-user programming for geographic information systems. In *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, ESP '97, pages 140–156, New York, NY, USA, 1997. ACM.

[35] The UNIPEN dataset. `http://www.unipen.org/`. Accessed: 2013-12-09.

[36] Christian von Ehrenfels. über gestaltqualitäten. In *Vierteljaheresschrift für Wissenschaftliche Philosophie*, 1890.

[37] Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, and Joe Woelfel. Sphinx-4: A flexible open source framework for speech recognition. Technical Report SMLI TR2004–0811, Sun Microsystems Inc., 2004.