Agent-based Programming Interfaces for Children

Supporting Blind Children in Creative Computing through Conversation

by

Tina Quach

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

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

© Massachusetts Institute of Technology 2019. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 17, 2019

Certified by . . . .
Andrew Sliwinski
Research Scientist
Thesis Supervisor

Certified by . . . .
Mitch Resnick
LEGO Papert Professor of Learning Research
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chairman, Master of Engineering Thesis Committee

Agent-based Programming Interfaces for Children

Supporting Blind Children in Creative Computing through Conversation

by

Tina Quach

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2019, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

Children that learn to express themselves creatively and think computationally are empowered to create rather than solely consume technology. They practice creativity and learn strategies for solving problems, designing projects, and communicating ideas. However, children with visual impairments cannot access most programming experiences designed for beginners. This deprives these children of the chance to play with expressing themselves through programs that they design and create themselves. In this paper, we introduce Codi, a software agent that children can talk to in order to create, play, modify, and explore programs built using natural language. Through the design and implementation of Codi, I explore how agent-based programming interfaces can make programming accessible to children with visual impairments and multiple disabilities. Interviews, workshops, and test sessions with visually impaired children at Perkins School for the Blind led me to draw two conclusions: (1) Agent-based programming interfaces like Codi can facilitate meaningful creative learning experiences for children who cannot see. (2) In order to support children's engagement with computational thinking concepts and practices, agent-based programming interfaces need onboarding experiences, learning resources, and facilitation that helps learners surface and pursue their own interests and integrate computational thinking concepts.

Thesis Supervisor: Andrew Sliwinski
Title: Research Scientist

Thesis Supervisor: Mitch Resnick
Title: LEGO Papert Professor of Learning Research

Acknowledgments

I have so many people to thank! I find myself almost overwhelmed with gratitude.

Andrew, working with you has helped me grow a lot during this project. Thank you for believing in me and giving me the chance to shape and pivot my project. You've consistently provided guidance along the way to help me imagine what is possible and turn ideas into reality. Of course, I can't tackle all my ideas at once. Thank you for being critical and constructive and balancing my tendency towards breadth-first with advice to narrow, focus, and prioritize. You are a role model for me, and I've been so lucky to have your mentorship. Thanks for sharing yourself—your stories and insights—to help me grow technically, academically, mentally, and emotionally.

I also want to thank all of Lifelong Kindergarten (LLK) and the Scratch Team. You have provided such an amazing home base and community. Our shared mission—engaging people in creative learning experiences—has shaped my own journey working on this project. Every time I've shared my work, you all have been so helpful with constructive feedback. Every question you've asked about it, every expression of care helped me keep going when times got tough. You all inspire me with the care and hard work you put into spreading the powerful ideas behind creative learning.

Almost every week, LLK Grads Dinner was another dip into a creative learning spiral. Thank you, Lily, Sean, Jaleesa, Shruti, Carmelo, Samarth, Lena, Carolina, Marian, Yusuf, Moran, Joy, Andrew, Mitch, and Natalie for the joy and opportunity to grow in that way.

I'd also like to thank students and staff at Perkins School for the Blind for sharing their interests and time with me! It has been so fun. Additional thanks to Vo for hosting my first visits to Perkins and helping me learn about how I can support children with visual impairments. The enthusiasm and experience you brought to design reviews and test sessions strengthened my excitement and deepened my passion for the work. And Miriam—thank you for being my point of contact at Perkins throughout the entire project. Your observations and insights helped make me a more effective facilitator and gave me more context to aid with understanding students as they engage with Codi.

Kara and Willow, thank you working with me to make it possible to create Scratch programs through natural language. It was amazing to collaborate and share ideas with you. We didn't know what we were doing at first, but our explorations helped us find an approach that works and provides children with new ways to engage with creative computing.

Beast of past and present, thank you for being home, keeping me company during late nights, for always cheering me on, and for bringing love and spontaneous fun to my life all the time. You are so special.

To my brother Ken, thanks for play testing my project and inspiring me. To my sister Nina, thanks for being supporting me and giving me more reasons to laugh. And finally, to my parents, Alan and Trang Quach, thank you giving me the opportunity and space to act on my own motivations. I wouldn't have made it here without you.

Contents

1 Introduction **17**

1.1 Scratch as Inspiration 18

1.2 Why a conversational interface? 19

1.3 Codi 19

1.4 Outline 20

2 Foundations of Creative Learning and Computational Thinking **21**

2.1 Papert's Theory of Constructionism 21

2.2 The 4 P's of Creative Learning 22

2.3 The Creative Learning Spiral 23

2.4 Empowering Consumers to Create 24

3 Challenges for Children with Visual Impairments in Creative Computing **27**

3.1 Most programming experiences are not accessible to children who are blind. 28

3.2 Assistive technologies for the blind have steep learning curves and are difficult to use. 29

3.2.1 Screen readers have a steep learning curve, making them time consuming to learn. 30

3.2.2 Screen reader usability is hindered by keyboard input and audio-only output. 32

3.3 Intersectionality in the identities and experiences of children with visual impairments is not usually recognized. 33
3.4 Children without sight face technical, logistical, and personal barriers to creative programming experiences. 34
  3.4.1 Technical Barriers 34
  3.4.2 Logistical Barriers 35
  3.4.3 Personal Barriers 36

4 Related Work **37**
4.1 Creative Programming Experiences for Children with Visual Impairments 37
  4.1.1 Computational Construction Kit for the Blind 37
  4.1.2 Accessible Visual Programming Languages 38
  4.1.3 Accessible Text-based Programming Languages 38
4.2 Conversational Agents 39
  4.2.1 Current User Experience of Voice Assistants 39
  4.2.2 Relationship Between a Child and an Agent 42
  4.2.3 Programming Conversational Agents 43
4.3 Voice and Natural Language as a Medium for Programming 45
  4.3.1 Programming by Voice 45
  4.3.2 Programming through Natural Language 47

5 Research and Design **49**
5.1 Accessibility and Intersectionality 49
5.2 User Experience 51
  5.2.1 First Encounter 51
  5.2.2 Creating a Project 52
  5.2.3 Playing and Exploring a Project 53
  5.2.4 Debugging a Project 54
  5.2.5 Editing a Project 56
5.3 Conversations as Transitions through a State Machine 57
  5.3.1 Reducing Cognitive Load 58

5.3.2 Audio Cues and Background Music 59
5.3.3 Coding Conversationally with Natural Language 60
5.4 The 4 P's of Creative Learning 62
5.4.1 Projects 63
5.4.2 Passion 64
5.4.3 Peers 66
5.4.4 Play 67
5.5 Computational Thinking 67
5.5.1 A Formal System and Tool to Create With 67
5.5.2 Computational Thinking Concepts 70
5.5.3 Computational Thinking Practices 74
5.5.4 Avoiding Confusion in CS Education 77
5.5.5 The Relationship Between the Child and the Agent 79
5.6 Reflections 82

6 Implementation 85

6.1 State Machine 85
6.2 Actions 86
6.3 Arguments 89
6.4 Handling Utterances 90
6.5 Constructing Programs with Natural Language 92
6.5.1 Designing a Grammar 95
6.5.2 Syntactic-Semantic rule pairs 95
6.5.3 Generalizing to Various Inputs 98
6.6 Virtual Machine 102

7 Evaluation 105

7.1 An agent-based interface can facilitate meaningful creative learning experiences for children who cannot see despite usability challenges. . 106
7.1.1 Creative Learning Spiral 106
7.1.2 Building and Applying Mental Models 111

7.1.3 System Intelligence 115
7.1.4 Usability Challenges 117
7.1.5 Boosting Usability 119
7.1.6 Supporting Self-Expression 121
7.2 Onboarding, learning resources, and facilitation are needed to effectively promote engagement with computational thinking concepts and practices. 123
7.2.1 Computational Thinking Concepts 123
7.2.2 Computational Thinking Practices 130
7.3 Feedback 131
7.4 Methodology 132
7.5 Summary 133
7.5.1 Findings 133
7.5.2 Limitations 134

8 Conclusion **135**
8.1 Contributions 135
8.2 Future Work 136
8.2.1 Promoting Community around Codi 136
8.2.2 Agent Customization and Personalization 137
8.2.3 Increasing Tinkerability through Immediate Feedback 138
8.2.4 Extending What You Can Create with Codi 139
8.2.5 Extending Where You Can Create with Codi 139
8.2.6 Abstracting, Modularizing, and Compiling in Codi's Programming Language 140
8.2.7 Making Codi More Conversational 140

A Tables **143**
A.1 Codi Actions 143
A.2 Projects from March Test Sessions 144

B Figures **147**

B.1 Learning Resources 147

B.1.1 Reference Cards from Wizard of Oz testing 147

B.2 Interview Questions 152

List of Figures

2-1 The creative learning spiral is an approach to learning that goes through Imagine, Create, Play, Share, Reflect, and back to Imagine. It's a spiral and not a circle because of the growth and learning that occurs with each iteration. The creative learning spiral helps learners develop creative-thinking skills and practices that promote flourishing. 23

3-1 Screenshot of using VoiceOver on MacOS to navigate scratch.mit.edu. The screen reader's focus is on the "Ideas" link in the toolbar. VoiceOver provides feedback that suggests what keypress combination the user may want to do next, "You are currently on a link. To click this link, press Control-Option-Space." The text spoken by VoiceOver is simultaneously visually represented as text in a box on the screen. 29

3-2 Screenshot of using VoiceOver on MacOS to navigate an online article. The screen reader's focus is on a paragraph in the article, which VoiceOver is reading aloud. The text spoken by VoiceOver is simultaneously visually represented as text in a box on the screen. 30

5-1 A Scratch program that plays drum sounds when the space key is pressed. 68

6-1 Overall System Architecture: User speech gets converted into text which then gets matched to possible actions the system can take. The system attempts to take the first matched action, which often results in a state change and feedback given to the user via audio cues and speech synthesis. 86

6-2 This diagram depicts the transitions between states. When children first interact with Codi, they start off in the Home state. They can record a sound, create a new project, or play an existing project. When they are playing a project, they may either edit the project, go to the home state, or ask to return to whatever state they were previously in. When they are inside a project, they can record a sound, play the project, go home, or go back to the state they were previously in. If they are recording a sound, they may return to their previous state. 87

6-3 This state machine depicts the portion of the system that deals with list navigation. This occurs when the child asks a question that has answers involving multiple items. From any state, a child can ask questions that may put them in the list navigation state in which they can hear the current, previous, or next three items. When they are done navigating, they return to their previous state. 88

6-4 The project is modeled as a simple state machine that maintains whether the project has been named and whether instructions have been added to it. 88

6-5 Textual natural language input is translated into a script through a system built on regular expressions, syntactic, and semantic rules. . . 93

6-6 Parse tree generated for the command "make a variable called x" . . 93

6-7 Parse tree rules: the start symbol maps to an instruction (`S`). The instruction `S` could be expand as an action list (`AL`). An action list can contain a single action phrase (`AP`). Action phrases included commands for creating (`CreateCommand`). The rest of the syntactic rules are designed to allow the grammar to be flexible to different ways to tell Codi to make a variable. 96

7-1 Resource card with instructions for creating a "Give me a compliment" project. 124

7-2 Resource card with instructions for creating a “Tell a knock knock joke” project. 124
7-3 Resource card with instructions for creating a “Make silly sounds” project. 125
7-4 Resource card with instructions for creating a “There’s an anxious cow” project. 125
7-5 Resource card with list of sound commands understood by Codi. . . . 125
7-6 Resource card with instructions for creating a “There’s an anxious cow” project using loops. 126
7-7 The code system, derived from the project goals, consists of six groups of codes. Hierarchy in the codes denote relationships between ideas in the same group. 133

B-1 Learning resource card from Wizard of Oz testing that lists commands you can say anytime. 147
B-2 Learning resource card from Wizard of Oz testing that lists commands for editing projects. 148
B-3 Learning resource card from Wizard of Oz testing that lists commands that work when you’re in the “Home” state. 148
B-4 Learning resource card from Wizard of Oz testing that lists loop commands. 148
B-5 Learning resource card from Wizard of Oz testing with instructions for renaming projects. 149
B-6 Braille learning resource card from Wizard of Oz testing with instructions for making the “Give me a compliment” project. 149
B-7 Steps 1 through 4 of the “There’s an anxious cow” project as braille cards for the Wizard of Oz testing. 150
B-8 Steps 5 through 9 of the “There’s an anxious cow” project as braille cards for the Wizard of Oz testing. 151

Chapter 1

Introduction

How can we empower children with visual impairments to engage with the powerful ideas in computational thinking through creative learning?

Creative learning and computational thinking promote flourishing in today's society, where creativity and the ability to understand, use, and build technology are critical. However, most programming experiences are not accessible to children with visual impairments. Visual programming languages like Scratch [76] are extremely popular among proponents of computer science education. However, visual programming languages pose difficulties for children with visual impairments [49, 77, 50] and need to be made compatible with assistive technologies. Still, assistive technologies for the blind have steep learning curves and are hard to use [47]. To support children with visual impairments in engaging with creative learning and computational thinking, we must recognize intersectionality in the identities and experiences of children with visual impairments. Children may have multiple disabilities including a cognitive disability.

This project has three overlapping goals:

1. To create a platform for creative learning experiences accessible to children with visual impairments

2. To support children in engaging with computational thinking concepts and practices

3. To support children in expressing themselves through projects in ways that build creative confidence

In this thesis, I explore how an agent-based programming interface called Codi can make programming accessible to children with visual impairments and multiple disabilities.

1.1 Scratch as Inspiration

Codi's design is inspired by Scratch. Scratch is both a block-based, visual programming language and an online community through which millions of children create and share interactive stories, games, and animations. Scratch is used in a variety of contexts including schools, after-school programs, and at home. It empowers its users to be active and engaged creators of technology and members of society. With Scratch, children create projects inspired by their passions and share with and learn from their peers. Although Scratch aims to be a platform that empowers children of all backgrounds and abilities to express themselves by creating programs, accessibility issues have been regularly raised by users [49, 77]. For example, it is difficult for color-blind users to identify block categories which are distinguished by color [71]. We hear from educators, parents, and children about the struggles of using Scratch with low vision in both formal and informal learning environments such as schools and after-school programs. In 2015, a parent, advocating for her son, posted in a Scratch discussion forum, "Scratch 2.0 is being used in Math club. The text is so small, and the graphics' detail is relatively subtle for a visually impaired person so he hasn't been able to use it, even when enlarged. Wondering if there is an accessible version out there, or what advice there is to make this more accessible for the visually impaired. He is quickly losing his interest in programming as a result" [72].

1.2 Why a conversational interface?

Exploring accessibility with blindness as a focus, I first explored designing and implementing screen reader compatibility for block-based programming. This involved designing workflows around keyboard controls and audio representation for the blocks. However, most children with visual impairments are not proficient screen reader users. Although designing and implementing screen reader compatibility would make Scratch technically accessible (meeting a standard form of access), many children—and adults—face large learning curves when learning to use a screen reader.

In order to engage with a screen reader accessible block-based programming language, children who are blind would have to first learn how to use a screen reader to access a web browser on a computer. Then, they would learn how to navigate the code editing interface. After, they must build an understanding of the blocks before they can learn programming concepts by building with the blocks. This implies that screen reader compatibility isn't enough for this audience to get started.

Just as Scratch's visual language lowers the barrier to entry for sighted children, a conversational, voice-based interface for programming could lower the barrier for children with visual or motor impairments to engage with computational thinking through creative learning. The conversational interface, designed for a screenless experience, is inspired by voice assistants and demonstrates the potential for programming through conversation. The ubiquity of conversational agents in mobile and home devices presents conversation with technology as a natural way to interact, especially for children who have not yet developed computer skills.

1.3 Codi

Codi is a computer program and agent that children can talk to in order to create, play, modify, and explore computer programs. As an agent, Codi provides a conversational, voice and text-based interface that is accessible to children regardless of visual ability. Designed to support creative learning experiences, Codi facilitates a creative process that minimizes cognitive load, integrates audio cues, and supports coding conversationally with natural language. In order to effectively support computational thinking, Codi must avoid common confusions in computer science education and provide agency in the creative process. As a result, Codi is designed to be a formal system and tool and only provides information and suggestions when the child asks for them. With Codi, children create projects consisting of programs that interact with sounds and audio recordings. A project could facilitate a game, tell a story, communicate a personal message, or be art or music. In this way, Codi supports children's expressing themselves through projects.

1.4 Outline

Chapter 2 describes some of the underlying educational philosophy for the project. **Chapter 3** provides background on the challenges children with visual impairments face when engaging with computing and the limited opportunities they have to engage in creative learning. **Chapter 4** situates my work in relation to related work in the domains of designing creative learning experiences for children with visual impairments, child robot interaction, programming by voice, and programming through natural language. In **Chapter 5**, I present the research and design decisions behind Codi. In **Chapter 6**, I share key ideas and architectures I used to implement an agent-based interface. I follow this in **Chapter 7** with in-depth, qualitative evaluation of Codi based on two blind students' experiences interacting with Codi in March 2019. Finally, in **Chapter 8**, I suggest ways to extend this research through improved natural language processing, further design of the agent's personality, and broadening of the creative possibilities on the platform.

Chapter 2

Foundations of Creative Learning and Computational Thinking

In this section, I share and discuss foundations of creative learning and computational thinking because they drive the design and evaluation of Codi and are core to the project goals.

2.1 Papert's Theory of Constructionism

The Lifelong Kindergarten group at the Media Lab, led by Mitch Resnick, is dedicated to engaging people in creative learning experiences. Lifelong Kindergarten is inspired by Seymour Papert's theory of constructionism [68]. Constructionism is built on Jean Piaget's "constructivist" theories, which assert that knowledge isn't just transmitted from teacher to students, rather students actively build the knowledge in their mind through experience [44]. Papert's theory of constructionism adds to Jean Piaget's theory, claiming that learners make ideas and knowledge when they are actively engaged in making an external artifact (such as a poem, machine, or painting) upon which they can reflect and share with others [44]. In constructionism, learners construct knowledge by building concrete artifacts about their interests and passions [44].

But there's more to constructionism. Rather than describing knowledge acquisition in only cognitive terms, constructionism recognizes the importance of affect, "forming new relationships with knowledge is as important as forming new representations of knowledge" [44]. People form relationships with knowledge when working on personally meaningful projects. Furthermore, "Constructionism also emphasizes diversity: It recognizes that learners can make connections with knowledge in many different ways...encourag[ing] multiple learning styles and multiple representations of knowledge" [44]. Piaget and Papert's Theory of Constructionism and its prioritization of affect and diversity guides the design of Codi as an agent and tool meant to give children the ability to express themselves and engage with their passions creatively and relationally through concrete creations.

2.2 The 4 P's of Creative Learning

Lifelong Kindergarten's approach to engaging people in creative learning is based on four key elements we call the 4 P's of Creative Learning—projects, passion, peers, and play. In "Give P's A Chance", Mitch Resnick elaborates on each of the P's [67]:

- *Projects.* People learn best when they are actively working on meaningful projects—generating new ideas, designing prototypes, refining iteratively.
- *Peers.* Learning flourishes as a social activity, with people sharing ideas, collaborating on projects, and building on one another's work.
- *Passion.* When people work on projects they care about, they work longer and harder, persist in the face of challenges, and learn more in the process.
- *Play.* Learning involves playful experimentation—trying new things, tinkering with materials, testing boundaries, taking risks, iterating again and again.

I see value in the way projects create focused goals and environments for people to apply, think, and learn—not only by doing projects that matter to them, but *also* by collaborating and sharing with peers. Furthermore, creativity is supported by a playful spirit in which children have fun, take risks, and discover for themselves.

2.3 The Creative Learning Spiral

The creative learning spiral is an approach to learning inspired by kindergarten and developed by Mitch Resnick (See Figure 2-1) [65]. The spiraling process is one in which "children imagine what they want to do, create a project based on their ideas, play with their creations, share their ideas and creations with others, reflect on their experiences—all of which leads them to imagine new ideas and new projects."

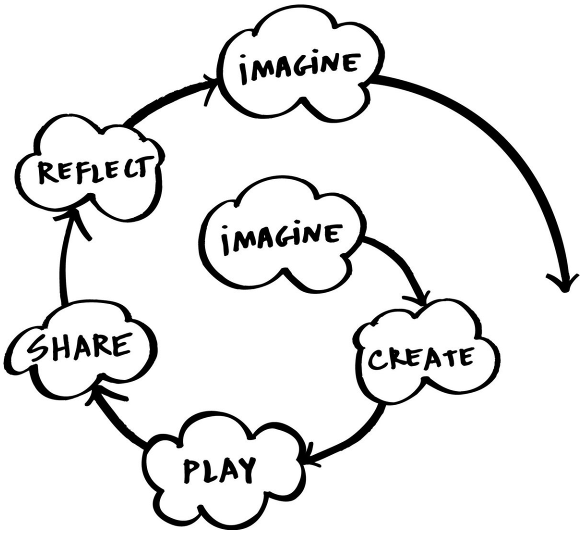


Figure 2-1: The creative learning spiral is an approach to learning that goes through Imagine, Create, Play, Share, Reflect, and back to Imagine. It's a spiral and not a circle because of the growth and learning that occurs with each iteration. The creative learning spiral helps learners develop creative-thinking skills and practices that promote flourishing.

This spiral is immensely powerful in that those deeply engaged in a creative learning cycle could say they are *flourishing.* This is because they are simultaneously engaging with five elements of flourishing: positive emotions, engagement, relationships, meaning, and achievement [78]. Children working on projects about things they care about are likely to be *completely absorbed in the activity* (engagement) and find *meaning* in their work and play. As they make progress on their projects and eventually complete the project, they feel *a sense of accomplishment and success* (achievement). As they create and share projects with peers, they *connect with others* (relationships), and as they play, they have fun and *feel good* (positive emotion).

2.4 Empowering Consumers to Create

As Mitch Resnick writes in "Learn to Code, Code to Learn", "the process of learning to code, people learn many other things... In addition to learning mathematical and computational ideas (such as variables and conditionals), they are also learning strategies for solving problems, designing projects, and communicating ideas. These skills are useful not just for computer scientists but for everyone, regardless of age, background, interests, or occupation" [66]. Computational thinking empowers consumers of digital media to practice creativity through programming in an age where the future of human work is imagination, creativity, and strategy [63, 64]. In his book *Lifelong Kindergarten*, Mitch Resnick makes the case for a transition to a *creative society* [69]:

> As the pace of change in the world continues to accelerate, people must learn how to adapt to constantly changing conditions. Success in the future—for individuals, for companies, for nations as a whole—will be based on the ability to think and act creatively...One of the best ways to help young people prepare for life in a creative society is to make sure that they have the chance to follow their interests, to explore their ideas, to develop their voices.

In 2010, Cuny, Snyder, and Wing defined computational thinking as the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent [31]. This definition ties computational thinking to an expression of ideas and solutions that can leverage computing. Computational thinking is even more powerful because we are surrounded by digital technologies; with computational thinking, we can understand and create digital technologies rather than just use them.

Through computational thinking, we can develop a better understanding of the digital world in which we live. People can use computational thinking concepts to express themselves and their ideas through projects. Through creative learning, people learn to break down complex ideas into simpler ones, to test ideas, assumptions, and prototypes after developing smaller parts, and to debug and fix issues as they arise. People equipped with computational thinking skills and experience creating technology are able to be not just consumers, but creators of technologies.

Chapter 3

Challenges for Children with Visual Impairments in Creative Computing

As technology continues to advance and be integrated into classrooms and schools, its accessibility becomes even more important. Introducing technologies that are not accessible to everyone in the classroom widens and creates gaps in learning and achievement. Even worse, they may lead to feelings of exclusion and decreased self-esteem. We must think about how we can design inclusive technologies and learning experiences. Supporting accessibility for all is challenging. It involves implementing compatibility with existing assistive technologies and designing interfaces that users can customize to their abilities. To reduce the scope of the design problem, I prioritize supporting children with visual impairments.

When I was investigating creative learning experiences for children with visual impairments, I visited The Perkins School for the Blind in Watertown, Massachusetts. Perkins has a Grousbeck Center for Students and Technology (GCST) where mentors and students come together to create and play with podcasts, computers, toys, craft materials. On a weekly basis at GCST, Miriam Zisook leads Maker Club, where students and volunteers collaborate with each other on hands-on projects. These projects utilize inexpensive craft materials and usually last for a single ninety-minute session. Most students with visual impairments do not go to a school like Perkins and likely have fewer opportunities to engage in creative learning. Even at Maker Club, students do not practice computational thinking through programming.

3.1 Most programming experiences are not accessible to children who are blind.

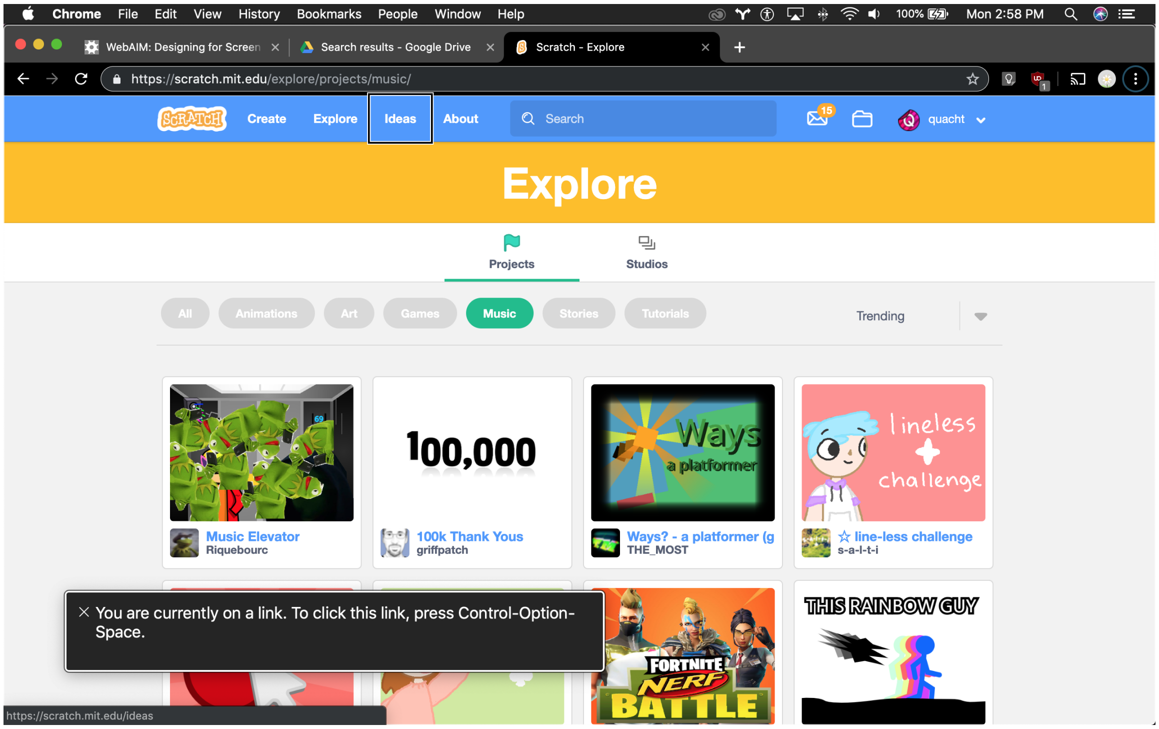
Efforts to lower the barrier to computer science and programming for novices, especially children, have employed strategies such as visual block-based programming interfaces [76, 8], tangible block-based interfaces [10, 16, 22], and new text-based languages that are compatible with assistive technologies [45].

Visual block-based coding has been particularly pervasive as the dominant paradigm because it removes the need to worry about syntax and embeds visual cues in the shape of the blocks to indicate what pieces fit together and thus how someone might approach constructing a program. Rather than type out a particular command with perfect syntax, a child can simply drag the corresponding block from their toolbox and drop it into their workspace. Then, the child can supply arguments if the block allows for it. The blocks provide abstractions, scaffolding, and direction. The success of visual block-based programming interfaces motivates creators of toy robots like, LEGO Mindstorms [40], Anki Cozmo [13], and Sphero [20] to adopt them as the interface children use to program their robots. However, visual block-based programming interfaces are currently not accessible to children with visual impairments.

Tangible block-based programming interfaces have potential because they involve encoding information in the spatial and physical connections between different blocks and do not need to rely on vision. Yet many products using tangible interfaces are not fully accessible because they rely on visual cues. Text-based languages that are compatible with existing assistive technologies work for some, but they are still error-prone and require children to overcome the steep learning curve associated with assistive technologies.

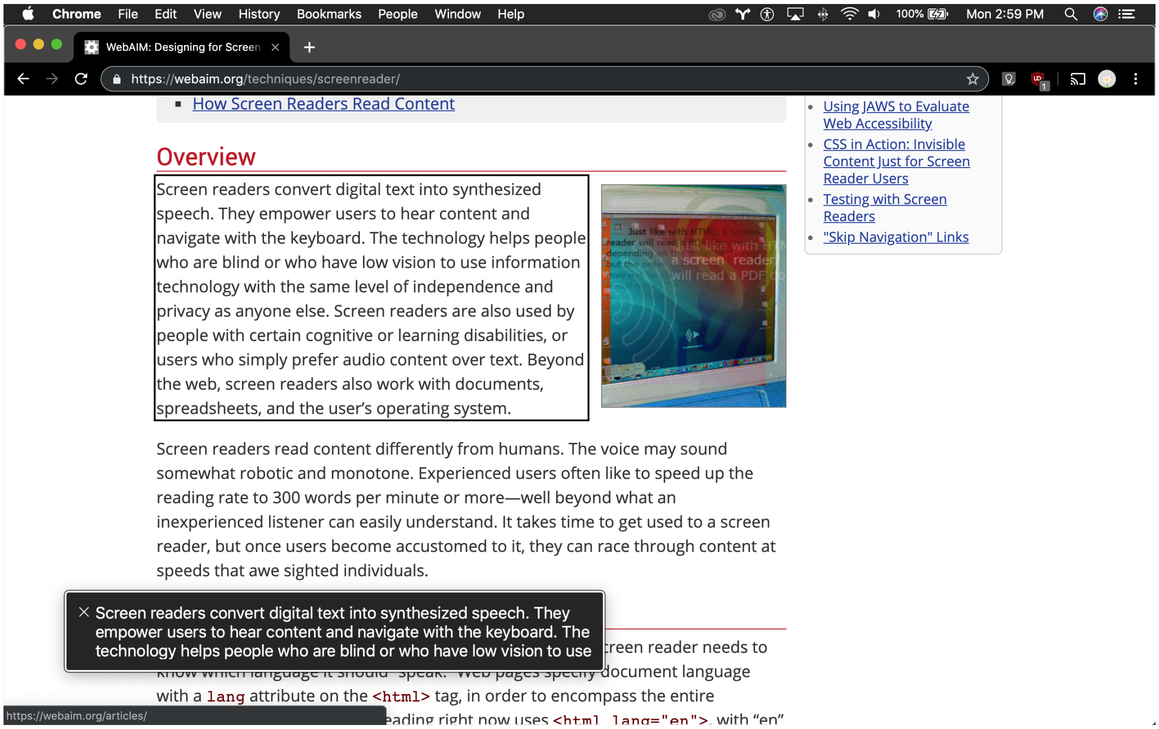
3.2 Assistive technologies for the blind have steep learning curves and are difficult to use.

Assistive technologies for the blind include braille displays, which are electro-mechanical devices that display Braille characters by raising dots through holes in a flat surface, and screen readers, which are software programs that convert visual content on a computer screen into text that can be communicated via speech synthesis or a braille display (See Figures 3-1 and 3-2). The difficulty brought on by using screen readers comes from both its steep learning curve and its limitations.


Chrome File Edit View History Bookmarks People Window Help
100% Mon 2:58 PM
WebAIM: Designing for Screen
Search results - Google Drive
Scratch - Explore
https://scratch.mit.edu/explore/projects/music/
Create Explore Ideas About Search
quacht
Explore
Projects Studios
All Animations Art Games Music Stories Tutorials
Trending
100,000
Music Elevator
Riquebourc
100k Thank Yous
griffpatch
Ways? - a platformer (g
THE_MOST
☆ line-less challenge
s-a-l-t-i
FORTNITE NERF BATTLE
THIS RAINBOW GUY
https://scratch.mit.edu/ideas

You are currently on a link. To click this link, press Control-Option-Space.

Figure 3-1: Screenshot of using VoiceOver on MacOS to navigate scratch.mit.edu. The screen reader's focus is on the "Ideas" link in the toolbar. VoiceOver provides feedback that suggests what keypress combination the user may want to do next, "You are currently on a link. To click this link, press Control-Option-Space." The text spoken by VoiceOver is simultaneously visually represented as text in a box on the screen.


Chrome File Edit View History Bookmarks People Window Help
100% Mon 2:59 PM
WebAIM: Designing for Screen
https://webaim.org/techniques/screenreader/
https://webaim.org/articles/

- How Screen Readers Read Content

- Using JAWS to Evaluate Web Accessibility
- CSS in Action: Invisible Content Just for Screen Reader Users
- Testing with Screen Readers
- "Skip Navigation" Links

Overview

Screen readers convert digital text into synthesized speech. They empower users to hear content and navigate with the keyboard. The technology helps people who are blind or who have low vision to use information technology with the same level of independence and privacy as anyone else. Screen readers are also used by people with certain cognitive or learning disabilities, or users who simply prefer audio content over text. Beyond the web, screen readers also work with documents, spreadsheets, and the user's operating system.

Screen readers read content differently from humans. The voice may sound somewhat robotic and monotone. Experienced users often like to speed up the reading rate to 300 words per minute or more—well beyond what an inexperienced listener can easily understand. It takes time to get used to a screen reader, but once users become accustomed to it, they can race through content at speeds that awe sighted individuals.

Screen readers convert digital text into synthesized speech. They empower users to hear content and navigate with the keyboard. The technology helps people who are blind or who have low vision to use

reen reader needs to document language with a `lang` attribute on the `<html>` tag, in order to encompass the entire eading right now uses `<html lang="en">`, with “en”

Figure 3-2: Screenshot of using VoiceOver on MacOS to navigate an online article. The screen reader's focus is on a paragraph in the article, which VoiceOver is reading aloud. The text spoken by VoiceOver is simultaneously visually represented as text in a box on the screen.

3.2.1 Screen readers have a steep learning curve, making them time consuming to learn.

Let's step into the shoes of Jess, a screen reader user. On her MacBook laptop computer, she uses her keypress combinations to control VoiceOver, the free screen reader that comes with Apple devices. On her smartphone and tablet, she uses swipes and taps. To use a screen reader on a computer with ease and efficiency, she memorizes some basic keypress combinations that allows her to jump to the next or previous element, paragraph, heading, form, link, and button and activate the current element. She learns that there are over a hundred different keyboard shortcuts she could try to memorize, but she's not really sure when she should use a certain shortcut, and it could be several months before she even tries all the different shortcuts. Using the subset of the keyboard controls she knows, she creates and relies on specific workflows to navigate through web pages. Although tutorials help by providing an interactive tour of the basics, she still finds it difficult to determine what keyboard controls apply at every step of using the screen reader.

Jess's struggle exemplifies why there are many screen reader workshops, training sessions, and bootcamps offered to people of all ages [18]. Furthermore, people learning to use screen readers turn towards online tutorials and forums for resources. One Reddit user posted about the learning curve for JAWS (a free screen reader compatible with the Windows operating system) [86]:

> "I have started listening to some of the daisy audio tutorials from the freedom scientific website and I have JAWS 18 installed on a computer. However, I am feeling a little overwhelmed. So here are my questions
>
> -are there resources I should be exploring aside from the freedom scientific website?
>
> -how long did it take before you felt comfortable with JAWS? Not necessarily all features I know there are a lot."

One user responded, "your state's Bureau of Vocational Rehabilitation/state assistance...as well as training in JAWS where someone comes to the house to teach him how to use it extensively" [53]. Another said, "I really think the main thing you need is just a lot of hands-on time and experience. I was a hard core Mac OS user for the first 7 years of my life as a blind person. Despite receiving JAWS training at multiple training centers I never spent much time with it or windows, and I never became comfortable with them" [36]. These responses highlight how the steep learning curve for screen readers necessitates outside support and large amounts of time.

In one computer training school for blind and low-vision adults, the students spent four months progressing from learning how to use a screen reader to web navigation, email, and text editing [22]. Lazar et al.'s study of 100 blind screen reader users revealed several ways using the screen reader itself is frustrating [47]. Learning to use a screen reader is similarly frustrating because it requires hands on time and experience with the screen reader. Each new interface or application takes significant time to explore and learn, and the same difficulties faced by these adults remain for children learning to use screen readers.

Vo, former Coordinator of the Grousbeck Center for Students & Technology (GCST) at Perkins School for the Blind, helped me empathize with the struggle and frustration of students learning how to use a computer. She shared that learning how to use a computer is really difficult for them because they need to learn what a computer is for and what it can do, they need to learn how to touch type, and they need to mentally build and maintain a representation of the computer's state. And once children have made it past the initial learning curve, the actual content they seek or the games they want to play are still inaccessible. In Vo's experience, children get defeated after learning how to use an assistive technology and realizing that they still can't access content.

3.2.2 Screen reader usability is hindered by keyboard input and audio-only output.

Screen reader usability is limited because screen readers do not convey 2D space. Web-based creative tools with complex interfaces pose another barrier because they are difficult to explore using a screen reader. The layering of screen reader on top of the interface of a complex creative tool is likely to create a large cognitive load [1]. Baldwin et al. observed screen reader usage among novices in a computer training school for blind and low-vision adults for four months and identified three problem areas within audio-only interfaces: ephemerality, linear interaction, and unidirectional communication [22]. Ephemerality refers to how the information conveyed by audio doesn't persist; rather, it must be kept in working memory or requested again. The linear interaction refers to the need to provide input before receiving output. This contrasts with the sighted experience where every interaction (such as typing) provides almost immediate visual feedback. The unidirectional communication comes from the screen reader's communicating verbally to the user but the user's not being able to communicate verbally back.

---

[1] In cognitive psychology, "cognitive load" refers to the amount of working memory required to complete a given task [82].

3.3 Intersectionality in the identities and experiences of children with visual impairments is not usually recognized.

> *"It's not just students who are completely blind. A lot of our students are low vision too...low vision as well as blindness as well as additional disabilities is what you're trying to encompass."*
> — Vo, Faculty at Perkins School for the Blind

My inspiration for this project came from investigating how to make Scratch accessible to children with visual impairments. With this goal in mind, I contacted Scratch users with visual impairments, teachers of students with visual impairments, MIT students with visual impairments, the MIT assistive technology information center, and the Perkins School for the Blind in Watertown, MA. Through these conversations, I learned about the large, unmet need for computing experiences for learners who are blind or have other visual impairments.

> *"...not many things are accessible for our students, and that's the problem. They have passions and they want to do these things but they're being told no because it's not accessible."*
> — Vo, Faculty at Perkins School for the Blind

Before we can understand how to improve accessibility, we must recognize that intersectionality exists across demographics. We cannot just design for the ideal blind child—someone who is financially well off and only has a single disability. We must consider a person at the intersection of cultural background and economic status. We must consider that people may have multiple disabilities. Common types of additional disabilities for children with visual impairments include physical, cognitive, and learning disabilities [9].

Furthermore, children with visual impairments in public schools may be just one or two students in a classroom of thirty students. Children in these contexts may not have much access to or support for assistive learning technologies. Accessible products are often powerful resources for them, but access to those may be limited too.

3.4 Children without sight face technical, logistical, and personal barriers to creative programming experiences.

3.4.1 Technical Barriers

Another barrier arises when content is incompatible with the assistive technology. For example, someone might want to play a game with a screen reader, but the game's user interface elements may not have proper labels to that give the user context for understanding the game. If the game provides visual feedback, there would also need to be a way to integrate audio feedback like audio cues or verbal feedback. Websites seeking compatibility with assistive technologies must include code annotated with labels and roles that inform assistive technologies how to navigate the content [2]. Compatibility depends on digital accessibility standards for websites, applications, and assistive technologies. These standards, developed with consideration of human rights and social and economic contexts, prioritize utility and focus on making online information and forms accessible [35]. However, online platforms are implementing more complex interfaces that these standards do not effectively support. Programming is a complex task often done with complex interfaces. Current standards and mechanisms for web accessibility were designed for interfaces like simple forms or information websites, not complex creative tools. Researchers and industry professionals are collaborating on an Accessibility Object Model that may provide tools necessary to implement accessible, complex interfaces on the web [27]. Still, compat-

---

[2] Accessible Rich Internet Applications (ARIA) is a set of attributes that define ways to make web content and web applications (especially those developed with JavaScript) more accessible to people with disabilities. It supplements HTML so that interactions and widgets commonly used in applications can be passed to Assistive Technologies when there is not otherwise a mechanism [6].

ibility with existing assistive technology may not enough be ensure usability because the technology may impose a large cognitive load. Unsighted computer users need to maintain a model of their current state and available next actions. This may be difficult to maintain with audio feedback because it is inherently ephemeral, unlike visual elements. Unsighted computer users either use a lot of working memory or quickly situate, reference, and search for information.

3.4.2 Logistical Barriers

Logistical barriers may also lead programming experiences to be inaccessible. In schools with limited staff support and access to digital and physical resources, students lack access to mentors that provide sufficient scaffolding and learning resources. For example, Miriam, an educator I interviewed, created physical metaphors for programming concepts using a baking sheet, magnetic braille, and puffy paint. Miriam used these physical metaphors to teach a Perkins student the concept of variable scoping during a computer science lesson. Using the physical representation, the student could rearrange code and use his spatial reasoning and touch to explore the concept. Additional financial resources may be needed in order to put children in environments with access to mentors and staff support that can create these types of accessible learning resources.

For those who can afford it, private schools like Perkins School for the Blind [17] provide education and services for students who are blind, have low vision, or are both deaf and blind. These services go beyond traditional academics by including disability-specific skills focused on independence, vocational training, social engagement, and more. Perkins's Expanded Core Curriculum attempts to use everyday tasks and moments as learning experiences [11]. Many students at Perkins that are participating in the expanded core curriculum lead structured lives. This structure may leave little room for creative play, exploration, and ideation.

3.4.3 Personal Barriers

Engagement with a new tool can be very personal. Someone engaging with a new tool needs enough confidence and security to imagine, explore, and take risks—to try something even if they are not sure it will work. Children without sight are often afraid of taking action in an unfamiliar context. In the Perkins context, this is usually because they are afraid they may cause irreparable damage by doing something unexpected or unspecified. Sometimes, they also fear failure. The activities they do in school are often geared towards accomplishing some task with a concrete, specific goal. However, creative activities are more open-ended and often require one to set goals for oneself. When someone learns through the process of creating something inspired by their own passions, they practice and build their creative confidence. The lack of opportunities to be creative means these children have fewer opportunities to build creative confidence that provides the security to imagine, explore, and take risks.

A playful spirit can relieve pressure in being creative, making it feel okay to try something new. When you can bounce ideas off of someone, when you have a certain area of interest that makes you excited—that's when creativity can flourish. In their book *Creative Confidence: Unleashing the Creative Potential Within Us All*, Tom and David Kelley claim that "[a]t its core, creative confidence is about believing in your ability to create change in the world around you. It is the conviction that you can achieve what you set out to do. We think this self-assurance, this belief in your capacity, lies at the heart of innovation" [46]. Creative learning isn't just about ability to come up with a new idea or something never done before. Creative learning is about making your ideas concrete so they can be shared, studied, and iterated upon with others. The iterative nature of creative learning means that those who go through it get a lot of practice and experience facing and overcoming failure. This allows people to build creative confidence.

Chapter 4

Related Work

In this section, I'll share existing creative learning experiences for children with visual impairments, insights from research on how children interact with agents, and past approaches to exploring programming by voice and programming through natural language.

4.1 Creative Programming Experiences for Children with Visual Impairments

4.1.1 Computational Construction Kit for the Blind

In 2002 at the MIT Media Lab, Rahul Bhargava adapted and extended a computational construction kit [1] and studied its use by blind and visually impaired children [24]. Through a series of themed one-on-one sessions with four children, Bhargava aided each child in getting familiar with the mechanics of interacting with the kit and learning about its capabilities. This experience prompted the children to reflect on the role technologies play in life and how technology may be improved or inspire other projects. Finally, the children partnered with Bhargava to create final projects. Conversations with the children revealed that the meaningful, interest-driven projects

---

[1] Bhargava writes, "A computational construction kit can be described as a set of tools or objects that allow one to create a computational artifact" [24].

provided direction and motivation for engaging with computational concepts. His work provides insights about the use of feedback in learning by the blind and visually impaired and how this facilitates or shapes a more ideal creative process for learning. Learning from this research, Codi focuses on project-based learning and makes feedback a critical aspect of interactions between the child and agent.

4.1.2 Accessible Visual Programming Languages

In academia and industry, people have explored how to make visual, block-based programming interfaces accessible to people with visual impairments. At the Rochester Institute of Technology and the University of North Texas, Stephanie Ludi identifies and addresses design issues of visual programming languages [50]. She explores audio cues and workflow design to support access to visual language programming environments for programmers with visual impairments [2]. She also modified and extended Blockly, a library for building visual block-based language, to create a screen reader-accessible, blocks-based language that sighted and unsighted users could use to collaborate together. In November 2017, Lauren R. Milne et al. shared their work building a fully accessible, block-based environment designed for children ages five to seven [54]. The touch-based coding environment uses audio and spatial cues to help blind children learn to program on an iPad. Ludi and Milne's work is valuable in exploring how to make visual interfaces more accessible to children with visual impairments by engaging with audio and touch. Although Codi's interface is primarily based in voice and natural language, I draw from Ludi and Milne's research on audio cues.

4.1.3 Accessible Text-based Programming Languages

Kane et al. created Bonk, a programming toolkit that enables novice programmers to explore computer science by developing and sharing interactive audio games [45]. It provides a basic programming interface for manipulating and playing back audio assets. Bonk also supports assistive technologies and accessible inputs and outputs.

Kane et al. tested Bonk with a group of high school students in a week-long workshop, and the participants successfully collaborated to create audio games—pushing the limits of Bonk in ways that informed and spurred additional feature development. For these children, Bonk's accessible introductory programming activities combat the idea that the field of computer programming is inaccessible. The high school students said they enjoyed learning the basics of a "real" programming language like JavaScript. One student said, "It was our first game in JavaScript and it actually had some substance." Bonk succeeded as a toolkit for programming interactive audio projects. Codi differs from Bonk mainly in its use of natural language and voice to the lower the floor for blind children (who may have multiple disabilities) to engage with creative computing.

4.2 Conversational Agents

Over the last few years, voice assistants have become popular for information access, entertainment, and companionship. Laptop computers, mobile devices, and smart speakers include voice assistants like Amazon Alexa, Google Assistant, Apple's Siri, and Microsoft's Cortana. These voice assistants are conversational agents, software programs that interpret and respond to statements made by users in ordinary natural language. These agents present human-like qualities through their personalities and have the ability to understand and respond to natural language.

4.2.1 Current User Experience of Voice Assistants

Researchers have explored the ways people interact with voice assistants and how people perceive their intelligence. In 2016, a study of 75 regular users of Amazon Alexa highlighted how users must trust the device enough to share information with it before it can actually be a useful tool and accomplish complex tasks [51]. Codi does not ask for personal information because it does not need it. However, Codi still needs the child to trust it if it is to effectively support a creative learning experience in which a child freely plays, tinkers, and shares.

Play is a common and important point of entry for engagement with voice assistants [51]. When people play, they discover features—reasons to keep trying different questions and requests. Furthermore, people are most engaged with the agent when they are playfully interacting or exploring the limits of the conversational agents by experimenting with, teaching, and testing it. In my system, I aim to similarly engage people in playful interactions that facilitate learning through experimentation and tinkering.

A user's technical background will inform the user's expectations of the capabilities and limits of these agents [51]. People with more technical background tend to predict that agents can do less and be more forgiving and resilient when the agents do not meet expectations. Those with less technical background tend to initially think the agent can do more than it actually can and are likely to believe that the agent is static, unable to learn. Although this study was done with adults, these observations may apply to children because these children are likely to have less technical experience. We must ensure that the agent doesn't fail before demonstrating capabilities because children may perceive the system's abilities to be more limited than they actually are. This could close them off to fully exploring and utilizing the capabilities of the agent to express themselves. Luger and Sellen shared that when users failed to get an answer from the agent after successive attempts, they abandoned the task and attempted something they knew that the voice assistant would be able to do. To prevent interactions in which the user gives up on something they want to due to lack of feedback, we must set realistic expectations to scaffold the learning process. This is especially important because creative learning experiences often require grit and iteration. Codi makes giving users an answer a priority so that they will get feedback that ensures that they've been heard and can continue trying to work with Codi.

Human conversations often involve a variety of cues that communicate intelligence; these cues serve to initiate a conversation, appropriately interrupt a conversation, provide feedback, handle misunderstandings or errors, and support turn-taking [29]. Without expected cues, people tend to assign human-like qualities (like the ability to understand outside context) to the agent or avoid complex tasks [51]. Some people limit the type of language they use and gradually abandon attempting to get the conversational agent to do anything out of scope. This is a clear example of a challenge in which designing and building is hindered.

People of all ages use voice assistants, but young children are especially interesting to investigate because their conceptions about technology are rapidly developing and changing. In 2017, Druga and Williams shared initial explorations of child-agent interaction using the Google Home, Amazon Alexa, Anki Cozmo, and the Julia ChatBot with 25 children between the ages of 3 and 10 years old [33]. The children judged an agent's intelligence relative to their own by reflecting on the agent's demonstrated knowledge, the questions the agent understood, and the agent's answers to their questions. Children thought the agent was as smart as they were when the agent could converse and share information about a topic the children already knew about. The children probed the agent with questions to learn more about its personality: what it was, how it worked, and what its limits were—was it able to see and feel? Based on these observations, Druga and Williams's work suggests that designers of agents should consider the agent's voice and prosody to support friendliness, consider how embodiment of the agent enables nonverbal communication of emotion, and facilitate understanding by having the agent provide more meaningful feedback and guidance as to why it cannot understand or respond to the user's input. Codi provides meaningful feedback and guidance when it cannot respond to user input, but it is not embodied, and its design does not control prosody.

Sciuto et al. studied in-home conversational agent usage and learned that children often struggle to use conversational agents because they inaccurately recognize children's speech [75]. This challenge will likely persist until speech recognition technologies are robust to variations in dialect, language, and prosody (which includes intonation, tone, stress, and rhythm). Codi uses existing speech recognition APIs that similarly struggle with children's speech. Sciuto et al.'s focus on the in-home context for conversational agents spurred me to consider how an agent's constant presence allows it to build trust over time. On the other hand, my system is not a general all-purpose assistant; it is a conversational tool for programming that could be integrated into an all-purpose assistant.

4.2.2 Relationship Between a Child and an Agent

Voice Assistants Represent Us

Voice assistant design and interaction studies raise thought-provoking questions surrounding representation, societal stereotypes, and associations. Manufacturers and product designers often make design decisions based on user biases, impressions, and preferences. However this may reinforce stereotypes rather than combat them. For example, most voice assistants have female default voices that tap into the idea that female voices are more friendly and likeable [73]. Commercial voice assistants like Google Assistant, Amazon Alexa, and Siri allow users to change the language and dialect of their speech, but there is still limited representation of styles of talking and of diverse personalities.

In 2018, Hampton et al. tested the role of five different mediators of the association between similarity between individuals and those individuals liking each other [41]. They found that before two individuals interact with each other, certainty of being liked is the strongest mediator. After individuals interact with other, fun and enjoyment and consensual validation are the strongest mediators. Just as a person is likely to better relate to and like someone similar to them, a user is likely to have more fun and better relate to an agent that talks and sounds like them. Similarity between the child and agent (or the lack of it) frames interactions between the two. Technology must acknowledge and represent a diverse set of people rather than some default set or ideal user.

Agent Personality

Children prefer agents with human-like voices and speaking patterns [33]. Furthermore, research suggests that an agent that seems more human is more influential [87]. Children are also more likely to confer human reasoning and emotion to agents with personality. We do not want children to assign human-like reasoning skills to Codi because that may lead to difficulty in building an accurate mental model of Codi and how Codi behaves as a computer program. A conversational agent may tailor its personality to a child. However, the Codi system does not do this. Rather, the child is empowered to shape the behavior of the system by creating programs that extend Codi's behavior.

Conversational Agents Can Shape Children's Perspectives

In Cynthia Breazeal's Personal Robots Group at the MIT Media Lab, researchers study child-robot interaction. These robotic agents are embodied, which allows them to use bodily cues, reflecting human nonverbal communication. Although my system is not embodied, aspects of this research may still apply to non-embodied conversational agents.

Research in Personal Robots Group demonstrates that agents can powerfully influence children's perspectives. In 2017, Park et al. found that interacting with a peer-like social robot with a growth mindset can promote the same mindset in children [60]. Children who played with a growth-mindset robot self-reported traits which seem to suggest a stronger alignment with a growth mindset and tried harder during a challenging task, as compared to children who played with a control. In 2018, Williams et al. built on this work by exploring the ability of a robot to affect children's moral judgements and disobedience. They found that embodied smart toys can influence children to change their judgements about moral transgressions [87]. Although Codi is an agent and can influence a child by modeling particular behaviors, Codi does not do so. Codi's main role is to be a predictable tool for project-based creative computing.

4.2.3 Programming Conversational Agents

The most prominent and popular commercial voice assistants invite consumers to add new skills that extend their functionality. These voice assistants provide programming interfaces that people may use to extend their functionality. However, people looking to create their own Alexa Skills [4] or Actions on Google [3] need to be somewhat comfortable with computer programming. In 2018, Google and Amazon began to provide resources and tools that make it easier for everyday people to control their assistant or smart home device. Both Amazon and Google use mobile applications for specifying a command or trigger phrase and mapping that to a sequence of existing commands. Alexa Skills Blueprints takes this further by creating skills that may be easily remixed through an intuitive graphical interface [5]. Alexa Skills Blueprints provides four easy steps: pick a blueprint, fill in the blanks, use your skill, and share with others. These lay the groundwork for making these devices and their associated agents programmable, but do not yet explore programming the agent through conversation.

To program a system, one must have a mental model of the system and of a program, its components, and its execution. Cho's "Mental Models and Home Virtual Assistants" reveals that people's mental models about home virtual assistants tend to be functional—users only understand the system in terms of how to use the system rather than understanding how the system actually works [30]. Cho observed two kinds of mental models: push models and pull models. Push models are characterized by a perception of the voice assistant as human-like. Users with a push model will provide background information and context before asking a question, believing that the context will help the assistant answer the question. On the other hand, users with a pull model will ask a series of simple questions in order to get to a final answer. From Cho's research, it's clear that the mental model that a child has of a conversational agent will influence what the child deems worth communicating to the agent. Cho's research provides a sense of how children might interact with Codi depending on how human-like Codi seems. This informs the language and actions Codi is designed to support.

Druga, Williams, and Park also highlighted how important programmability is for helping children make sense of how the mind of computational objects might work. Parents and children were confused when asked if the fact that the robot was programmed means it's intelligent [34]. Druga, Williams, and Park hypothesize that if participants could probe and learn the extent of a robot's ability to autonomously learn and react in new scenarios, they might be less confused [34]. This implies that the system should be explicit about not only its programmability, but how it works.

4.3 Voice and Natural Language as a Medium for Programming

4.3.1 Programming by Voice

In the last 5 years, voice programming has become a viable alternative to typing for people with repetitive stress injuries or other mobility challenges [58]. However, voice programming has not been done with natural language. Instead, voice programming tools map short voice commands to operations like moving the cursor to the next open parentheses, accepting an auto-complete suggestion, or inserting a commonly used code template [84]. For example, "exbuff" goes to the end of the file, "slap" adds a new line, "dealt" inserts a template for defining a new function, and "jive" transcribes the next utterances directly without evaluating them [84]. In 2013, Tavis Rudd developed a rule-based, non-conversational, voice system that enabled him to code quickly and efficiently by voice [84]. It was a sort of voice-based smart keyboard. He mapped custom, one-syllable commands like "york", "zork", and "hork" to controls such as highlighting or jumping to certain markers in the code. His controls relied on visual feedback from the screen and were based on existing software development workflows tied to typical programming program syntax. There aren't well publicized attempts to combine auditory feedback with voice programming technologies. Rather the majority of existing voice programming interfaces rely on visual feedback. An agent-based interface for programming requires auditory feedback, and my project explores what kind of feedback to give and when should this feedback be delivered. Many text editors interface with screen readers, but the learning curve associated with screen readers is not friendly to newcomers.

Programming by voice has a long history, dating back to 2005, when Nichols, Wang, and Gupta presented strategies for voice programming interfaces in their discussion of ALVIN, the voice-based scripting language they created to allow users to define navigation strategies on-the-fly while browsing audio pages, much like people visually browse web pages [56].

Nichols, Wang, and Gupta recognized that "the computer can simply ask the user for additional information needed to solve the problem at hand" in order to resolve ambiguity. In their system, users can define entry points into previous code; this supports a nonlinear editing flow. They followed three design principles in their system: be as close to natural language as possible, give users more ways to write correct code than to generate errors, and make all reasonable efforts to prevent the programmer from having to revisit or edit existing code. Nichols, Wang and Gupta hoped to minimize revisits and edits to existing code, but revisiting and editing existing code is a natural part of learning and tinkering. Resnick and Rosenbaum define tinkering as "a playful, experimental, iterative style of engagement, in which makers are continually reassessing their goals, exploring new paths, and imagining new possibilities" [70]. Code is inherently iterative, so designs should make it easy for the user to go back and edit existing code rather than try to minimize the frequency of editing.

In 2006, Begel and Graham shared "An Assessment of a Speech-based Programming Environment" [23]. They recognized that "spoken programs contain lexical, syntactic and semantic ambiguities that do not appear in written programs." Based on these insights, they designed Spoken Java, a syntactically similar, yet semantically identical variant of Java that is easier to speak. They created a Eclipse IDE plugin to support the combination of Spoken Java and an associated command language. Their evaluation with expert Java developers showed that most developers could easily learn what to say but were reluctant to speak literal code out loud and found programming by voice to be slower than typing. Rather than targeting expert programmers and developers who aim to be more efficient, we aim to create an enjoyable, natural interface for children and beginners to create programs just by talking.

4.3.2 Programming through Natural Language

In 2012, Benjamin M. Gordon and George F. Luger from the University of New Mexico shared “Progress in Spoken Programming” in which Gordon writes, “General-purpose speech-based programming languages...suffer” because “existing languages were designed for unambiguous parsing and mathematical precision, not ease of human speech” [39]. Gordon solves this problem with a new spoken programming language that comes with its own editing environment. Gordon later builds on the research of VoiceCode and Spoken Java by sharing the programming language he built based on conversational English [32]. Gordon’s work highlights challenges with programming languages modeled after natural language but does not study them in the context of speaking the text or not being able to see the text. This is where Codi can contribute to programming by voice.

In 2015, Wagner and Gray shared Myna as a hands-free, voice-based programming interface for motorically challenged children [85, 80]. Myna provides a set of voice commands that children may use to control the original block-based programming interface. An example of a command is “drag and drop when clicked”, which utilizes the event block “when clicked”. The interface uses commands that are relevant to the graphical interface and relies on sight to provide visual feedback of the execution of the voice commands. When designing for a screenless experience, the natural language is helpful but references to graphical user interface interactions like “drag and drop” are irrelevant. Unlike Myna, Codi does not and cannot rely on visual feedback and references to a graphical interface.

In early 2018, Stefanidi et al. programmed intelligent environments in natural language using ParlAmI, a multimodal conversational user interface in which users create “if-then” rules that dictate the behavior of an intelligent environment (an environment augmented by Internet of Things (IoT) devices) [80]. Stefanidi et al. recognized the importance of giving the user the ability to easily say “No” and provide negative feedback so that the agent may ask, “Where was my mistake?” This approach is similar in spirit to Desilets, Fox, and Norton’s approach with VoiceCode, where they expect the user to say “not what I said” or “not what I meant” to provide negative feedback. These researchers present great examples of how a conversational interface can effectively ask users clarifying questions to get feedback.

Existing work in voice programming and programming by natural language have only recently begun to explore conversational interfaces in which the system provides visual and auditory feedback.

Chapter 5

Research and Design

My research has 3 intersecting goals:

1. To create a platform for creative learning experiences accessible to children with visual impairments

2. To support children in engaging with computational thinking concepts and practices

3. To support children in expressing themselves through projects in ways that build creative confidence

5.1 Accessibility and Intersectionality

At the Perkins School for the Blind, I conducted interviews that led me to revise my conception of what a blind user is and account for multiple disabilities in my design. The stereotypical blind user is someone that is neurotypical, but at Perkins it is common for children to have multiple disabilities. Perkins does not collect statistics about how many students have multiple disabilities nor keep count of the occurrences of other disabilities. Kate Dietz, Admissions Coordinator at Perkins, mentioned that no one she knew collected data on cognitive and developmental deficits coinciding with visual impairment. However, as Admissions Coordinator, Kate knows that some students at Perkins have autism [1], cerebral palsy [2], and charge syndrome [3]. Kate mentioned that many causes of blindness are brain-based and can affect multiple systems, including cognition [81]. Perkins school is skewed towards kids with some kind of developmental delay because neurotypical children are more likely to be put in a public school. Throughout this project, students from Perkins were my main source of inspiration and feedback.

Designing for multiple disabilities can create a variety of complex challenges when designing accessible systems. For example, some children with cognitive disabilities are not able to type and use screen readers that require them to remember many keyboard commands. Reflecting on the coincidence between blindness and delayed development and cognitive disabilities, it was my assessment that a voice-based, conversational interface based in natural language could facilitate creative learning experiences for this intersectional audience. Natural language can make the expression of computational thinking concepts easier and more intuitive when it allows children to bypass learning unnatural syntax associated with most text-based languages.

Codi is designed to be accessible to children with visual impairments who may have multiple disabilities. Therefore, Codi's interface does not require sight to be used. Codi relies only on a microphone, speaker, and a single keyboard key to facilitate a voice-based experience. To start and stop speech recognition, the child presses the key. If Codi is talking, the child can interrupt Codi by pressing the key to start speech recognition. When a screen is available, the system presents a graphical user interface. Codi also allows users to type what they would normally say out loud. This is useful for clarification because speech recognition systems have trouble distinguishing between homonyms that create multiple valid ways to interpret someone's speech. Typing to Codi can also be useful for people with disabilities who may have sight and can manipulate a keyboard, but cannot speak.

---

[1] Autism, or autism spectrum disorder (ASD), refers to a broad range of conditions characterized by challenges with social skills, repetitive behaviors, speech and nonverbal communication [7].

[2] Cerebral palsy involves difficulties with muscle movement and coordination [21].

[3] Charge syndrome involves hearing loss, vision loss, and balance problems [1]

5.2 User Experience

To get a sense of the system, let's go through some conversations that model how a child might interact with the system to create, play, debug, explore, and edit programs. Dialogue between a person and the system is marked with a `monospaced` font. Speech that is not directed at or coming from the system, is left in the default font. The system speech within a project is denoted by ***`italicized, monospaced`*** font. Most conversations refer to Codi, but when the speaker is "System," the agent is a theoretical agent that reflects behavior **not** implemented by Codi.

5.2.1 First Encounter

When a child first interacts with the system, the agent provides an introduction and a prompt for beginning to engage with the system by playing a project.

> **Codi**: `Hi, I'm Codi! I'm a tool you can use to program and interact with audio projects. Anytime you need help or don't know what to do, you can say "Codi, help" or ask "What can I do?". I will try to answer any questions you have. To start, why don't you say "Give me a compliment" to play the "Give me a compliment" project?`

The introduction equips the child with an important tool: a way to ask the system for guidance or potential next steps by saying "Help!" or "What can I do?" The system prompts the child to play a project as the first action to give an example of what a child might create with the system.

> **Child**: `Give me a compliment`
>
> **Codi**:`*cues project start*`
>
> **Codi**: *`You're amazing`*
>
> **Codi**: `*cues project end*`
>
> **Child**: What else can I do?

Codi: `Here's something you can try. Say "new project" to create a new project.`

5.2.2 Creating a Project

When the child indicates that they want to make a new project, the agent initiates the same flow for creating a project every time. It prompts for a project name, accepts and confirms the child's next utterance as the project name, and then asks for next steps until the child says there are no more steps or that they're done with the project.

Child: `New project.`

Codi: `What do you want to call it?`

Child: `It's a dance party!`

Codi: `OK. When you say, "Codi, it's a dance party!" I'll play the project. What's the first step.`

Facilitator 1: Say *`turn up the music`*.

Codi: `OK. What's the next step?`

Child: `Start the` *`dance celebrate`* `sound.`

Codi: `OK. What's the next step?`

Child: `Say` *`Do the electric slide.`*

Codi: `OK. What's the next step?`

Child: `Say` *`Stomp your foot five times.`*

Codi: `OK. What's the next step?`

Child: `I'm done.`

Codi: `You're in the "Home" state. Now you can say Codi, it's a dance party!" to play the project.`

Child: `It's a dance party!`

Codi: *cue project start* _Turn up the music_" *start the "dance celebrate" sound* _Do the electric slide. Stomp your foot five times._ *cue project end*

This flow is predictable and easy to understand because it repeatedly asks for a next step until the child expresses that they are finished creating the project. However, it privileges a linear creation flow over a nonlinear one. One aspect of the linear flow is that the child needs to name their project before they start making. This is an important design consideration because some children might not know what they want their project to do, but the name of the project pushes the child to provide a direction for the project's design. The linear creation flow expects the child to already know what they want to do. Furthermore, a child in the creation flow can only add steps to a project. It is only after "finishing" a project for the first time that a child can begin modifying a project using edit commands to insert, replace, or delete certain steps. Another challenge is that a child may not know what potential next steps they can include in their projects. To address this, the child can ask, "What commands are there?" to learn about what commands can be used in a project.

5.2.3 Playing and Exploring a Project

A child playing an unfamiliar project is often curious about how the project was made. This ability to "see inside" gives the child a chance to do that. Here's an excerpt from a testing session that demonstrates this.

Facilitator 1: What does a cat do?

Codi: *cue project start* Meow Meow Meow...

Child: *while Codi meows in the background* Oh. Wow! How did you do that? Just...you..told it.

Facilitator 1: *while Codi meows in the background* So. First we'll tell it to stop.

Facilitator 1: Stop.

Codi: `*cues project end*`

Facilitator 1: Okay

Child: Hehehe

Facilitator 1: Okay, so actually I think a good thing we can do is see inside the project and that will tell us what steps it took to make that project happen.

Child: Okay.

Facilitator 1: Okay. So whenever you're ready you can press the button and say "See inside 'What does a cat do'"

Child: `See inside "`*`What does a cat do`*`?".`

Codi: `Opening project "What does a cat do?" for editing. There is one step.`

Facilitator 1: So from here you could ask "What is step one?" or say "Step one" and it'll tell you what step one is.

Child: `Step 1`

Codi: `Step 1, play the` *`meow`* `sound forever.`

5.2.4 Debugging a Project

A child will often debug a project by asking, "What's inside?" and then going through the steps to verify that the instructions making up the program are what they expect. They may also look towards example programs and compare how the project's unexpected behavior compares to their expected behavior. Here's an excerpt from one of the test sessions.

Child: `[System], Rhino Song.`

Codi: `*cues project start* *does nothing* *cues project end*`

Facilitator 1: That seems unexpected...can I can I try testing up one of the projects I made?

Child: Yes.

Facilitator 1: Okay.

Facilitator 1: I think I called it some new basic beats.

Facilitator 1: New basic beats.

Codi:*cues project starting* _*plays beats*_

Child: Hmm.

Facilitator 1: Stop.

Codi: You're in the "Home" state.

...

Child: Yeah, I'm thinking that I would like to see inside "New basic beats".

Facilitator 1: Sure. Yeah. All right.

Child: [System], see inside "*New Basic Beats*".

Codi:Opening project, "New Basic Beats" for editing. There are four steps.

Child: What is step 1?

Codi: Step one, when the project starts, do the following forever set instrument to *piano*, play note *60* for *one* beat, play note *72* for *one* beat, play note *84* for *one* beat *two* times. That's it. That's it.

...

Child: Oh, I just had an idea.

Facilitator 1: Yeah.

Child: Maybe I should have set [the] instrument.

Facilitator 1: Hmm. Okay. Yeah. That could work.

Child: Yeah.

Facilitator 1: Yeah!

Child: Maybe I should have done that first before saying "play note".

In the excerpt, the child investigated the steps implementing an existing project and compared it to the project he aimed to debug in order to find what may have been causing the problem.

5.2.5 Editing a Project

A child can tinker with the project by editing it—inserting, appending, replacing, and deleting steps. This supports an iterative approach to project creation, going back and forth between playing a project to test it out and seeing inside to review and modify the program. Continuing the conversation above from Subsection 5.2.4, the child edits the project.

Child: `Insert` *`set instrument to electric piano`* `before step` *`1`*`.`

Codi:`*cue success*`

Facilitator 1: Yay! Okay. Shall we keep going then? Are you gonna try testing the project?

Facilitator 1: I'm going to hand the computer over to you.

Child: `Go home.`

Codi: `You're in the "Home" state.`

Child: `[System], Rhino Song.`

Codi:`*cues start of project*`

Codi:`*plays note*`

Facilitator 1: Awesome!

Child: And that was the right note! Hahaha!

Here's another example of editing a project that uses the "It's a dance party" project.

Child: `What are all the steps?`

Codi: `Start the` *`dance celebrate`* `sound. Say` *`Do the electric slide.`* `Say` *`Stomp your foot five times.`*

Child: `What is step` *`two`*`?`

Codi:`Say` *`Do the electric slide.`*

Child: `Replace step.`

Codi: `What is the number of the step to replace?`

Child: *`Two`*

Codi:`What is the new instruction?`

Child: *`Say Do the cha cha slide.`*

Codi:`Replaced step two.`

Child: `Add a step.`

Codi: `What is the instruction you want to add to the end of the project?`

Child: *`Say Everybody clap your hands!`*

Codi: `*cues success*`

Child: `Add a step again.`

Codi: `What is the instruction you want to add to the end of the project?`

Child: *`Play the clap sound.`*

5.3 Conversations as Transitions through a State Machine

To navigate the potential complexity of the tool, we create a set of states that the child can transition between (See Figures 6-2 and 6-3).

For example, when a child first begins interaction with the system, they are in the “Home” state, the default state from which a child may play a project, see inside a project, record a sound, or ask questions. Each of these actions correspond to transitions to their own states. When the child is in the “Play Project” state, they can play, pause, and stop the project. If they ask questions about or change how the project is implemented or simply ask “What’s inside?”, they will transition to the “Inside Project” state. A child also transitions to the “Inside Project” state whenever the child creates a new project (by saying “New project”, “Make a new project”, “Create a new project”. Inside a project, a child can learn what instructions make up the project and then modify the project by inserting, deleting, and replacing steps. A child might ask whether Codi has a certain kind of sound, resulting in a transition into the “Navigating a List” state for iterating through the matching sounds. If none of these sounds are satisfactory, the child might want to record a sound, switching to the “Recording a Sound” state.

These states provide a context that help the child and the tool make sense of each other. By making the states explicit in the system’s dialog, we give the child a way to conceptualize and verbally express the system in terms of its parts. As the child converses with the agent, the child navigates through these states and the system responds with verbal feedback and audio cues to inform the child where they are, what they’ve found, whether or not their action was successful, and how their state has changed.

5.3.1 Reducing Cognitive Load

In cognitive psychology, “cognitive load” [4] refers to the amount of working memory required to complete a given task [82]. The design of the system aims to reduce extraneous cognitive load and allow children to more easily engage with computational thinking ideas. Furthermore, sound’s ephemerality may make it hard for a child to

---

[4] Cognitive load theory identifies three kinds of cognitive load: intrinsic, extraneous, and germane. Intrinsic cognitive load is the effort naturally associated with the task itself. Extraneous load is controlled by the educator, facilitator, or tool; it is the way information or tasks are presented to a learner. Germane cognitive load refers to the work put into creating a permanent store of knowledge.

remember everything the system said, so the system supports questions that clarify current state and give a sense of the available choices.

When a child asks, “What is the current state?”, Codi may say something like “You are in the Inside Project state. Your current project is ‘Tell me a joke’.” If you were in the middle of an action, Codi might additionally say “The current action is ‘Replace a Step’.” A child might also ask, “What did you say?”, “What step am I on?”, “What can I do now?”, “How many steps are there?”, and “What sounds are there?” This helps reduce the cognitive load so a child doesn’t need to keep everything going on in the project in his or her head. For some of these questions, it can be tedious to listen to these lists. It can also be difficult to analyze or remember what is in the list. This makes a list navigation paradigm important for usability.

To reduce the cognitive load of formulating commands, Codi will ask you for any missing arguments every time you trigger an action. Take the following conversation for example.

Child: `Replace a step.`

Codi: `What step number?`

Child: *`1`*

Codi: `What is the instruction?`

Child: *`Say "I think you're very beautiful today."`*

Codi: `Replaced Step 1.`

In this example conversation, the child did not have to provide the arguments to the “replace a step” command all at once because the system prompts for missing parts.

5.3.2 Audio Cues and Background Music

The system uses a variety of short audio cues to communicate feedback and state. Codi plays a short “beep” when a project starts and the reverse of that beep when the project stops. Codi also plays a “snap” sound when speech recognition starts or stops. When the system understands an utterance made by the user it plays a short “chime”. When the system rejects an utterance because it can’t understand it, the system plays a “Bonk!” sound. These make the system easier to learn and use by providing quick and immediate feedback to the user. The audio cues are a salient part of the auditory experience, so it’s important to make sure their intent is clear.

Based on feedback from testing sessions at the Perkins School for the Blind, the audio cues follow patterns found in existing voice assistants such as using a sound and its reverse to indicate an event and its opposite. During a test session, the participant said, “I also had a thought that if we were to reverse the sound that [the system] makes when you recognize the project it would almost be the same as when you’re using Google speech recognition and it recognizes what you’re saying ... I feel like that would make I feel like that would make more sense...”

5.3.3 Coding Conversationally with Natural Language

You can talk to Codi like you might talk to a friend. Codi can understand multiple expressions of the same idea or instruction. You don’t have to learn and parse a new syntax for expressing an idea. Natural language doesn’t require extensive training, as it is ubiquitous and familiar. It can handle certain redundancies and variations in user-defined instructions without making the programming more difficult for the child or requiring a more robust programming language. However, there is a trade-off between familiarity and clarity. Natural language itself is complex and often contains ambiguity.

For example, during a workshop, two participants came up with an example conversation that highlights the challenge with ambiguity and proposes a way it gets resolved through feedback from the child during execution. Note that when the speaker is System, the agent is a theoretical agent that reflects behavior not implemented by Codi.

Child: `Describe a character.`

System: `I don’t know how to describe a character.`

Child: `First, ask me for a name.`

System: `Okay, I'll remember it as "name".`

Child: `Then, ask me for a color.`

System: `Okay, I'll remember it as "color".`

Child: `Then, say the color then the name.`

System: `Okay.`

Child: `That's it.`

System: `Okay, I know how to describe a character now.`

Child: `Describe a character.`

System: `Okay, "me for a name"?`

Child: `No, ask "Give me a name"`

System: `Okay, "Give me a name"`

Child: `Andrew`

System: `Okay.  "me for a color?"`

Child: `No, ask "Give me a color"`

System: `Okay.  "Give me a color"`

Child: `Rainbow`

System: `Okay.  "The rainbow, then the Andrew"`

Child: `No, say "Rainbow Andrew"`

System: `Okay, "Rainbow Andrew"`

As we can see, a pure natural language programming system would be difficult to implement and could be confusing and unproductive to users as they struggle to specify the intended behavior of the program. Codi aims to strike a balance between flexibility and clarity by constraining natural language to a simple programming language that uses a formal grammar. This formal grammar is a set of rules that show valid ways to generate grammatical strings by starting with a single start symbol and expanding these symbols until they are part of the base alphabet. These rules can generate an infinite number of possibilities while constraining what is a valid string to be understood by the system. Furthermore, Codi's design includes calling attention to ambiguity and asking for clarification by presenting different interpretations and asking the child to decide between them. This raises the question—how do we disambiguate a user's speech and then present those disambiguations in a meaningful way? To distinguish between multiple interpretations of a command, we can present each unique parse of the same verbal expression by executing it. If multiple interpretations lead to the same effect, then we don't need to ask a child to distinguish between them. A child can compare the behaviors of different parses and choose one. For example:

Child: `Make a variable called` *y*

System: `Okay, what's the next step?`

Child: `Make a variable called` *x*

System: `Okay, what's the next step?`

Child: `Set` *y* `to` *5*

System: `Okay, what's the next step?`

Child: `Set` *x* `to` *y*

System: `Did you mean set` *x* `to the letter y or set x to the variable y which is 5?`

The conversation above exemplifies that a system can ask the child clarifying questions to resolve ambiguities that arise when using natural language to specify program instructions.

5.4 The 4 P's of Creative Learning

At Perkins School for the Blind, an instructor shared that few students have agency over the projects they work on and the things they create. The project ideas usually come from instructors and the projects they do make often don't get shared with and appreciated by their peers. We hope to give these children more agency and opportunity to collaborate and share. The Codi system is designed to support the 4 P's of creative learning: projects, passions, peers, and play.

5.4.1 Projects

> *"Excuse me, [System], could you please fart?" "When I say fart..."*
> — 6th grader at Perkins

While designing the system, I wanted to understand what kinds of projects to support. I considered what children at Perkins would enjoy. During a visit to Perkins in April 2018, I met with Vo, the Coordinator of the Grousbeck Center at the time, to ask her, "What are the sorts of media that children with low-vision and blindness like to interact with and create? What technologies? Games?" She shared that they like music and jokes. They love to engage with the world as if it is a "sound safari", collecting sounds and then playing them back to each other so that they can guess the source of the sound. They also like playing games, like Hangman, Wheel of Fortune, Simons Says, and "choose your own adventure" games. Vo thought the students would like projects that allowed them to bring their personal touch through content such as a project to share and test on trivia about yourself and assets including sounds and recordings. Children at Perkins also like funny soundboards. I saw evidence to support the interest in music and funny sounds during Wizard of Oz testing with children at Perkins.

Child: `Sing me a song.`

System: `I don't understand.`

System: `[System], sing me a song.`

Child: `[System], new project.`

System: `What do you want to call it?`

Child: `Sing me a song.`

System: `What’s the first step?`

Child: `Well, I tell you a song and then you have to sing it.`

System: `I don’t understand.`

Child: `When I say sing a song "No more tears left to cry" by Ariana Grande, you have to sing it!`

Projects in the system are likely to appeal to children’s interests, as long as there are a variety of assets and the ability to create and add custom assets. In this system, a project consists of two parts: sound assets and a computer program that interacts with and controls the playback of these sound assets. The computer program consists of a set of instructions that will get executed in order by the agent. Parallelism and concurrency may be introduced when children specify event-based instructions. With Codi, children create named projects and invoke them by name. For example, the “Tell a joke” project can be invoked by saying “Codi, play the ‘tell a joke’ project” or simply, “Codi, tell a joke.” This adds new ways in which a project can be meaningful by encouraging children to frame projects as extensions of the ability of the agent (“Tell a joke!”) or as new ways to interact with the agent (“How is your day?”).

There are many ways in which a Codi project may be meaningful to the creator or audience of a project. A project may play a specific function, such as to facilitate a game or tell a story. A project’s content might be meaningful, perhaps communicating a personal message or creating or reproducing art or music. The process itself may be meaningful as a learning experience that allows children to discover and explore a specific concept. If the project is created by or for a peer, the social context may make the project meaningful. Meaning may come from the content, playing a project, making the project, and social connection.

5.4.2 Passion

During one of the testing sessions, a fourteen-year-old student at Perkins wanted to make a baby simulation game because she “want[ed] kids and teenagers to have experiences with pretend babies so they can have responsibilities”. She saw the baby simulation game as a way for her peers to practice taking care of a baby as a way to practice and demonstrate responsibility. The project idea clearly came from her passions—her care for her peers as well as desire to gain and demonstrate responsibility. One of the first things she wanted while creating the project was a baby crying sound, but the sound library didn't have one. The sound recording feature was not yet implemented so we couldn't record the exact baby crying sound. We didn't give up! Instead, we ended up improvising by using the speech commands (`Say` *`wah wah wah wah`*). By connecting projects to individual children's passions, we can create learning experiences that are hugely motivating for them. The passion and interest a child has may give them purpose—a reason to explore, a reason to delve deeper, and a reason to keep going when things get tough. A child's project becomes an opportunity and medium for self-expression as well. To open up opportunities for children to create projects based their passions, it helps to ensure that a large space of potential project types and topics that can be explored and expressed by the system.

We aim to support key interest areas in the domains of sound because sound is the prominent expressive medium. When children create with Codi, they have access to a library of expressive sounds and tones ranging from animals sounds, cartoon sound effects, to musical notes. Children can make music, creating beats with various percussion instruments, playing music notes with a variety of instruments, and modifying tempo. They can also tell jokes and stories, make tutorials and reminders, and more by integrating their own recorded sounds and speech synthesis and recognition. Children select their own project goals and topics, and may be inspired by Codi's built-in example projects. The system itself is seeded with some default projects that serve as examples for the range of projects a child might make while also showing some computational thinking concepts.

Sound recording doesn't just fill in gaps in the sound library. Sound recording provides another avenue for expressiveness. Not limited to the existing set of assets, children can make the exact sound they want. Sound recording also gives children the ability to bring their own personal voice and context into their projects. This freedom for expressiveness creates more opportunity for children to explore their passions.

5.4.3 Peers

Advice from staff at the Perkins School for the Blind highlighted the importance of supporting their students' connections to peers through sharing, playing, and creating together. Collaboration and sharing between peers gives children opportunities to learn from and with each other. They made three main requests: support remixing and sharing, facilitate teamwork, and make the product engaging for sighted children as well.

In this context, remixing refers to referencing or using parts of existing creations to create something new. Remixing is great for "Play" because it lowers the barrier to entry for getting started, and it ties to "Peers" because it often involves building off of someone else's creation. Codi is designed for pair programming. Two children working together can feel free to talk to each other and only have Codi listen when they are ready. Here are two examples of pair programming from a testing session. In this first example, Codi doesn't start listening immediately, which allows the child to freely ask the facilitator if she'd like her to try.

> **Facilitator 1**: Mm... *presses button to start speech recognition* `Do you have any cooing sounds?`
>
> **Codi**: `*ends speech recognition* *cues failure to understand* You are navigating a list from the start. Say 'next' or 'previous' to move through the list or say 'exit list' to go back to where you were before.`
>
> **Child**: Do you want me to try?

In the second example, the child and the facilitator press a key to initiate speech recognition and take turns adding steps to the child's baby simulator project.

> **Codi**: `OK. When you say "[System], the baby is crying." I'll play the project. What's the first step?`
>
> **Facilitator 1**: *presses button to start speech recognition* `Say` *`wah wah wah wah.`*

Codi: `*ends speech recognition* OK. What's the next step?`

Child: `*presses button to start speech recognition* Say` *`It's okay, baby.`*

5.4.4 Play

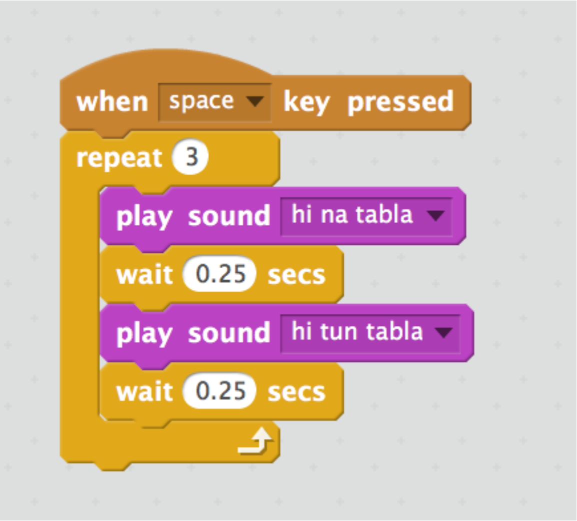
What elements of the system bring delight and surprise? Children playing with the system and trying out commands continually build, apply, and test the model of how the system works. Challenges associated with this process contribute aspects of play. Seymour Papert identifies this as "hard fun", where an activity is fun because it was hard rather than in spite of being hard [59]. Furthermore, to bring delight and surprise to the system, we add elements of randomization. A study evaluating randomness as a resource for design found that randomization in music listening made study participants feel refreshed [48]. The researchers behind the study, Wah Leong et al., also identified that randomness can be a creative tool that generates interest and variety. When a child wants to create a project and asks Codi what sounds there are, Codi returns a list of three random sounds. Any subset of these surprise sounds could spark an idea for a project topic. While the tool and assets associated with the Codi are a set of materials that the child can work with, the randomness places focus on new subsets of the material in ways that playfully prompt discovery, imagination, and creativity.

5.5 Computational Thinking

5.5.1 A Formal System and Tool to Create With

The ubiquity of computing makes the ability to think computationally extremely powerful and valuable. Most children have access to computers and mobile phones on which they play games, connect with others, and more. Computational thinking can empower children to move beyond the role of consumer and into the role of a creator in this digital age. It also gives them a lens for not only understanding how digital technologies work, but also imagining and inventing new ones. Children may employ computational thinking concepts to solve problems and create in new ways. Codi helps children develop critical and creative thinking. A major goal for this project is to support children in building skills and knowledge that will empower them to express themselves both in and outside of the system. As a systematic and predictable (i.e. formal) system, Codi gives children practice building and applying a mental model of how such a formal system works to create something concrete—a computer program.

Codi's design as a formal system relies on a language for expressing computational thinking concepts. Designing agent-based programming languages has trade-offs between "intuitive but limiting" and "less intuitive, but more expressive". For example, let's take a look at a Scratch program that plays drum sounds when the space key is pressed (See Figure 5-1) and then analyze four different ways to describe the program, ranging from more conversational (with more ambiguities) to more explicitly instructional (and fewer ambiguities).



```
when space key pressed
repeat 3
    play sound hi na tabla
    wait 0.25 secs
    play sound hi tun tabla
    wait 0.25 secs
```

Figure 5-1: A Scratch program that plays drum sounds when the space key is pressed.

1. When the space key is pressed, play the *hi na tabla* sound and wait a *quarter of a second*, and then play sound *hi tun tabla* and wait another *quarter of a second*. Do that *three* times.

2. `When the space key is pressed, play the` *`hi na tabla`* `sound, wait a` *`quarter of a second`*`, play sound` *`hi tun tabla`*`, and wait another` *`quarter of a second`*`. Do all instructions` *`three`* `times.`

3. `When the space key is pressed, repeat the following` *`three`* `times. Play the` *`hi na tabla`* `sound. Wait` *`0.25`* `seconds. Play the` *`hi tun tabla`* `sound. Wait` *`0.25`* `seconds. That's it.`

4. `When the space key is pressed, create a repeat` *`3`* `loop. In the loop, play the` *`hi na tabla`* `sound. Wait` *`0.25`* `seconds. Play the` *`hi tun tabla`* `sound. Wait` *`0.25`* `seconds. Outside of the loop...`

In the first statement, "`that`" is used to refer to the 4-block command sitting inside the loop. However, from the sentence alone, it is possible to assume that "`that`" actually refers only to the last command, "`wait` *`0.25`* `seconds`", or any subset of the commands. A computer system that understands a conversational program description must be able to parse English language so that it can infer what "`that`" refers to and what "`another`" is. The second statement clearly states that all instructions are repeated, but it is unclear what subset of commands actually respond to the space key press. The third statement disambiguates the context in which the commands are run (when the space key is pressed). It also uses "`That's it`" to clearly define the scope of the repeat loop. The fourth is the explicit and verbose instruction.

In this system, we chose to design a grammar that supports phrasings like the third example instruction to balance between natural language, clarity, expressiveness, and simplicity. The constrained context of building a program enables us to identify instructions that would be ambiguous in other contexts and use them as concise expressions of lengthier commands. For example, the user could say "`Faster`" to communicate the idea of increasing the tempo of the outputted sound. Using a restrictive grammar limits the ability for the child to communicate naturally and freely, but Codi uses one for understanding user's instructions to support the child's construction of an accurate notional machine.

5.5.2 Computational Thinking Concepts

Contributing to discussions around computational thinking, Karen Brennan and Mitchel Resnick identified seven computational thinking concepts: sequences, loops, parallelism, events, conditionals, operators, and data [28]. For each concept, I define the concept and provide a concrete example from a project in created with Codi.

Sequences In Codi (as well as nearly all other procedural programming languages), an activity or task is expressed as a sequence of individual steps or instructions that can be executed by the computer in order. For example, a project might have the following instructions:

1. `Say` *`it's morning time.`*
2. `Play the` *`bird chirping`* `sound.`
3. `Say` *`I hear birds chirping. It must be a sunny day.`*
4. `Play the` *`tropical birds`* `sound.`

Loops Using the bird project above as an example, what if we wanted multiple chirps so that we could convey that the birds are talking to each other? It would be tedious to repeat the same “`Play the` *`chirp`* `sound`” instruction over and over. Codi allows children to express loops naturally. A child can say “`Play the` *`chirp`* `sound` *`5`* `times`” and get a project that sounds like *chirp!chirp!chirp!chirp!chirp!*

If the child wanted to have some time in between chirps they might instead say “`Do the following` *`5`* `times. Play the` *`chirp`* `sound. Wait` *`1`* `second. That's it.`” The project would instead sound like *Chirp! Chirp! Chirp! Chirp! Chirp!*

It might be more satisfying if not every chirp sounded the same. We can imagine that one bird might chirp a few times at a specific pitch. The child might say to the system, “`Do the following` *`3`* `times. Set the pitch to a random number between` *`-100`* `and` *`100`*`. Play the` *`chirp`* `sound` *`2`* `times. That's it.`” The project would then sound like *chirp!chirp!CHIRP!CHIRP!Chirp!Chirp!*

To add back in the pauses, we need to nest the loops. A child trying to make this happen might say, "Do the following *3* times. Set the pitch to a random number between *-100* and *100*. Do the following *2* times. Play the *chirp* sound. Wait *1* second. That's it. That's it." Here, nesting loops is less natural because of the need to define the scope of looping with "that's it." The project would then sound like *chirp! Chirp! CHIRP! CHIRP! Chirp! Chirp!*

Events Events refer to one thing causing another thing to happen. Event-based programming can bring a projects to life by making them interactive. Rather than have a project that executes a bunch of instructions linearly for the child to passively perceive, the program can respond to someone saying a particular word or pressing a particular button. This concept could be used in a "Tell me a knock knock joke" project:

1. Say *knock knock*.
2. Whenever I say *Who's there?*, say *Woo*.
3. Whenever I say *Woo who?*, say *Don't get so excited, it's just a joke*.
4. Wait *1* second and then say *haha*.

Parallelism Parallelism refers to sequences of instructions happening at the same time. One design decision we explored was whether or not to support parallelism in the program. Should a program only involve the execution of one instruction at a time? Since audio as a medium is inherently linear and time-based, one instruction at a time makes it easier to attribute instructions to their execution when a project is played. Playing with parallelism within a linear medium may be difficult, and supporting parallelism adds complexity to both the system and the user's mental model of the system.

Upon deeper exploration and evaluation of this complexity "cost", we chose to support parallelism and concurrency because it enables the tool to be much more expressive. It also creates an opportunity for more exciting and satisfying projects, especially in the domain of music. Here's a project that plays notes and beats concurrently to create the background music for a freestyle rap battle: `When the project starts, do the following forever. Set instrument to` *`piano`*`, play note` *`60`* `for` *`1`* `beat, play note` *`72`* `for` *`1`* `beat, play note` *`84`* `for` *`1`* `beat` *`2`* `times. That's it. That's it. When the project starts play the` *`closed hi hat drum`* `for` *`1`* `beat forever. That's it.`

Conditionals Conditionals refer to the ability to make decisions or execute particular actions based on certain conditions. Children can use conditionals to create multiple outcomes in their programs. For example, a choose your own adventure game project created with Codi might look like:

1. `Play the` *`rain`* `sound.`
2. `Say` *`your friend is waiting for you at the cafe. Are you going to brave the weather to see her there? Yes or no?`*
3. `Listen and wait.`
4. `If the speech is` *`yes`*`, play the` *`dance celebrate`* `sound and say` *`She's going to be so happy.`* `That's it.`
5. `If the speech is` *`no`*`, play the` *`crying`* `sound and say` *`She's going to be so sad.`* `That's it.`

Operators Operators include mathematical, logical, and string expressions. These include mathematical operations like adding, subtracting, dividing, multiplying, and comparing (checking for relative size or equality). These also include string operations like concatenation and calculating the length of a string. Codi understands multiple ways to express these operations in natural language. For example, Codi understands that "*`1`* `plus` *`1`*", "*`1`* `added to` *`1`*", and "`sum of` *`1`* `and` *`1`*" are all the same. Combining operators with randomness is also very powerful. A child can employ randomness in a project to increase the playability of a game without hard coding every single path. A project that does this with Codi could look like:

1. Make a variable called *path*.
2. Set *path* to a random number between *1* and *3*.
3. If *path* is *1*, play the *meow* sound. That's it.
4. If *path* is *2*, play the *bark* sound. That's it.
5. If *path* is *3*, play the *moo* sound. That's it.

Data Data involves storing, retrieving, and updating values. When programming with Codi, a child can create variables, which can store single numbers or strings, and lists, which can store a sequence of numbers or strings. One classic thing a child might want to do in a game is keep score. To do this, they will rely on variables that may keep count.

Data is also valuable because it can maintain other state in a program and help organize resources and materials. For example, consider this shopping project:

1. Make a list called *basket*
2. Add *strawberry* to list *basket*
3. Add *mango* to list *basket*
4. Add *peach* to list *basket*
5. Say *your shopping basket contains*
6. Log[5] *list*.

When the project played you would hear Codi say, "Your shopping basket contains strawberry, mango, peach."

---

[5] The `log` command results in Codi saying the value of the argument.

5.5.3 Computational Thinking Practices

Karen Brennan and Mitchel Resnick also highlighted four computational thinking practices that focus on the process of thinking and learning: being incremental and iterative, testing and debugging, reusing and remixing, and abstracting and modularizing [28]. With Codi, children can test and debug their projects through iteration. They are also given opportunities to remix and reuse, to reference or use parts of existing creations to create something new.

Testing and Debugging

The system supports debugging by allowing the child to step through the program or select parts of the program to execute. For example, consider the following hypothetical situation.

Child: What state am I in?

Codi: You are in the "Inside Project" state. Your current project is "What animals are in the farm?"

Child: What is the current step?

Codi: Step two, play the *meow* sound and louder and play the *meow* sound and louder and play the *meow* sound.

Child: Play the current step.

Codi: *cues project start*

Codi: *Meow*, *Meow*, *Meow*

Codi: *cues project end*

This conversation demonstrates that a child can easily test portions of their project. This contributes to a child's development of testing and debugging practices like reducing the scope of changes that need to be made.

Being Incremental and Iterative

As a child creates their program, they can test what they have so far and then continue to keep building or make any modifications they see fit. Let's see an example in a continuation of the conversation above.

> **Facilitator 1**: What did you think of Step 2?
>
> **Child**: It's good. Now I can move to the next step. I want to bring a farmer to the scene in the project. `Add a step.`
>
> **Codi**: `What is the instruction?`
>
> **Child**: `Change your voice to` *`alto`* `and say` *`I see you checking out my animals! Do you want one?`*
>
> **Codi**: `Okay, what's the next step?`
>
> **Child**: `No more steps.`
>
> **Child**: `Play the project.`
>
> **Codi**: `*Bark*`,`*Bark*`,`*Bark*`.`*Meow*`,`*Meow*`,`*Meow*`. *`I see you checking out my animals! Do you want one?`*

Reusing and remixing

As a child creates, they might take a break to find some inspiration in other projects, checking out what's inside and then modifying them. A child can ask the system, "What projects are there?" to get a list of existing projects. All users using the same device and browser contribute to the same bank of projects. Then, a child may play the project or explore the instructions that make up the project. This can reveal techniques that inspire their new possibilities in their own projects.

Abstracting and Modularizing

When a child plays a project, they may use the name of the project as an abstraction. For example, "Tell me a joke" is a short phrase that tells Codi to execute a series of steps. This abstraction would be more meaningful if the child tells Codi, “Tell me a joke,” as an instruction inside another project. This framework of running programs inside of programs would allow children to modularize their programs and build complex behaviors based on simple existing projects they’ve already created. This would align with the generative nature of human language in which you create new concepts by combining symbols for concepts that already exist [37].

This is not yet possible with Codi, but here’s how it could work, using one of the students at Perkins who wants to create a baby simulator project. The student described the desired functionality at a high level, “So I want it to be so that you have to take care of your baby, like each 20 minutes it’ll give you like a notification that you need to play with your baby and it’ll grow up. You know what I mean. Like it’ll start off like a newborn and then grow up. I just think that’d be so cool.” Assume the student already made projects that implement specific baby behaviors:

1. Cry
2. Listen for the sound of someone’s voice
3. Giggle
4. Walk (play the sound of footsteps)
5. Ask a question
6. Scream
7. Apologize

The student can then use these projects as abstractions to create higher level abstractions like “Act like a newborn” or “Act like a toddler.”

Child: `New Project.`

System: `What do you want to call it?`

Child: `Act like a newborn.`

System: `Okay, when you say "Act like a newborn," I'll play the project. What's the first step?`

Child: `First, cry.`

System: `Okay, what's the next step?`

Child: `Whenever you hear` *`It's okay, baby,`* `giggle. That's it.`

System: `Okay, what's the next step?`

Child: `Listen and wait.`

System: `Okay, what's the next step?`

Child: `That's it.`

In Codi's current implementation, abstractions only occur semantically in the way projects can be started or invoked. For example, a project invoked by the greeting "Codi, good morning" could be an abstraction for a set of instructions that help facilitate a child's good morning routine. A project invoked by the statement "Codi, it's snowing" could be an abstraction for creating a wintery scene using audio, playing particular sounds and music. A project invoked by a question like "Codi, how are you doing" or a command like "Codi, say something nice" are also abstractions for the instructions that make up the programs they refer to. Codi's design provides minimal support for practicing abstracting and modularizing.

5.5.4 Avoiding Confusion in CS Education

In their 2018 survey of pre-college computer science education, the Stanford Transformative Learning Technologies Lab identified overlapping domains in which students face difficulties in learning computer science [25]. One domain is where students build a general understanding of what programs are and what can be done with them. Another domain is where students develop notional machines, a model of the computer as it relates to executing programs. In an interview in the same survey, Guzdial shares that, "The notional machine is unnatural for us. The inhumanness of computers makes them harder to understand...The computer is a non-human agent that is doing what was specified, and not what was intended." The unnaturalness of a notional machine is what makes developing and applying it so challenging. Bringing in natural language as a way to express computational thinking concepts might help in building a model of how the system works. In order to interact with the system, the child will need to apply the notional machine they've developed. Applying the notional machine is an exercise in thinking like a computer, which may not come so naturally.

In 1983, Bonar and Soloway shared that a common source of problems for novice programmers is the intuition that programming is a conversation with a human-like creature that can infer meaning that is not explicit in the code [26]. This is a danger that the Codi system faces as a conversational interface for creative programming. However, Codi cannot and does not infer meaning that is not explicit in the natural language.

In 1986, Pea found misconceptions around intentionality and the notion of a "'hidden mind' inside the machine" [61]. Specifically, some novice programmers believe that the program has goals and can see what is happening to itself. Codi doesn't communicate a specific goal. Instead, it suggests actions that promote project exploration and creation when a child asks "What can I do?" or asks for help.

To manage expectations about the abilities of the system and help the child build a more accurate mental model, the agent should not come off as much more intelligent than it actually is. We can also support the child by avoiding opaque nondeterminism, which occurs when the agent behaves inconsistently or randomly in unexpected ways that make the agent's behavior unpredictable. See the following scenario for example of opaque nondeterminism where the agent's behavior is inconsistent:

Child: `Play a` *`meow`* `sound.`

System: *`*Meow*`*

Child: `Play the` *`meow`* `sound.`

System: *`*Meow*`*

Child: `Play the` *`cat`* `sound.`

System: `*Meow*`

Child: `Play the` *`bark`* `sound.`

System:`*Bark*`

Child: `Play the` *`dog`* `sound.`

System: `I don't have a sound called` *`dog`*`.`

Children will likely come up with explanations and reasons for the computer behavior—reasons that do not actually exist [25]. Codi's design attempts to mitigate this through predictable behavior.

5.5.5 The Relationship Between the Child and the Agent

Designers of agent-based interfaces for children must recognize that they are shaping a dynamic relationship between the child and the agent. They may shape this relationship through the distribution of agency, the agent's personality, and the agent's intelligence.

Child Agency

In designing the system, I create a close-started and open-ended[6] experience that initially provides guidance and scaffolding and then increases user agency. The system provides a scaffolded experience for those who have may struggle cognitively or those who have not yet built up the confidence to explore new tools and domains without guidance. Specifically, Codi uses prompts to move the conversation forward when the

---

[6] In his PhD dissertation, Jay Silver presents the concept of *open construction kits*. In doing so, he presents questions asked as part of the kit design process. Each of the questions is tied to a part of the construction process [79]. The construction process can be seen as a story with a prologue (How did the people get there and where did they come from?), beginning (What loose materials are they working with?), middle (What tools are they using?), end (What is the final project?), and epilogue (Where is the project placed? What is the afterlife of the project?). He presents open constructions kits as construction kits in which a part of the kit is purposely left unspecified so that someone can bring a part of the world into the construction. Each component of the construction process story could be closed or opened. For example, a painting kit is closed-beginning with a specific set of materials being the paint, canvas, paintbrush, but it is also open-ended in what the painting can be and closed in the substrate, where the painting is placed. Jay's discussion of open construction kit design focuses on whether certain parts of the construction process are open or closed.

child may not know what to try or do next. The system gives the child agency in the majority of interactions so that they have room to engage with multiple pathways to creation and engagement with projects. The child is empowered to ask questions so that they can receive guidance when they deem they need and want it. Agency shapes a child's relationship and communication with the agent.

My design decisions characterize the agent as a tool, rather than a co-creator, student, or teacher. This gives the child ownership of the projects they create. The agent doesn't do more than provide the "materials" or "platform" for the child to explore, experiment, and create. Their projects are recognized explicitly as projects rather than framed only as behaviors taught to the agent [7]. The reason for creating the project is not solely to extend the capabilities of the agent, but to use the agent as a platform for creating new possibilities for play and thinking about computational thinking.

Agent Personality

Although people are working on gender-neutral synthetic voices, most speech synthesis voices are gendered. The gender of the voices assigned to voice assistants impacts the way people perceive these agents. Most voice assistants have female voices. Amazon Alexa's female voice is meant to be "cordial" and "pleasing" [74]. The perception usually reflects existing social dynamics and stereotypes. BMW had to recall its Five Series car in Germany because many male German drivers refused to take directions from a woman [55]! In picking a voice for Codi, we conformed to what was likely to be preferred rather than attempting to shift gender stereotypes. Codi's default voice is that of a female adult whose speech is meant to be encouraging, polite, and transactional. This is consistent with the out-of-the-box abilities of the agent, which are meant to be helpful, not necessarily playful. This contributes to the role of the agent

---

[7] In early designs of Codi, projects were not introduced as projects but as teaching the agent. Take the following conversation for example:
Child: `Tell me a joke!`
System: `I don't know how to do that.  Can you teach me?`
Child: `Sure!`
System: `What's the first step?`

as a tool for creation. However, the agent also inspires projects because projects are the child's medium for shaping the agent's personality and abilities. The following conversation from a test session demonstrates how a child may shape the agent's personality trhough projects.

> **Child**: Can she have conversations with you?
>
> ...
>
> **Facilitator 1**: What do you think a conversation would be like?
>
> **Child**: So, like, how is your day? Or like, how are you doing?
>
> **Facilitator 1**: Yeah. So right now. Codi doesn't really support that.
>
> **Child**: Maybe I can make a project.
>
> ...
>
> **Codi**: `What do you want to call it?`
>
> ...
>
> **Child**: `How are you doing?`
>
> ...
>
> **Codi**: `Ok, when you say Codi, how are you doing? I'll play the project. What's the first step?`
>
> ...
>
> **Child**: `Say` *`I'm good thank you. I hope you're having a great day.`*

Codi's design provides a friendly base personality through voice and friendliness, but is not very playful. However, children have some ability to shape Codi by programming.

Agent's (Lack of) Intelligence

We must consider and shape how intelligent Codi seems because this sets the expectations for what Codi can do and understand. These expectations can constrain or open up possibilities for what a child imagines is possible to do or build with Codi. A child that thinks Codi is more intelligent than Codi actually is may have trouble communicating with Codi in ways that Codi understands. The child may also attempt to explore project paths that are not supported by the system. The negative feedback from these attempts may discourage the child.

We aim to help children develop accurate expectations for Codi's behavior as soon as possible. Specific design decisions include framing Codi's personality and abilities as limited during onboarding, having Codi be systematic and repetitive when responding to commands, and having limited flexibility in Codi's underlying grammar. Codi's personality as conveyed through voice, intonation, and diction influence what level of authority Codi has. Since Codi has an adult voice, it has a level of authority. Yet the restrictive grammar reduces Codi's competency in understanding commands, which portrays Codi as less intelligent.

5.6 Reflections

Throughout the research and design process, there were a few key ideas that guided my decisions in navigating the design tensions

Here, I share quotes from Andrew Sliwinski and Mitch Resnick.

"The concepts should be tools or exist to empower and enable people to be more expressive" Codi supports creative learning spirals and computational thinking concepts because they add to people's ability to express themselves and will be useful in the future.

"When in doubt, bias toward the floor" If anything is going to make understanding harder, don't do it

"The biggest piece of advice is to cut" The goal is to get to a core, base experience from which we can expand upon. It's worth simplifying the interface–even if it's not as human or as "smart"... it makes it easier to test and easier to understand.

"Creative confidence isn't about feeling empowered to create. It's about feeling empowered because of creation." The child feels powerful because they can create and

when they create. Often, the passion must come first to make the act of creating empowering.

"Optimize for optimism" Be more confident in pushing and optimizing for the ideal workflow. You can assume default intentions and provide ways to exit or cancel those.

Chapter 6

Implementation

In this section, I highlight the architectural design and implementation decisions involved in making Codi. The Codi system consists of a front-end interface implemented as a web application that utilizes speech recognition to turn a child's speech into text and conducts Natural Language Processing (NLP) on the text to determine the user's intent. NLP determines the child's intent by searching for matches in the actions that the system can take. If the child's intent is understood (i.e. matched to one of the system's actions) then the system will execute that action (See Figure 6-1).

6.1 State Machine

At a high-level, the Codi system is a state machine that responds to natural language voice or text input. A state machine is a system that has a set of states and defines transitions between those states. In other systems, the state machine transitions may be taken with a specific probability that is coded into the system. In Codi's case, we keep things predictable by making sure that, given a state and an utterance, the resulting transition or response by the system will always be the same. Figures 6-2 and 6-3 condense most of the main state machine of the system into smaller state machines. Missing from Figures 6-2 and 6-3 are the "transitions" in which a child stays in the same state. A child may ask questions in any state, and these usually

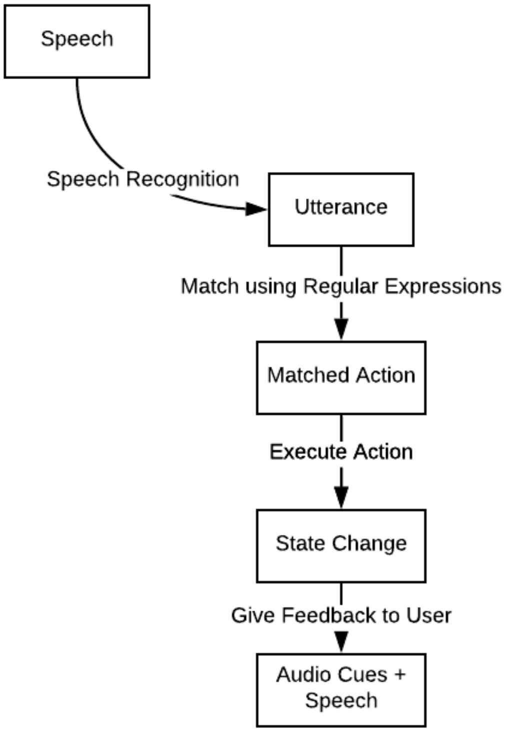


Figure 6-1: Overall System Architecture: User speech gets converted into text which then gets matched to possible actions the system can take. The system attempts to take the first matched action, which often results in a state change and feedback given to the user via audio cues and speech synthesis.

result in the child staying in whatever state they were already in [1].

6.2 Actions

When a child says something to Codi, Codi attempts to match the child's utterance to an action. Actions are categorized based on the action's context and purpose. "General" actions allow the child to navigate the interface and manage their projects. "Edit" actions occur inside projects and involve exploring and modifying projects. "Help" actions involve asking the system questions to learn more about the state and

---

[1] A child can say the following (or supported rephrasings of them) at any time and stay in the same state: "What did you say?", "What did I say?", "What's the current project?", "What's the current state?", "How many projects do I have?", and "What commands are there?". They can also start and stop background music or audio cues, play or rename a recording, rename or delete a project, get a greeting from Codi.

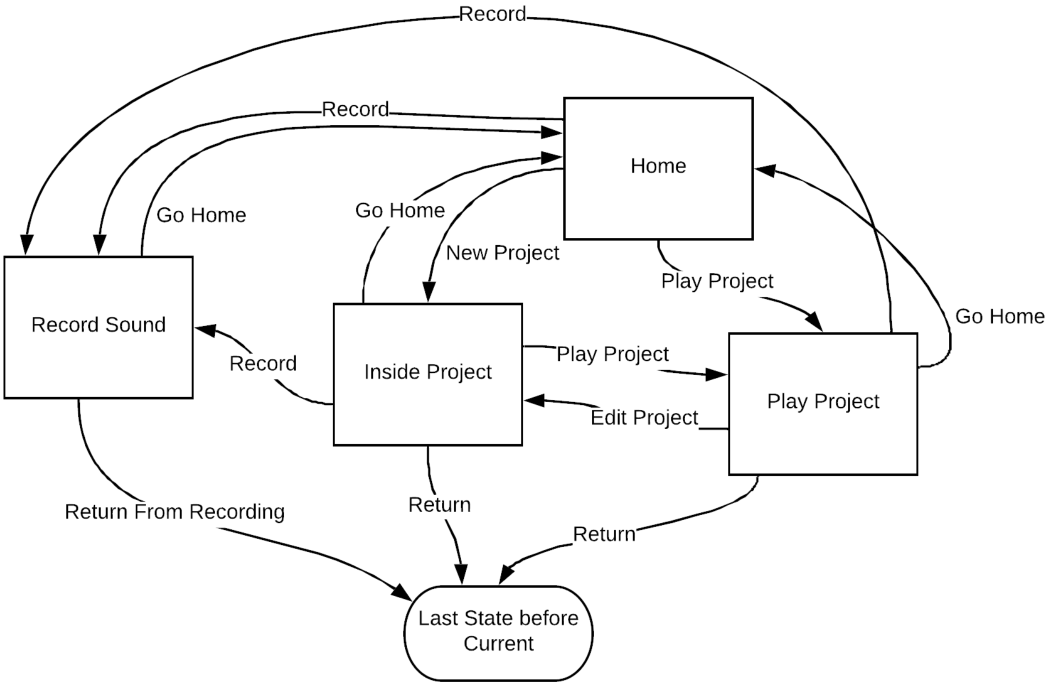


Figure 6-2: This diagram depicts the transitions between states. When children first interact with Codi, they start off in the Home state. They can record a sound, create a new project, or play an existing project. When they are playing a project, they may either edit the project, go to the home state, or ask to return to whatever state they were previously in. When they are inside a project, they can record a sound, play the project, go home, or go back to the state they were previously in. If they are recording a sound, they may return to their previous state.

available assets and commands. The "Interrupt" category has a single action, which is to cancel the current action. Finally, the "List Navigator" actions allow someone to navigate a list and exit it.

Every action is associated with a state transition and a trigger, which is a regular expression (regex) [83] used to match an utterance to an action. For example, the `playCurrentProject` action has the trigger `/play (?:the)? ?(?:current)? ?project|start (?:the)? ?(?:current)? ?project|test (?:the)? ?(?:current)? ?project/`. This trigger allows the child to say the following forms and variations in which "start" or "test" replace "play": `play the current project`, `play the project`, `play current project`, and `play project`.

The action may also have a context validator, which is a function that checks to

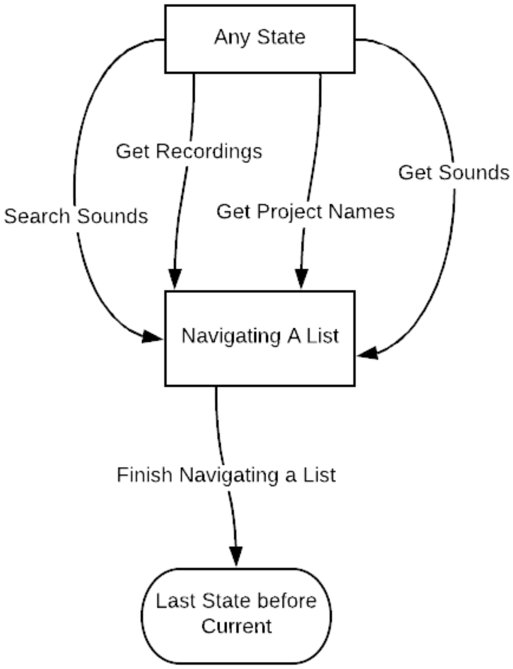


Figure 6-3: This state machine depicts the portion of the system that deals with list navigation. This occurs when the child asks a question that has answers involving multiple items. From any state, a child can ask questions that may put them in the list navigation state in which they can hear the current, previous, or next three items. When they are done navigating, they return to their previous state.

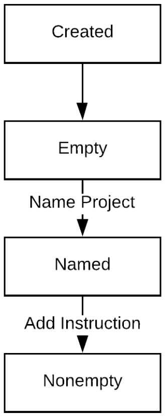


Figure 6-4: The project is modeled as a simple state machine that maintains whether the project has been named and whether instructions have been added to it.

see whether the action is valid. For example, in order to play the current project, there must be a current project defined. The regex enables the system to detect the action in the child’s utterance, while the context validator checks to see if the action can actually be triggered. If the context is invalid, Codi tells the child what exactly about the context is invalid in the form of Codi’s speech feedback. For example, Codi might say, “`You don’t have a current project. Create, open, or play a project to set a current project.`”

Every action also has an ideal trigger. If the child asks “`What can I do now?`” or “`What commands are there?`”, Codi picks a possible action and presents the action by sharing its name, description, and the action’s ideal trigger, an example phrase that demonstrates the action. If the action takes an argument, the ideal trigger will include dummy arguments. For example, the “insert after step” action would have the ideal trigger “`After step number` *`1,`* `insert` *`play the meow sound.`*” Actions provide a standard interface for Codi to communicate about and execute actions. Arguments are a part of the standard interface provided by actions. See Appendix A.1 for a list of Codi Actions.

6.3 Arguments

When an action has arguments, its regular expression trigger extracts them. For example, the `checkSound` action has the trigger `/do you (?:have|know) (?:a|an|the|this|any) (.*) sound/`, in which the `(.*)` extracts the name of the sound from questions like “`do you have a bouncy sound?`” and “`do you know any dolphin sounds?`” The action’s specification lists the arguments so that they can be matched to the arguments extracted by the regular expression. Each argument has its own name and description and may have a validator that will provide feedback to the child when the argument is invalid. The `play` action provides an example of a situation where a validator comes in handy: a child can only play a project that exists. A validator for the `play` action takes the project name value extracted by the regular expression and checks that the system has stored a project by the given name. If the child specified a name that doesn’t match any existing project, Codi replies, “I don’t have a project called [project name].”

If the child triggers an action without providing the required arguments, Codi asks the child for the arguments one-by-one. This is a common approach used in conversational interfaces called slot-filling [19]. It allows the system to obtain additional arguments necessary to carry out an action associated with a given intent. If a child says "insert a step" while they are editing a project, Codi will ask the child for the what step to insert, at what step number to insert the step, and whether to insert before or after the given step number. Each argument's name and description is used to request arguments during slot-filling. Arguments are critical for computation and enable slot-filling.

6.4 Handling Utterances

When the system receives an utterance from the child, it must pass the utterance to the appropriate module for processing the utterance. There are five candidate modules: *project manager*, *project*, *list navigator*, *action*, and *argument*. A module's appropriateness is based on the current state and a priority order. The priority order is needed because an utterance could match multiple triggers, but the system will always take the first match.

1. In Codi, *project manager* handles interrupts such as canceling.
2. If the child is currently navigating a list, the *list navigator* will handle the utterance.
3. Next, *project manager* handles questions about assets, states, and commands.
4. After handing interrupts and questions, we start from the narrowest context (i.e. providing an argument) and broaden from there. If the current action is missing an argument, the *argument* module will handle the utterance by attempting to set the value of the argument to the child's utterance.
5. After step 3 forces arguments to be satisfied, the current *action* module handles the utterance by executing the action with the given arguments.

6. If there is no current argument or action, the system checks for a current project. If there is one, the *project* module tries to handle the utterance, looking for intents related to building, exploring, or modifying the project.

7. Finally, if none of these smaller scopes apply, the general state machine will match the utterance to higher level system navigation commands implemented by the *project manager* module. These commands include "Play the project" and "What's inside [the current project]?"

Let's walk through an example. Assume the child is just starting to interact with Codi and says, "`Create a new project`". When this happens, the system goes to step 7, and the *project manager* module handles the utterance by asking the child for the project name and transitioning into the "Inside Project" state. Then, when the child replies with the project name "`Make silly sounds`", we jump to step 6, where the current project (i.e. *project* module) handles the utterance. Codi will then prompt for the first step of the project. When the child replies, "`Play a random sound`", the *project* module handles the utterance again.

To demonstrate handling utterances in narrower contexts (steps 4 and 5), let's continue with our current scenario and say the child finishes the project, plays the project, and wants to edit the project. The child asks,"`What's inside?`" to enter the project, and start editing, asking Codi to "`replace a step`." First, Codi goes to step 6, where the *project* module creates an action to execute and sets it as the current action. If the current action (replace a step) has missing arguments, Codi prompts the child to slot-fill the argument "`What is the number of the step to replace?`" and sets the current action. Setting the current action results in the next utterance from the child being processed in step 4. When all arguments are satisfied [2], the action gets executed and the current argument and current action become unset.

When Codi's main state machine is handling the utterance, it employs a few techniques to bolster speech recognition accuracy and compensate for speech recognition inaccuracies. The system employs fuzzy matching to compare a child's utterance to

---

[2] The number of the step to replace and the new instruction are both specified.

known formulations of commands based on the regular expressions associated with each action. Specifically, we use the JaroWinkler [88] distance measure to score the match. When the Jaro-Winkler distance score is less than .25 (a threshold determined by trial and error), Codi asks, "Did you want to [action name]?" If yes, Codi will execute the action, and if no, Codi says, "Okay. I will not [action name]."

6.5 Constructing Programs with Natural Language

In the system, every project is a program represented by a state machine (See Figure 6-4). This state machine stores the project's instructions, maintains a pointer to the current instruction in the project, and has a reference to a project editor that implements all of the "Edit" actions referenced in section 6.2. As a child adds new instructions to the project, Codi verifies that each new instruction is valid. The instruction is valid when it is parseable using the grammar [3] we've designed to represent natural language commands in the form of an abstract syntax tree that can be executed by an interpreter. In collaboration with Kara Luo and Willow Jarvis [4], I created a program that accepts text and builds up a program that can be exported as JSON (JavaScript Object Notation) that gets loaded into the virtual machine for execution. In this section, we delve into the implementation of the system used to translate natural language instructions into programs.

As depicted in Figure 6-5, the program first uses regular expressions to extract unrecognized words that must be added to the grammar before the system can parse the input. The unrecognized words either represent a number or is the name of a variable, list, message, sound, voice, or language. Then, the grammar is used to generate a parse tree for the instruction (See Figure 6-6). The semantic rules are evaluated on each parse tree to generate a script (i.e. program instructions). This occurs for each input. A series of inputs is used to construct and modify a JSON specification of a project.

---

[3] See Subsection 6.5.1 for more informtion about the grammar.

[4] Kara and Willow were my project partners for the MIT course 6.863 NLP taught by Bob Berwick.

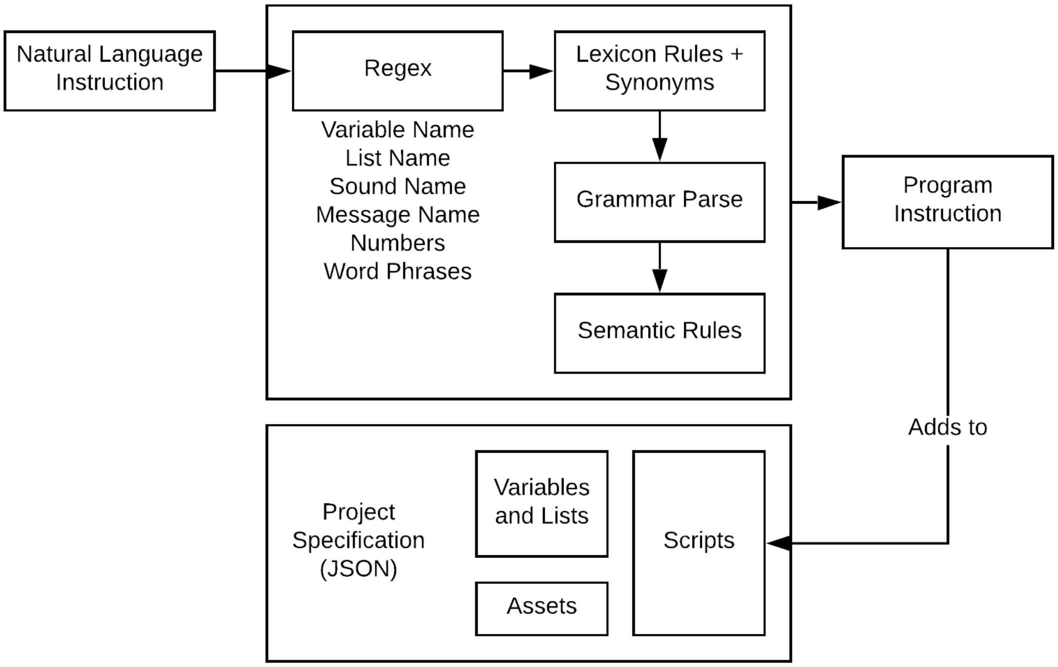


Figure 6-5: Textual natural language input is translated into a script through a system built on regular expressions, syntactic, and semantic rules.

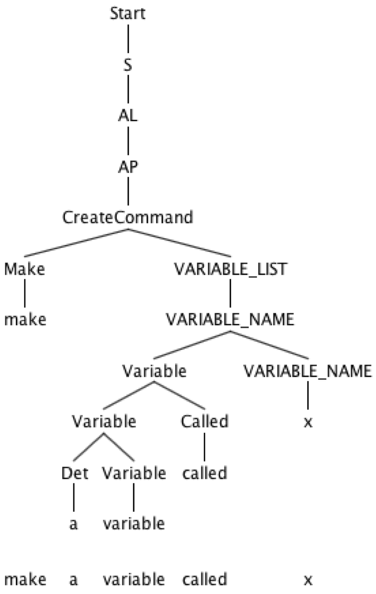


Figure 6-6: Parse tree generated for the command “make a variable called x”

While the aim of the system is to provide a flexible programming interface, there are several restrictions on how a child may interact with the system.

1. **Data structures are expected to be declared and created before they are used in other commands.** Like most conventional programming languages, the system expects a statement creating a variable or a list with an assigned name before this name could be used to reference this variable or list in the context of other instructions.

2. **The instructions are expected to be *complete*.** A *complete instruction* is one that defines the context in which an action should be completed (if any) as well as the complete set of actions that are attached to this context. If there are multiple actions to be taken, all of them should be provided in this complete instruction.

   Three examples of *complete instructions* are provided below.

   ```
   "make a variable called x"
   "play the meow sound"
   "if the timer is greater than 2 then reset the timer
      thats it"
   ```

   Each of the sample instructions provided above can behave as a standalone command. In the case of the third statement, both the condition to test on (timer is greater than 2) and the corresponding action to be taken (reset the timer) are clearly defined. The use of the trigger word `"thats it"` is discussed in Section 6.5.3 in further detail.

   In contrast, three *incomplete instructions* are listed below.

   ```
   "if x is greater than 3"
   "x is not equal to y"
   "repeat add 1 to x"
   ```

   In the first incomplete instruction, the condition is declared for a conditional command, but no action is attached to it. While the second failed statement may be considered a completely valid boolean phrase, there is no action attached to the statement. To make this boolean phrase a *complete instruction*, the child could use it as a condition in a conditional command and attach an action as the response for when the statement evaluates to true. The third failed case is a fragment of a loop command where the duration of the loop is not clearly defined. The child may replace this command with *"`repeat add 1 to x 5 times`"* or *"`repeat add 1 to x forever`"*, either of which would be considered a *complete instruction*.

6.5.1 Designing a Grammar

In this context, a grammar is a set of syntactic rules that show valid ways to generate grammatical strings by starting with a single start symbol and expanding these symbols until they are part of the base alphabet. For example let's consider the grammar that would be able to create the parse tree depicted in Figure 6-6. The grammar could look like the set of lexicon and syntactic rules in Figure 6-7.

To design the grammar, we considered how children express and describe programs and commands in natural English. We manually generated example sentences that correspond to the commands we aimed to support. Beyond natural language commands, the way humans describe the interactions and relationships between these commands is also flexible. As a result, we considered how entire programs are described in natural language which I discussed in Subsection 5.5.1. Our grammar is designed to be able to parse the example sentences we generated.

6.5.2 Syntactic-Semantic rule pairs

Every syntactic rule has a semantic rule. While the syntactic rule presents a way to break the text into meaningful parts (word phrases), the semantic rule defines a function that takes these parts as arguments and makes meaning of them. These semantic rules filter and act on only the words that correspond to arguments. Once a parse tree is fully generated using the syntactic rules, the semantic rules are applied

Syntactic Rules:

```
Start -> S
S -> AL
AL -> AP
AP -> CreateCommand
CreateCommand -> Make VARIABLE_LIST
VARIABLE_LIST -> VARIABLE_NAME
VARIABLE_NAME -> Variable VARIABLE_NAME
Variable -> Variable Called
Variable -> Det Variable
```

Lexicon Rules:

```
Make -> make
Det -> a
Variable -> variable
Called -> called
```

Figure 6-7: Parse tree rules: the start symbol maps to an instruction (`S`). The instruction `S` could be expand as an action list (`AL`). An action list can contain a single action phrase (`AP`). Action phrases included commands for creating (`CreateCommand`). The rest of the syntactic rules are designed to allow the grammar to be flexible to different ways to tell Codi to make a variable.

to each parse tree constructed by the grammar.

Let's go through an example that will clarify how the syntactic and semantic rules work together to process a child's instruction. The system generates a parse tree for the instruction "`make a variable called x`" (See Figure 6-6). We apply the semantic rules starting from the leaves of the tree in Table 6.1. Each node's children are arguments for the semantic rule function corresponding to the parse rule that relates the node to its children.

To support multiple variations of the same command, we define multiple syntactic rules to correspond to the same function via nearly identical semantic rules. In this way, our system can compute on only the meaningful arguments in the input and drop filler words. For example, in the instruction "`set element` ***`1`*** `of the list called` ***`numbers`*** `to` ***`1`***", the auxiliary phrase "`the list called`" does not provide any additional information, and the child may naturally choose to exclude it from the

#	Syntactic Rule	Semantic Rule	Explanation of Semantic Rule
1	`Start ->S`	`return processSentence(S)`	Returns the resulting instruction along with other important project information such as variables, lists, and sounds
2	`S ->AL`	`return AL`	Returns the action list
3	`AL ->AP`	`return getAP(AP)`	Returns action phrase (AP) wrapped in brackets if the command requires it
4	`AP ->CreateCommand`	`return CreateCommand`	
5	`CreateCommand ->Make VAR_LIST`	`return createVar(VAR_LIST)`	Creates and stores variable and returns a [”doWaitUntil”, .1]
6	`VAR_LIST ->VAR_NAME`	`return wrapInList(VAR_NAME)`	Returns the variable name inside a list
7	`VAR_NAME ->Variable VAR_NAME`	`return VAR_NAME`	Returns only the name of the variable instead of including words that may precede the variable name such as ”variable” or ”the variable”
8	`Variable ->Variable Called`	`return None`	Returns nothing because these words are not represented in the final program instruction.
9	`Variable ->Det Variable`	`return None`	Returns nothing because these words are not represented in the final program instruction.

Table 6.1: Every syntactic rule has a corresponding semantic rule that takes as arguments each fragment in the right side the syntactic rule. Each semantic rule doesn't have to use all of its arguments. For example, “the variable x” can be parsed as a VAR_NAME with syntactic rule 7 (VAR_NAME ->“the variable” “x”). The corresponding semantic rule drops the Variable fragment and returns only the VAR_NAME on the right hand side because but only “x” is meaningful.

instruction.

6.5.3 Generalizing to Various Inputs

Given a set of syntactic and lexical rules, the parser can only parse a sentence if every token is accounted for in the rules. However, a programming system needs to support numbers and variable names, which cannot be specified in the grammar ahead of time as there is an intractable number of possible variable names and numbers that may be specified when building the program. Furthermore, children may unintentionally use synonyms in communicating instructions. For example, a child looking to "`play the` *`bouncy`* `sound`" might say "`start the` *`bouncy`* `sound`" or "`begin the` *`bouncy`* `sound`" instead of play. To be more flexible, the system handles synonymous phrases.

Handle names, number phrases, and word phrases Children may give names to variables, messages, or lists so they can refer back to and appropriately modify them. Children may also choose to include numbers in their instructions to set a variable to a particular value, index into a list, or perform arithmetic operations, comparisons, or define bounds for random number generation. A child using speech synthesis or recognition in their projects must provide word phrases as arguments. For example, consider the following natural language instructions where the arguments are highlighted:

> `Whenever you hear` *`Hi, Codi`*`, say` *`Hi! What's your name?`* `and listen and wait.`
>
> `Make a variable called` *`name`* `and set it to the speech.`
>
> `Say` *`Nice to meet you`*
>
> `Log` *`name`*

All together, this presents a challenge: the grammar needs to be able to handle and parse these numbers, constant names, and word phrases correctly, and it is impossible to predict which numerical values or constant names will appear in the instructions before they are received.

To handle these unknown words, the system first uses regular expressions to detect these phrases in a way that selects the argument and infers the argument's type, which could be a message name, list name, word phrase, sound name, and more. The system then creates lexicon rules to add these names to the grammar based on their type. Lastly, all unknown tokens are assumed to be number phrases and added to the grammar via lexicon rules. At the end of this process, the system is ready to parse and validate the input.

Recognize synonyms When engaging with natural language systems, children often use synonyms in place of rule and phrase key words [38]. To make our system more robust against such linguistic variations, we use a natural language toolkit [15] to find synonyms for keywords that may be substituted by synonyms.

The lexicon rules contain a list of terminal phrases that the system accepts. In general, these rules take the form of declaring a base word non-terminal in association with a list of synonyms (or a single word for reserved words like *variable*) that can replace the base word with little or no change to the meaning of the command. For example, the lexicon file contains a mapping of the non-terminal *Make* to the list *["make", "create"]*. Such a rule allows for both "`make a variable called` *x*" and "`create a variable called` *x*" to be accepted and be considered equivalent.

The system generates synonyms for a set of predefined non-terminals and adds these synonym sets (synsets) to the grammar as lexicon rules. This allows synonyms to be recognized and parsed correctly. The synonyms are identified by querying for all synsets in the natural language toolkit that contain the specified words that are also matched in the word's part of speech. All words in those synsets are gathered without repetition. Then, this synonym gathering component adds all synonyms of a word to the lexicon file as terminals mapped from the original non-terminal.[5]

Not all keywords can be substituted by its synonym. For example, a list is behav-

---

[5] The parser used by our system does not allow for terminal phrases with multiple words. Occasionally during the search for synonyms using the natural language toolkit, we found multi-word synonyms we liked such as "take off" for "subtract" and "clock time" for "time." To handle such multi-word synonyms, our synonym generator segments the multi-word synonym into multiple semantic rules joined together by a syntactic rule that maps the original non-terminal of the keyword to the synonymous phrase. This requires no change or modification of previously written syntactic, semantic, or lexicon rules, and thus is modular in terms of functionality. The system currently supports synonym phrases with up to 2 words.

iorally different from a set or a dictionary, though all of them are storage structures and could be synonyms for each other in terms of natural language. Another example is the keyword “volume”, which in our context, is how loud a sound is. However, people commonly use “volume” to refer to portion or amount of three dimensional space. As a result, most if not all synonyms of “volume” are likely to be useless to our system. Generated synonyms were manually filtered such that only the synonyms that were appropriate for substitution were kept and used. See Table 6.2 for a few examples of synonym commands that demonstrate the range of words (from nouns to verbs to adverbs) that were deemed appropriate to substitute.

Table 6.2: Examples of how synonyms were used to support multiple forms of the same instruction which all map to the same script.

Instruction	**Generated Script**	**Alternative Instructions**
“play the meow sound”	`[‘doPlaySoundAndWait’, ‘meow’]`	*“play the meow audio”*
“change volume by 5”	`[[‘changeVolumeBy:’, 5]]`	*“alter volume by 5”* *“vary volume by 5”*
“play the meow sound forever”	`[‘doForever’, [[‘playSound:’, ‘meow’]]]`	*“reiterate play the meow audio perpetually”* *“echo play the meow audio everlastingly”* *“recur play the meow sound always”*
“stop all sounds”	`[‘stopAllSounds’]`	*“discontinue all sounds”* *“quit all sounds”* *“block off all sounds”*

The ability to recognize synonyms makes our system more robust for those whose natural tendencies or memory of the commands do not match that specified by our original limited vocabulary. This is especially helpful for supporting children and people with cognitive disabilities.

Handling nesting in complex logic The main difference between natural language and drag-and-drop programming is that all the instructions will be given sequentially in time, regardless of the relation between one statement and another. However, the child may want to generate more complex code structures, like nested loops, using natural language.

Consider the instruction "`repeat the following 5 times <Action A> repeat the following 6 times <Action B>.`" There are two possible interpretations for this command. The first includes two separate loop structures. The first loop structure loops over *Action A* five times, and the second one loops over *Action B* six times.

```
Repeat 5 times:
        Action A
Repeat 6 times:
        Action B
```

The second interpretation has one nested loop structure. Wherein each passing of the loop involves performing *Action A* once, and repeating *Action B* six times. Overall, *Action A* is performed five times, whereas *Action B* is performed a total of thirty times.

```
Repeat 5 times:
        Action A
        Repeat 6 times:
                Action B
```

In this case, the ambiguous instruction produces two different interpretations with different behavior. Therefore, it is necessary to incorporate trigger words at the end of a loop instruction to indicate the end of list of instructions to loop over. However, in normal utterances, it is less natural to say "end the if loop", which would require the child to have a clear understanding of the structure of the program already. To make our system idiomatic, we chose to the phrase `thats it` to signal the end of a loop or a conditional command. To illustrate how this works, let us review the example given above. To generate code that follows the form of the first piece of code, the instruction will follow the form "`repeat the following 5 times <Action A> thats it` `repeat the following 6 times <Action B> thats it"`. In contrast, to produce the second piece of code the instruction will look like "`repeat the following 5 times <Action A> repeat the following 6 times <Action B> thats it thats it"` where the two `thats it` phrases close off the inner and outer loop, respectively.

As demonstrated above, we are able to distinguish between two loops behaving in a linear manner or a nested manner. For similar reasons, the trigger phrase `thats it` is implemented in conditional commands (if commands, if-else commands, etc.) and event commands (`when <Event A> do <Action A>`) as well to allow for unambiguous program generation.

6.6 Virtual Machine

Codi executes projects and steps in a virtual machine that runs the scripts generated by the NLP module from Section 6.5. The virtual machine is heavily influenced by the Scratch Virtual Machine (VM) [12]. The Scratch VM constructs and maintains the state of Abstract Syntax Trees (ASTs) [43] that each represent only the structural, content-related details of a program. The Scratch Virtual Machine maintains an AST for each target (programmable, code-running object) the user is programming. In Codi's case, there is a single target per project so only one AST is built and maintained when a project gets loaded.

Scratch VM and Codi's VM share the same targets, threads, runtime, and sequencer. *Threads* are sets of instructions can occur in response to particular events. Any instruction that is not attached to (a set attached to) a particular event, becomes attached to an event representing the project being played. Every thread in the virtual machine consists of at least one instruction. A stack frame stores metadata for executing the instructions and there exists a stack frame for every block in the stack. Some of these frames are empty. The *runtime* manages targets, scripts, and the sequencer. It may push, stop, and restart threads. The *sequencer* provides a runtime for threads and their targets. With the sequencer, you can either step a single thread or step all threads.

Each project in Codi is represented by a *project* module that uses a state machine to maintain the set of natural language instructions that form the project (For more about the project module, see Section 6.4). When Codi plays a project, Codi sends the natural language instructions to the NLP component, which processes each natural language instruction to generate a program specification represented in JSON. The program stores information about what sounds were used, what variables and lists were created, and what scripts were created from the natural language input.

To run a project, Codi passes all of the project's natural language instructions to the NLP component to generate JSON. Codi then passes the JSON Codi's virtual machine which loads the project. To execute the current step, Codi creates a new project definition that only contains the current step, and passes the JSON to the virtual machine to load and run the project.

Chapter 7

Evaluation

In structuring the evaluation for this project, I connect back to this project's three intersecting goals:

1. To create a platform for creative learning experiences accessible to children with visual impairments
2. To support children in engaging with computational thinking concepts and practices
3. To support children in expressing themselves through projects in ways that build creative confidence.

To evaluate Codi, I conducted testing sessions to see the extent to which children with visual impairments engage with parts of the creative learning spiral when interacting with Codi. I observe what mental models of the system were built and how these models were applied and changed by children as they interacted with projects. I also note the extent to which children are able to express themselves using Codi. In this evaluation, I draw from two sets of testing sessions, each set containing one hour-long session per student. The participants were two students at the Perkins School for the Blind. The two testing sessions were the 6th and 7th time I was meeting one of the students, who I'll refer to as Participant 1. For the other student, Participant 2, these sessions were first exposures to coding and Codi.

In this section, I highlight two main conclusions and dig into how I got to these conclusions: by analyzing and integrating excerpts from the test session transcripts that fall under specific themes that connect to the project goals.

7.1 An agent-based interface can facilitate meaningful creative learning experiences for children who cannot see despite usability challenges.

7.1.1 Creative Learning Spiral

First, we look at how children engage with the five stages of the creative learning spiral: imagine, create, play, share, and reflect when using Codi.

Imagine The agent-based interface was inspired by current technology and supported reflection on current technology. When Participant 1 imagined projects, he drew inspiration from the system's assets and from meaningful technology in his life, his calendar and his rhino toy which plays an electronic song.

> **Participant 1**: Because right now I kinda partially have in mind that maybe we could make like a pretend calendar project.
>
> **Facilitator 1**: Ooh, OK. Can you tell me more about this? What's your idea?
>
> **Participant 1**: Well. My idea is that we could say we could have a pretend event that would....we could basically have each event fit into each step.
>
> **Facilitator 1**: Okay. Yeah. What would an example of an event be?
>
> **Participant 1**: Well. Computer programming which we're doing right now is an event.

Facilitator 1: Mmhm.

Participant 1: Data could be an event.

Facilitator 1: Mmhm.

Participant 1: Maker Club could be an event.

On the other hand, Participant 2 found inspiration in the agent itself.

Participant 2: Can she have conversations with you?

Codi: `Exiting list *cues failure to understand* You are in the "Home" state. I heard you say "Can she have conversations with me".`

Participant 2: Oh, whoops.

Facilitator 1: Hehe, okay. Yeah. So that's the idea that she could have a conversation with you.

Participant 2: She can?!

Facilitator 1: Well, that's the goal.

Participant 2: Ohhhhh!

Facilitator 1: What do you think a conversation would be like?

Participant 2: So, like, how is your day? Or like, how are you doing?

Facilitator 1: Yeah. So right now. System doesn't really support that.

Participant 2: Maybe I can make a project.

Facilitator 1: Yeah.

...

Participant 2: `Whenever you hear` *`Are you excited for breakfast?`*`, say` *`I'm only a computer. I don't eat breakfast.`* `That's it.`

Codi: `*cues success*`

Participant 2: Okay.

Facilitator 1: Beautiful!

Codi: `OK. What's the next step?`

Participant 2: Oh, I have another idea.

Facilitator 1: OK, I'm ready for your idea.

Codi:`*cues failure to understand* I heard you say beautiful.`

Participant 2: Ooooohooooohohooo!

Facilitator 1: Hahaha, ok, so whenever you're ready just hit the space-bar. What's your idea?

Participant 2: I could say, "Do you like choc–whenever you hear. Do you like chocolate?" It could be like, "I'm an AI assistant I do not like–I don't have an opinion on that." Hahaha!

Facilitator 1: Hahah! sure!

Participant 2: `Whenever you hear` *`Do you like chocolate?`*`, say,` *`I'm in AI assistant. I don't have an opinion on that.`* `That's it.`

Participant 2 wanted to extend the behavior and personality of voice assistants so she took advantage of the opportunity to extend Codi.

Create Codi provided participants with multiple modes of creation, but they tended to make simple, similar projects. When Participant 2 and Participant 1 were creating projects, they used an iterative process in which they tested the project by playing it, reflected on the project, and then edited the project based on their reflection. This ties back to the computational thinking practices and tinkering.

Reviewing the projects created in the workshop reveals projects that express few computational thinking concepts. Most of them depend on speech synthesis and playing sounds, which are the simplest instructions. All projects changed the way the agent and the child and the agent related to each other. Some projects were practical (as in Participant 1's "What's on my calendar?" project) while others contributed to imagination, storytelling, and role-playing as skill building (as in Participant 2's "The baby is crying" project).

Even though Codi was designed to support projects that range widely in theme and complexity, Participant 2 and Participant 1 created projects that were quite similar and simple. It was hard to create more complex projects because it's difficult to verbally express and understand complex commands, like "`When the project starts, do the following forever set instrument to` *`piano`*`, play note` *`60`* `for` *`one`* `beat, play note` *`72`* `for` *`one`* `beat, play note` *`84`* `for one beat` *`two`* `times. That's it. That's it.`" However, Codi supports simple, one-line projects like "Surprise me" and "Say something nice" that provide great example for children to start with.

Play In these testing sessions, Participant 1 and Participant 2 expressed joy when their projects were successful, when the system made mistakes, when they made mistakes, and when they joked with their peers, the facilitator(s).

But rather than focus on these moments, I'd like to highlight how interaction with Codi raised opportunities for taking a playful approach to project creation. I saw this especially in testing sessions with Participant 2 in the first March testing session with Participant 2.

As Participant 2 was creating her first project "The baby is crying", Codi prompted Participant 2 for the next step. Participant 2 didn't know what to put in the next step.

> **Participant 2**: So should I just experiment and keep going?
>
> **Facilitator 1**: That's up to you. What do you want to do?
>
> **Participant 2**: I want to kind of see what would happen if I kept going with it.

In the transcript excerpt above, curious Participant 2 views the process of creating with Codi as exploration and experimentation. Early experiences creating with Codi involve discovery and integration of new evidence (gained by interactions) to build a model of what Codi is, what Codi can do, and how to use Codi.

> **Facilitator 1**: Okay so what we want is to... how do you want to change the program?
>
> **Participant 2**: Hmmm, I dunno I feel like it's still kind of confusing me I feel like I need to play like, I don't know, to get to know it a little more.
>
> **Facilitator 1**: Yeah... yeah, do you have questions that maybe I can help answer?
>
> **Participant 2**: I'm just curious, like I have to see it, kind of more like, on what it can do I guess

In the transcript excerpt above, Participant 2 acknowledges play as a method for learning in the face of confusion. Although Codi was meant to be a tool such that a child would be playing while making and interacting with projects, interacting with Codi itself is acknowledged as play.

Share In these testing sessions, we saw the role of peers in the collaboration between the facilitator and the child. As a facilitator, I also got to be a co-creator and mentor, supporting the students with their projects. The students shared questions, decisions, inspirations, and also feelings about the project with me. Together, we raised questions, debugged, and celebrated bug fixes. Pair programming with Codi made the project creation process a socially meaningful experience. See Subsection 5.2.4 for a snippet of a test session in which a participant debugs their project.

Reflect In the testing sessions, we saw reflections on the process, the project, and past experiences that influence the students understanding of their process, project, and the tool itself.

One category of reflections centered around ways to improve the project. After playing his "What's on my calendar" project for the first time, Participant 1 shared, "You know, I guess that there was one thing that I forgot to say before I said computer programming...I meant to say 'Here are your events.'" After playing a project, Participant 2 reflected on it and said, "I think I should edit that a little." After playing her "How are you doing?" project, Participant 2 evaluates her project and reflects on whether it is satisfying to her, "Do you think that's how she[the system] should reply? Hehe, I think she[the system] should." I also asked Participant 1 what he thought of his Rhino Song project compared to the other projects he worked on in the past. This brought him back to thinking about how he could improve the project. He said, "I thought it[the Rhino Song project] was pretty interesting. And I think it involved like more things I was curious about...such as identifying notes and like beats? Oh, I forgot to mention. Also, compared to the toy I try to replicate, the beats were in a pretty slow tempo."

Other reflections centered around the experience of making projects itself. After Participant 1 finished his "What's on my calendar?" project, I asked him "How did you feel after making the project?" and he said, "I felt pretty good about it because I felt like I was really close to an idea of how the calendar skill for something like Amazon Alexa was implemented." As someone who is interested in computer science and software engineering as a potential career, Participant 1 found it empowering and validating to create a project that mirrored functionality created by professional software engineers. When I asked Participant 2 what she thought of her experience, she said, "I mean, I really enjoyed it. The projects are really fun. It's fun to make projects." I asked, "Do you normally make projects, like, other kinds of projects?" to which she replied, "Not really. Not usually." Her feedback ties back to what I had learned earlier about the need for more opportunities and agency over projects for children without visual impairments.

7.1.2 Building and Applying Mental Models

In the test sessions, I sought to observe the ways in which participants built and applied models as well as the challenges faced in the process. I also sought to learn about the models the children developed. This section helps with understanding how

Codi supports computational thinking skills, one of which is building and applying mental models.

Children Use Different Strategies For Participant 2, playing the project first is a strategy for building a mental model of what instructions to expect before going straight into the code.

> **Facilitator 1**: Should I give you an example?
>
> **Participant 2**: Yes, please.
>
> **Facilitator 1**: OK, so I think I made a program. Let me see and try to figure out what I called it. I wrote it down. I think it was “Testing speech synthesis”. OK, let’s see. `See inside "Test speech synthesis"`.
>
> **Codi**: `Opening project "Test speech synthesis" for editing. There are two steps.`
>
> **Participant 2**: `Could I hear the project first or?`

Later on in the testing session when Participant 2 is in the “Inside Project” state, Participant 2 asks, “Do I–am I at that point where I can just say it[the instruction] or do I have to say add a step?” Later, she asks, “Should I ask what state I’m in?” These questions confirm that she’s developing and verifying her notional machine because they reveal that she knows what actions she can take to clarify her understanding.

Agents Inspire and Influence Model Building In the testing sessions, both participants sought to learn more about the agent and its implementation and how it relates to the other computing technologies in their lives. In particular, both participants found the sound cues intriguing.

Here’s a snippet of conversation from a test session with Participant 2.

> **Facilitator 1**: If you want to know if it’s listening to you that *imitates low-high cue* means it’s listening and *imitates high-low cue* means it’s not.

> **Participant 2**: OK. Oh, you changed that too.
>
> **Facilitator 1**: Yeah. Just make it more clear, hopefully. Does it make it more clear?
>
> **Participant 2**: Yeah! How'd you do that? You just–
>
> **Facilitator 1**: Oh, I went in–So, this entire website I coded myself so I can go in and change the code to use certain sounds.
>
> **Participant 2**: You find the sounds too?
>
> **Facilitator 1**: Mmhm! Yeah. I found a snap sound, and then I changed the pitch of a sound, and then I combined two sounds together.
>
> **Participant 2**: Oh, that's really cool.

Here's a snippet of conversation from a test session with Participant 1.

> **Participant 1**: So yeah. Yeah. Because I sort of–because when I–my mind went on to comparing. This to, like, Google voice recognition and I know how–I've gotten so used to Like with Google Voice Recognition, it can really, like it can–it's implemented in a way where I really know what each sound means at the end because when it recognizes it'll go *imitates low-high cue*. And then it will bring up the results. Well if it does not recognize it which is more rare, then it'll go like *imitates high-low cue* and then not do anything after that.

As students build and apply the mental models, they get a sense of the intelligence of the system. In the following scenario, Participant 1 plays the "Give me a compliment" project, and the second facilitator follows.

> **Participant 1**: `[System], give me a compliment.`
>
> **Codi**: *`You are amazing`*
>
> **Facilitator 2**: `[System], give` *`me`* `a compliment.`
>
> **Codi**: *`You are amazing.`*

> **Participant 1**: You know, uh, it can't hear different. It can't detect different individual voices. So I knew it was going to say the same thing because it detected the same words and it can't detect different voices.

Participant 1's commentary reveals his understanding of speech recognition and demonstrates how it decreases how intelligent he thinks the agent is. Participant 1 recognizes that Codi doesn't distinguish between speakers in giving a compliment and cites that as the reason the second facilitator got the same compliment has him. This implies that Participant 1's expectations for the agent were anchored by human-human interactions in which people give compliments customized to the recipient. However, "seeing inside" the project would reveal a single step: "`say` *`you are amazing`*." The real reason Codi says "You are amazing" is not because the speech recognition cannot distinguish between people. It is because the project instructions specify that Codi say "You are amazing." Here, we see an example of how an agent-based interface may encourage children to confer human-like reasoning to the agent.

Models Prompt Reflection on User Agency At the end of the first March session with Participant 1, I asked him, "If you had to explain how [the system] works to a friend, how might you do that? Like if you were introducing or telling them about the thing that you were working with and what you made, how would you talk about it?" He responded:

> **Participant 1**: Well. I would probably start by saying well [it] is this program that's sort of like where you can sorta make computer programming projects using your voice. I think that would be what I would start with. So. The way it works is you follow many different steps. For creating the project, it'll—it'll prompt you to do certain things to create the project. So. I think that's sort of giving people like a basic understanding of [it].

The way Participant 1 would present the system to a peer reveals that he considers the system to play a greater role in guiding and prompting whereas the user of the system follows the steps. His explanation doesn't convey much about user agency and creative possiblities other than that you "can sorta make computer programming projects".

Expressing Mental Models is Challenging At the end of the second March session with Participant 2, one of the counselors at the Grousbeck Center at Perkins asked her, "So what were you working, on?" Participant 2 hesitated to respond and deferred to me to share that we were creating programming projects using a tool I made. An agent-based interface for creative programming is software that presents itself as a sort of black box. You can get a mental model of how to use the black box based on its behavior and the feedback, but it is still challenging to communicate and synthesize information about the system's behavior. Presenting Codi effectively to those who have not interacted with it before involves connecting to concepts and experiences of the listeners—something Participant 2 did not have much time to explore.

7.1.3 System Intelligence

The lack of intelligence of the system decreases the usability of the system while adding an element of silliness and play. Participant 2 regularly confirmed commands with me before engaging with the system, indicating that she didn't remember what commands would work and also that she knew the agent was not smart enough to handle many of the ways she might express a particular command.

> **Participant 2**: Should I–How do I say it? See inside "Say something nice"?
>
> **Facilitator 1**: Mmhm.
>
> **Participant 2**: `See inside "Say something nice."`

Students also saw the system's lack of intelligence in the sound library search results.

Facilitator 1: I think it[the System] didn't understand. Oh let's try. `Do you have any baby sounds?`

Codi: `I found 24 sounds.`

Participant 2:Oh!

Codi: `Here's one called clown honk.`

...

Participant 2: `Do you have any` *`crying`* `sounds?`

Codi: `I found one sounds. Here is one called thunderstorm.`

Participant 2: Hahah, oh boy.

The students usually found joy in the mistakes made by the system due to the system's limited listening skills, knowledge, and understanding. This joy contributed to the playful experience of interacting with an agent in order to create a project. However, it would have been better for the kids to find joy inherent in their projects and the process of designing and making them rather than in the silliness or absurdity of some of the interactions they had with Codi. The silliness and absurdity component of play was not anticipated, but contributes to a sense of randomness, exploration, and surprise that goes along with play.

The intelligence of the system influences the relationship that the agent has with the child. A highly intelligent system that is able to understand natural language can increase usability by reducing friction and making it easy to execute commands. This would give the child more time and opportunities to express themselves, to have power as a creator. One way in which Codi supported a child's creative process while reducing the cognitive load is through its support of slot filling (See Section 6.3 for more on slot filling).

However, a more intelligent system may be harder for a student to model mentally. Children build a model of how the agent works by interacting with it, and an intelligent agent may have more complex behaviors that do not get explained. Without explanation, a student is left to infer and can build an inaccurate mental model of an agent-based system. During the testing sessions, there were several times in which the student was confused about what state Codi was in, and this gave them some trouble using Codi as a creative tool until they corrected their mental models through feedback or guidance. Incorrect mental models lead children to attempt actions that either they cannot do because of what state they are in or take them into unexpected states. Participant 2 would forget that Codi wasn't actually listening to her and wouldn't until she pressed a key. Participant 1 forgot about the need to finish a project before playing it.

7.1.4 Usability Challenges

Although Codi is designed to make it easy for children to get started with playing and creating projects, there were 6 key factors that reduced the usability of the system. Here they are, each with an example:

1. State machine transitions and thus current state are not always clear.

   > **Participant 1**:`What is step` *`two`*`?`
   >
   > **Facilitator 1**: You may want to check what state you're in right now.
   >
   > **Participant 1**: *starts speech recognition* *stops speech recognition* Hmm.
   >
   > **Facilitator 1**: I could also tell you. You're currently in the 'play project' state. So if you want to modify the project, you have to be inside the project first.
   >
   > **Participant 1**: Hm. `[System], see inside "Rhino Song".`
   >
   > **Codi**: `Opening Project Rhino Song for editing. There are two steps.`

2. The speech recognizer makes systematic mistakes.

   > **Participant 1**: `Play Note` *`90`* `for` *`one`* `beat.`

Codi: `I heard you say play note 94 one beat.`

Facilitator 1: How do you prefer to add the note?

Participant 1: I'd rather just–I think I'd rather just type it.

3. Codi disrupts flow when giving feedback in response to incomplete utterances.

   Participant 2: `... Insert..`

   Codi: `... Insert..`

   Codi:`*cues failure to understand*`

   Participant 2:Sorry!

   Facilitator 1: No, don't be sorry!

   Participant 2:So I just say insert–

   Facilitator 1: I just paused to the mic. Whenever you're ready!

   Participant 2: So I just say insert before Step 3.

4. Conforming to grammar is difficult.

   Participant 1: Okay so. I'm not sure the command for editing.

5. Supported commands are not flexible enough.

   Participant 2: `Replace step` *`one`* `with` *`say I think you're very beautiful and I like your personality`*

   Codi: `There is no step one with say I think you're very beautiful and I like your personality. What step number?`

   Facilitator 1: `One.`

   Participant 2: *`One.`*

   Codi: `What is the instruction?`

   Participant 2: *`Say I think you're very beautiful today.`*

   Codi: `Replaced Step 1.`

The system didn't properly pull arguments from Participant 2's command to replace a step even though it was intuitive. Given the difficulties, it seems like Participant 2 decided to reduce the complexity and length of her instruction.

6. Bugs can surface in non-obvious ways.

> **Participant 2**: Next, `add a step.`
>
> **Codi**: `No more steps.`

Here, Participant 2 is asking to add a step to the project. However, Codi recognizes "Next" in the command and interprets Participant 2's utterance as "What is the next step?" because the trigger is too broad. When Codi responds, "No more steps," Participant 2 could have interpreted this as the system telling her that she can't add any more steps—the opposite of what we want! We want the system to encourage creation not fight back against it.

Bugs reduced the usability of the system by creating confusions as the student built a mental model of the system. These confusions were addressed by the facilitator immediately to address misunderstandings. However, this puts the child in backseat role during that time and limiting the creative possibilities. Despite these challenges, children can still get comfortable and confident creating with Codi.

7.1.5 Boosting Usability

Codi's multimodal interface, predictable conversations flows, and support for questions made it easier to learn and create with Codi.

First, the voice interface provided an alternative to the keyboard for children unfamiliar or uncomfortable using a computer and keyboard. Although it is designed for the screenless experience, Codi also provides a visual interface that supports collaboration between people of different abilities, such as someone with visual impairments working with someone without visual impairments. The multimodality of Codi means that Codi takes advantage of the interfaces available. This also supports the use of Codi in different contexts, by different people. Someone who cannot speak out loud, perhaps because they are in a library, may want to type to Codi instead of talking aloud to Codi.

Second, the state machine had predictable and easy-to-follow flows like the project creation and project editing flows. Children can express their intent to Codi through a range of expected phrases, but Codi suggests the simplest phrases when giving examples the child can easily recall and say them in future interactions. Through interactions with the system, a child builds a model of what phrases work well for executing different transitions. These phrases are meant to be simple. When they are not simple because they require arguments, Codi conducts slot filling. Consider the instruction "`Replace step` *`1`* `with` *`say I think you're very beautiful today.`*" A child can do this action in parts as Codi ensures that the two mandatory arguments for replacing step have been specified: the step number to replace and the instruction with which to replace the step.

Child: `Replace a step.`

Codi: `What step number?`

Child: `1`

Codi: `What is the instruction?`

Child: `Say` *`I think you're very beautiful today.`*

Codi: `Replaced Step 1.`

The Codi system, a state machine, only allows state transitions consistent with established flows for creating, editing, and playing and playing a project. When a child got confused about where they are, they asked "What state am I in?" and "What step am I on?" to situate themselves. However, they didn't take advantage of "What can I do now?" which would have illuminated possible next steps for continuing to create and explore with Codi. This is because Participant 1 and Participant 2 were not at a loss for what to do. They had a clear idea of what they wanted to implement in the system, and were looking for how to do something rather than for something to do. The "What can I do now?" question is still useful for showing how to accomplish a task because Codi answers the question by suggesting an action and demonstrating what to say to execute the suggested action.

Participant 1 and 2 drew from Codi's existing sound assets and asked Codi questions to learn about what assets were available. When they asked Codi, "What sounds are there?", Codi presented three sounds at a time by playing the sound and then saying the name of the sound. This reduced the cognitive load of processing and remembering sounds so that Participant 1 and 2 could easily go back to their projects and integrate sounds of interest or record their own.

As mentioned in Subsection 3.2.2, Baldwin et al. identified three problem areas within audio-only interfaces: ephemerality, linear interaction, and unidirectional communication. Codi still has ephemerality and linearity because Codi's primary output is sound. However, Codi addresses the third problem area of unidirectional communication by presenting a conversational interface that aids in mitigating the problem of ephemerality. The ephemerality of sound requires those interacting with Codi to store what they hear in working memory. Codi alleviates this by answering questions like "What did you say?", "What did I say?", and "What state am I in?" whenever the user asks.

At the end of the first March session with Participant 1, I asked, "Do you think that you'd feel comfortable playing with [the system] on your own? Like if I could give you a link to the website and you could just go on and play with it." He responded, "Yeah."

7.1.6 Supporting Self-Expression

In the March test sessions, I also aimed to evaluate the extent to which children could express themselves with Codi in a way that builds their creative confidence. This ended up being quite challenging to evaluate without a baseline: what was their level of creative confidence before programming with Codi? What were their interests and ideas before interacting with Codi? Rather than compare to a baseline, I aimed to see what I could learn from participants from the projects they created and to understand how intentional their self-expression was.

Participants expressed aspects of themselves in through project ideas and the behaviors they sought to explore. However, in most cases, the participants were not intentionally trying to share something about themselves in the project. Participant 2 created a project that expressed her values:

> **Participant 2**: *Say something nice.*
>
> **Codi**: Ok, when you say "[System], say something nice," I'll play the project. What's the first step?
>
> **Participant 2**: Say *you are very pretty and beautiful. I like your personality.*
>
> **Codi**: Ok, what's the next step?
>
> **Participant 2**: I'm done.

From this simple project, I could see that Participant 2 cared about beauty—both inside and out. Her project also conveys information about how she wants to agent to relate to her. Yet Participant 2 did not intend for me to interpret her project in this way, so I would argue that this is not intentional self-expression.

Participant 1 liked music and wanted to recreate a song from Rhino, one of his childhood toys. Codi supported Participant 1's self-expression and learning by giving him a new platform on which to create his own version of the Rhino Song. However, creating this song with Codi was challenging and we spent a large amount of time troubleshooting the system mishearing our commands (e.g. "play note *90 four* times" misheard as "play note 94 times"). At the end of the test session, Participant 1 shared, that he "was hoping that we would be able to like do like the at least the first measure of the song."

Codi provided opportunities for self-expression, but participants did not explicitly make the conscious choice to use it as a medium for self-expression. Furthermore, Codi's usability challenges hindered self-expression as well by slowing down the project creation process.

7.2 Onboarding, learning resources, and facilitation are needed to effectively promote engagement with computational thinking concepts and practices.

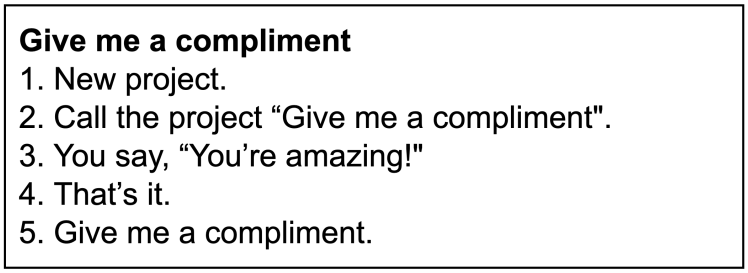
In order to support children's engagement with computational thinking concepts and practices, agent-based programming interfaces need onboarding experiences, learning resources, and facilitation that helps learners surface and pursue their own interests and integrate computational thinking concepts.

7.2.1 Computational Thinking Concepts

The March testing sessions revealed that although the Codi system alone can support a subset of computational thinking practices, it is difficult to hit learning goals related to computational thinking concepts without proper onboarding, learning resources, and facilitation. This was especially clear when comparing the Wizard of Oz tests I did as part of my iterative design process to the March testing sessions I did to evaluate my system.

Wizard of Oz Tests In the Wizard of Oz tests, I simulated the Codi system, and the child programmed through conversation with me. As I did this, I bounced between being a facilitator and being the system. The child had access to cards that noted exactly what they could say to Codi to create projects intended to either broaden the kind of project a child could make or introduce a computational thinking concept. These cards were printed in Braille for children without vision and large-print (size 30 font) for children with low vision.

The first card in the set introduces how to use Codi's speech synthesis functionality. It also presents and example of framing a project as the instructions for executing a specific command (See Figure 7-1). It has one step so that a child can quickly feel a sense of success and experience the project creation flow.

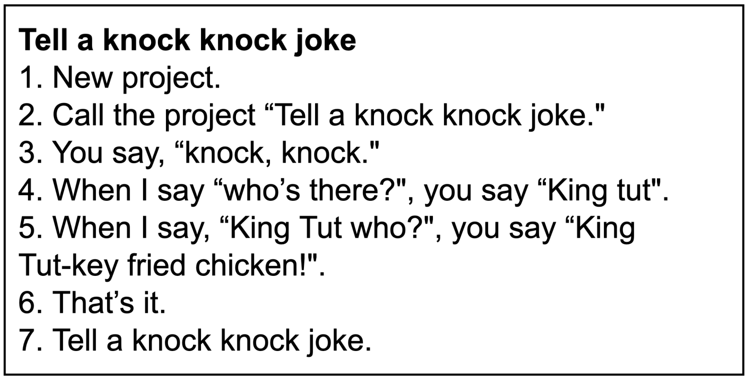


Give me a compliment

1. New project.
2. Call the project “Give me a compliment".
3. You say, “You’re amazing!"
4. That’s it.
5. Give me a compliment.

Figure 7-1: Resource card with instructions for creating a “Give me a compliment” project.

The next card (Figure 7-2) introduces the ability for a project to receive input and react to events using speech recognition. It’s a bit more involved, but highlights the potential for programming conversation flows to extend the agent’s abilities.



Tell a knock knock joke

1. New project.
2. Call the project “Tell a knock knock joke."
3. You say, “knock, knock."
4. When I say “who’s there?", you say “King tut".
5. When I say, “King Tut who?", you say “King Tut-key fried chicken!".
6. That’s it.
7. Tell a knock knock joke.

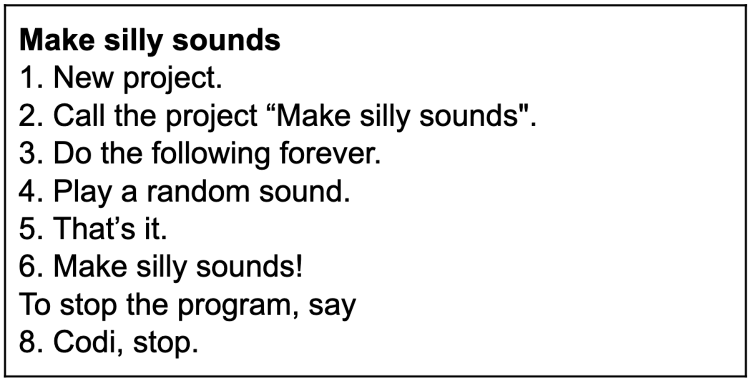
Figure 7-2: Resource card with instructions for creating a “Tell a knock knock joke” project.

The third card (Figure 7-3) introduces two concepts: randomness and looping. It is a great project to introduce because the element of surprise makes it simple and fun.

The fourth card (Figure 7-4) shows an example of how a child can change the way a sound is played in a program. The project exemplifies a domain of projects that helps contribute to imagination, storytelling, and scene-setting.

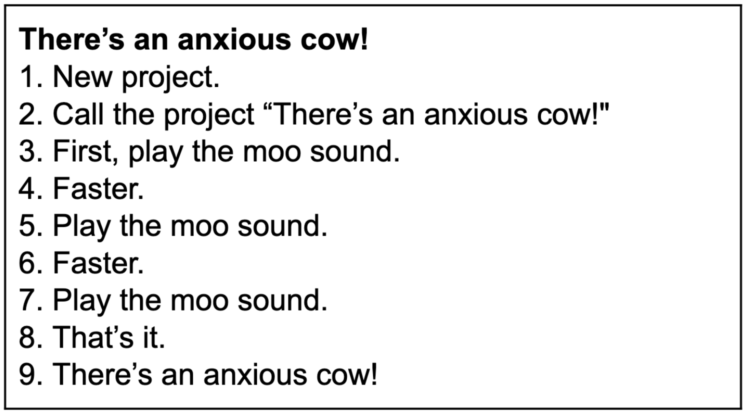
An additional card (Figure 7-5) extends the introduction of sound commands with a list of sound commands the child can try.

The child is then given the opportunity to explore different sound commands. The facilitator may use the next card (Figure 7-6) to introduce looping a specific number



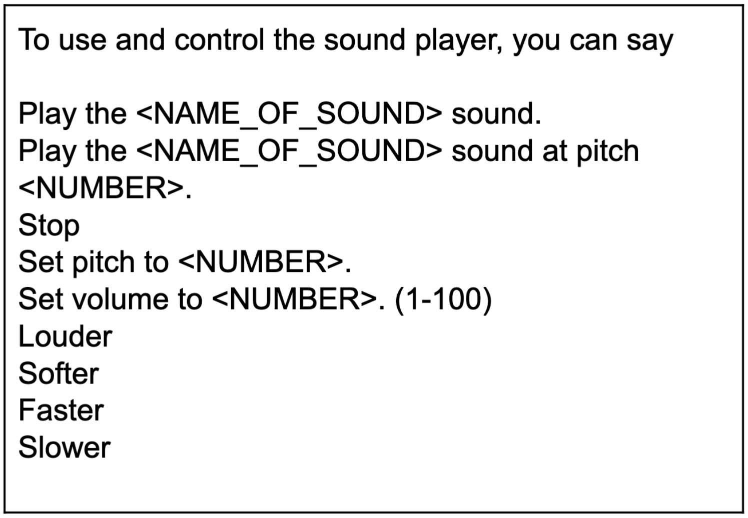
Make silly sounds
1. New project.
2. Call the project “Make silly sounds".
3. Do the following forever.
4. Play a random sound.
5. That’s it.
6. Make silly sounds!
To stop the program, say
8. Codi, stop.

Figure 7-3: Resource card with instructions for creating a “Make silly sounds” project.



There’s an anxious cow!
1. New project.
2. Call the project “There’s an anxious cow!"
3. First, play the moo sound.
4. Faster.
5. Play the moo sound.
6. Faster.
7. Play the moo sound.
8. That’s it.
9. There’s an anxious cow!

Figure 7-4: Resource card with instructions for creating a “There’s an anxious cow” project.

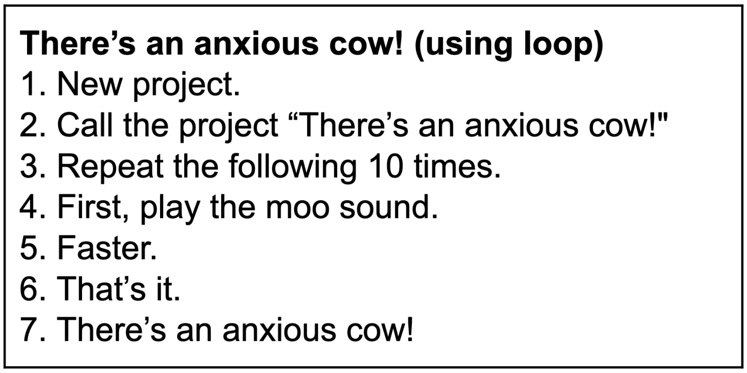


To use and control the sound player, you can say

Play the <NAME_OF_SOUND> sound.
Play the <NAME_OF_SOUND> sound at pitch <NUMBER>.
Stop
Set pitch to <NUMBER>.
Set volume to <NUMBER>. (1-100)
Louder
Softer
Faster
Slower

Figure 7-5: Resource card with list of sound commands understood by Codi.

of times as a way to prevent unnecessary repetition in the code. The child can see that rather than repeat “`Play the moo sound`” and “`Faster`”, the child can specify these instructions once inside of a loop that repeats these instructions. This extends the previous introduction of loops by giving the child the chance to pick the number of iterations in the loop.



There’s an anxious cow! (using loop)
1. New project.
2. Call the project “There’s an anxious cow!"
3. Repeat the following 10 times.
4. First, play the moo sound.
5. Faster.
6. That’s it.
7. There’s an anxious cow!

Figure 7-6: Resource card with instructions for creating a “There’s an anxious cow” project using loops.

The rest of the cards are references so that a child has a tangible list of commands that can be used. At the time of the Wizard of Oz tests, the cards were seen as a crutch. Ideally, the system would answer any questions the child has about how to specify certain commands and what kinds of commands are available at different points in the system. See Appendix B.1.1 for the rest of the cards.

Projects created during Wizard of Oz Tests touched on computational thinking concepts beyond sequences and events. Below, I share the instructions as provided by the participant. They are not always perfectly formulated. I keep these imperfections to show the kinds of interactions that may happen.

In the “Only Bouncy Sounds” project, the participant builds a sequence of instructions using sound commands to output a sequence of increasingly high pitched bouncy sounds.

Only Bouncy Sounds

`First set pitch to` *`1`*`.`

`Then, play the` *`bouncy`* `sound.`

```
Then, set pitch to two.
Then, play the bouncy sound.
Set pitch to 5.  Oh no, I mean 3.
Then, play the bouncy sound.
Then.  Set pitch to 3.
Then, play the bouncy sound.
Then.  Set pitch to 4.
Then, play the bouncy sound.
Then.  Set pitch to 5.
Then, play the bouncy sound.
That's it.
```

The “Only Bouncy Sounds” project is a great use case for loops. The participant created “Fast bouncing” as a simple first project integrating loops. However, the sequence of instructions was incorrect. Placing the repeat command at the end resulted in the the bouncy sound playing only once.

Fast bouncing

```
Set pitch 10.
Then play the bouncy sound.
Repeat the following forever.
That's it.
```

When I provided the exact example of how to use the forever loop, the participant successfully used the loop command to create a project that plays the bouncy sound ten times.

Bouncing forever

```
First repeat the following forever.
Set pitch 10.
Then, play the bouncy sound.
That's it.
```

The participant aimed to modify the project to make the sound alternate between pitches.

Alternating Bounce (Attempt 1)

```
Repeat the following forever.
Set pitch 10.
Then play the bouncy sound.
Repeat the following forever.
Set pitch 20.
Then play the bouncy sound.
That's it.
```

The output of the project was a bouncy sound of pitch 20 being repeated over and over. When I asked the participant what about the output was unexpected, he said that the pitch was staying at 10. This was incorrect so I told him that it's actually at pitch 20 and why. The participant might not have understood that the computer interprets the actions into specific commands and follows these exactly. When I showed him the correct program, he understood his mistake.

Alternating Bounce (Final)

```
Repeat the following forever.
Set pitch 10.
Then play the bouncy sound.
Wait .5 seconds.
Set pitch 20.
Then play the bouncy sound.
Wait .5 seconds.
That's it
```

With facilitation and resource cards, the child was able to create projects that integrated loops and sound editing rather than engage only with sequences and events.

March Testing Sessions As a facilitator during these March testing sessions, I did not guide the student towards particular concepts, and I did not provide physical, printed or embossed project cards. Instead, the sessions were guided by ideas that a child had for a project they wanted to do. Their project creation process took an iterative approach in which simple basic functionality was accomplished before moving on.

In the projects created during the March testing sessions (See Appendix A.1 and A.3), sequences and events were the only computational concepts explored. The participants effectively explored sequences whenever they were inside a project, asking about the current, next, or previous step, or modifying a project by inserting, deleting, or adding steps. Sequences are core to children's understanding that their projects are programs made of sequences of instructions. While the participants explored sequences effectively, they faced barriers when exploring events.

The only event used in these testing sessions is based on speech recognition. I made the "Test speech synthesis" project to demonstrate the "Whenever I say" event using the command "`Whenever I say` ***`hello`***`, say` ***`hi friend`***`. That's it.`" However, playing the project confused the participant because both Codi and the project handled the "Hello" meant only for the project. Codi handles the hello by responding with a greeting.

> **Facilitator**: `Test speech synthesis.`
>
> **Codi**: `*cue project start*`
>
> **Facilitator**: `Hello.`
>
> **Codi**: `What's up?`
>
> **Facilitator**: `Hello.`
>
> **Codi**: *`Hi, friend.`*
>
> **Codi**: `*cue project end*`

This behavior is confusing because those listening to the system must determine what to attribute behavior to, the agent or the project played by the agent. Later, Participant 2 creates the "Talk to me" project relying mainly on the "Whenever I say" event. In order to trigger the "Whenever I say" event, the system must be executing the "`Listen and wait`" instruction. We had trouble continuously listening and waiting, which made it difficult to test the event handlers Participant 2 coded in the project. This made our exploration of events less successful.

7.2.2 Computational Thinking Practices

Across the four sessions, I found nine instances in which debugging occurred, one instance where reusing and remixing occurred, and one instance where being incremental and iterative occurred. There were zero instances where the idea of abstraction and modularity was discussed or brought up as something the student was aware of. In the rest of the section, I present key conversations from the test sessions that exemplify computational thinking practices.

During one of the testing sessions, Participant 2 and I (Facilitator 1), collaboratively debugged a project created by Participant 2. This collaboration highlighted that it's not just helpful to investigate current state of the program. When debugging, it's helpful to know what changes were done to the program and easily undo those changes. Codi doesn't yet store this, but as a facilitator, I was still able to go back into the history to view the sequence of steps:

> **Facilitator 1**: Okay, so this is a sequence of steps we took. We said, "[System], see inside 'Rhino Song'", "add a step", "set instrument to". That didn't work. And then we said, "Set instrument to piano."
>
> **Facilitator 1**: And then and then later we said, "Insert set instrument to electric piano before step one." And then we said, "Delete step one." I mean "Step three." I guess...
>
> **Participant 1**: Yeah. You know what I said—I—Now I know what happened.
>
> **Facilitator 1**: Yeah?

> **Participant 1**: I must have deleted—I must have forgotten that—I must have been thinking primarily about step one and then not realized that then there were four steps because the added step was step four. But I was thinking about step one. So I think I must have deleted the—
>
> **Facilitator 1**: The second.
>
> **Participant 1**: The second note.
>
> **Facilitator 1**: Yeah. It's awesome that you figured that out!

Another example of iterative development based on testing periodically:

> **Participant 1**: Yeah, because I'm actually not quite sure if I got the right note. I might have gotten it either a little bit lower or a little bit higher, but let's test it out and see.
>
> **Facilitator 1**: Yeah. All right. If you're ready, you can play the project.

7.3 Feedback

There are many ways to improve Codi. During the sessions, we received feedback from the students. The majority of explicit feedback came from Participant 1. When he was creating his Rhino Song project, he learned that he needed to insert "`set instrument to` *`electric piano`*" at the beginning of the project to hear the note play.

> **Participant 1**: `Add a step.`
>
> **Codi**: `What is the instruction?`
>
> **Participant 1**: `Set instrument to` *`electric piano`*`.`
>
> ...
>
> **Codi**: `Ok, what's the next step?`
>
> **Participant 1**: `I'm tempted to say move step three to step one to try to move it.`

Here, I realized that although I was providing the basic vocabulary needed to make the change that he wanted, a more intuitive interface would draw analogies to the physical world, suggesting that one could move a step like a physical object and the steps after it would also shift.

There were also instances in which the students' attempts at commands informed how the grammar could be made more flexible to better support natural language. Both participants struggled to remember how to see inside a project and open it for editing. Participant 2 asked, “You say ‘`Edit` *`The baby is crying?`*‘”, and in a different session, Participant 1 asked, “Is there a command like maybe say ‘check the steps’?”

Feedback from test session participants suggest that Codi would be easier to use if Codi's commands for manipulating program drew from physical analogies and allowed children to customize Codi's audio cues.

7.4 Methodology

In the testing sessions, I excluded the braille and large-print learning resource cards used in previous iterations as described in Section 7.2.1. I relied on the system and facilitator to provide guidance and fill in the blanks as as the facilitator and child worked together. These workshops were characterized by collaboration because I recognized the importance and power of peers and collaboration on promoting play in the process. As a facilitator, I was a mentor aiming to help the students realize their ideas through a program.

I used a service to automate the transcription of the audio and then reviewed the transcripts to correct inaccuracies in speech recognized and speaker attribution and include missing sound cues and project executions. I apply grounded theory [52] to analyzing transcripts from the testing sessions. See Figure 7-7 for the code.

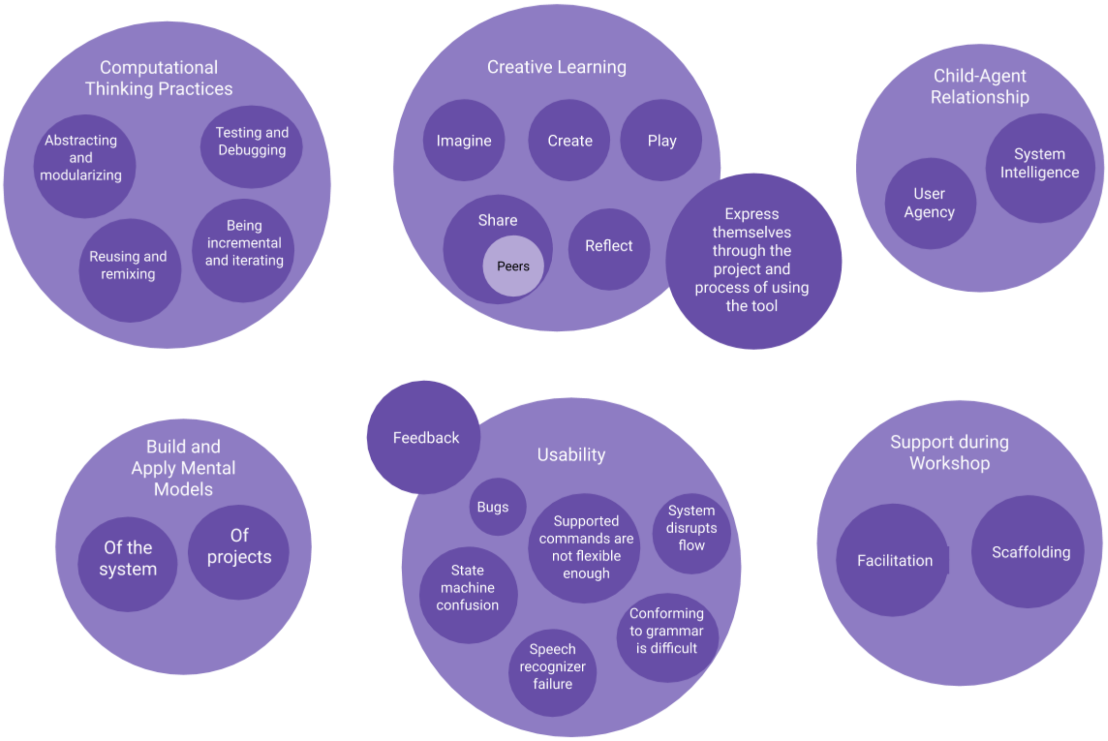

Computational Thinking Practices
Abstracting and modularizing
Testing and Debugging
Reusing and remixing
Being incremental and iterating
Creative Learning
Imagine
Create
Play
Share
Peers
Reflect
Express themselves through the project and process of using the tool
Child-Agent Relationship
User Agency
System Intelligence
Build and Apply Mental Models
Of the system
Of projects
Feedback
Usability
Bugs
Supported commands are not flexible enough
System disrupts flow
State machine confusion
Conforming to grammar is difficult
Speech recognizer failure
Support during Workshop
Facilitation
Scaffolding

Figure 7-7: The code system, derived from the project goals, consists of six groups of codes. Hierarchy in the codes denote relationships between ideas in the same group.

7.5 Summary

7.5.1 Findings

Despite usability challenges, an agent-based interface can facilitate meaningful creative learning experiences for children who cannot see. These experiences also include opportunities for children to express themselves. However, an agent-based programming interface alone is not enough to promote engagement with computational thinking. Onboarding, learning resources, and facilitation are necessary to surface opportunities for connecting to computational thinking concepts.

When children *imagine* what's possible to create with Codi, they are often inspired by Codi, and *create* projects that add to Codi's behavior and personality. Codi did not inspire a diversity of projects where there was little scaffolding and learning resources provided. Because Codi provides a non-traditional interface for programming, Codi prompts *play* as a way to learn how to use and create with Codi. In these test sessions, we clearly saw collaboration between multiple people creating projects and we saw *sharing* of projects between the facilitator(s) and the participant. Finally, the Codi test sessions also involved participants *reflecting* on both the creative process and thinking about ways to improve the projects.

From this evaluation, we saw children use different strategies for constructing models of system behavior saw how agents inspire and influence model building. We also saw that intelligence can increase usability at the cost of increasing system complexity and risking children having difficulties building accurate models.

7.5.2 Limitations

This evaluation involved two students from Perkins who participated two test sessions each. The evaluation would have been more meaningful if done with more students with visual impairments, not just the ones at Perkins. It would have also been informative to conduct test sessions with sighted children to see how they approached using both the different available modes of interaction with Codi. Furthermore, these March test sessions did not use learning resources and structured onboarding that may have resulted in deeper engagement with computational thinking concepts and a greater variety of projects created. As I coded the transcripts, I noticed that I missed many opportunities to ask the participant for their theories and thoughts before giving an answer or explanation of the system. This makes it hard to evaluate understandability of the system.

Chapter 8

Conclusion

8.1 Contributions

I designed and developed a tool and platform for children to create programming projects using voice and natural language. This extended existing work on creative learning experiences for children with visual impairments, child-robot interaction, programming by voice, and programming through natural language. I researched the educational context of supporting children with visual impairments by conducting interviews, co-designing, and running test sessions of Codi with students at Perkins School for the Blind. Although this tool was designed with a focus on accessibility for children without vision, the tool is accessible to children with limited motor skills, and also supports children who cannot speak aloud with by allowing them to type what would normally be spoken to Codi.

In this paper, I share what I learned about engaging children with visual impairments who may have multiple disabilities in creative learning experiences. I provide a framework for designing an agent as a tool and an architecture for implementing an agent-based programming interface. My evaluation of Codi reveals that agent-based programming interfaces can facilitate meaningful creative learning experiences for children who cannot see, but that onboarding, learning resources, and facilitation should be carefully considered to promote engagement with computational thinking concepts and practices.

This project provides a powerful use case for applying natural language processing to programming language design to help lower the barrier to entry to engaging with computational thinking and give more people the ability to program and make change in the digital world around them.

8.2 Future Work

At the end of my test sessions, Miriam, who teaches computer science and runs Maker Club at Perkins School for the Blind, shared that Codi is an immensely valuable tool for children at Perkins to try programming and engage in project-based learning. With Codi, children can make computer programs and sketch out programs they may want to develop further on other platforms. Miriam asked me what might happen next, and I saw many ways to iterate on this project—there is so much more to explore in the domain of agent-based programming interfaces for children.

Future iterations could have the potential to support meaningful connections between children using Codi. They could give children more agency by allowing them to shape how Codi communicates with them. They could make it easier to tinker with Codi and widen what and where a child can create with Codi. Future iterations could make Codi more conversational so that children face fewer obstacles in the creation process. They could also extend Codi's capabilities as a programming language to support stronger engagement with abstracting and modularizing—important skills in system design and communication.

8.2.1 Promoting Community around Codi

As I mention in Section 5.4, interviews with faculty at the Perkins School for the Blind, highlight that promoting community and connections between peers is hugely important for these children. Currently, children can collaborate in-person to create projects with Codi. However, they cannot yet store and load projects from peers that were not made on the same device. A social experience within Codi where you can view other's projects—including those created on a separate device—could promote community for peer learning and exploration. This could be executed using an online community with cloud storage so that people can access other's projects via links. For an offline community, children may want to export and import Codi project files.

Children could share projects by telling the agent what projects to make public, and discover others' projects by asking something like "What projects did Tina make?" They could also save their project, send it to their peers, and have their peers load the projects on their own devices. As an agent, Codi could also frame projects in a way that gives credit to the creator of the project and encourages the audience to remix the project. For example, Codi could say, "Did you like that project from Tina?...Well, what do you want to change about it? We can remix!" We also imagined how a child might want to remix a project, "Codi, change your voice to a robot for this section!" or "Codi, I didn't like that at all. Remove that line!"

8.2.2 Agent Customization and Personalization

Codi responds to every person the same way, no matter who they are and no matter what past interactions they have with Codi. Future work could explore how Codi may relate to a child and support children's creative learning long-term. By allowing the child to customize the agent, we can explore and engage with issues of representation in agent personality and child-agent dynamics. Additional research may investigate whether customization can influence a child's creative learning experience and engagement with computational thinking.

Agents may also use affective computing[1] to better relate to the child and understand and react to the child's mood to support a child in the creative learning process. This raises questions about ethics because the agent will need to make judgements about the child's mood, but feedback from the child may be meaningful and personalized to a child rather than drawing from generalizations over data that is not representative of all demographics of children.

---

[1] Affective computing is "computing that relates to, arises from, or deliberately influences emotion or other affective phenomena" [62].

8.2.3 Increasing Tinkerability through Immediate Feedback

While editing a project with Codi, a child can test their project or execute the current step as a way to get feedback about their project. This helps the child iterate on and explore their projects. Codi can be more tinkerable with a "playground" mode that gives children immediate feedback in response to their instructions. This "playground mode" is inspired by REPLs (Read–eval–print loops). REPLs are interactive computer programming environments that take single user inputs, evaluate them, and return the result to the user [57]. REPLs facilitate exploratory programming and debugging because the programmer can apply instructions incrementally and inspect the result at each step. The read-eval-print loop involves the programmer more frequently than the classic edit-compile-run-debug cycle [57]. In the "playground" mode, Codi would interpret user's utterances as programming instructions and execute them immediately instead of storing them in a project. A "playground" mode would also be valuable for a child learning what project instructions Codi understands and how Codi understands them.

Another approach to presenting immediate feedback to the user is to have multiple agents form the interface. One "editor" agent, like Codi, facilitates the project editing. A second, "performer" agent acts out the instructions. This could also work well in a collaborative setting where one person creates with the "editor" agent while the collaborator tests the project by interacting with the "performer" agent. Introducing a third, "tester" agent could allow a child to write automated tests in which the "tester" agent and "performer" agent communicate with each other. This would explore how agents and humans may work together and play roles on a team in a way that reflects the sort of collaboration expected in the real world.

A multi-agent approach can provide new ways to understand and program with agent-based interfaces. The design of multi-agent interfaces for programming can naturally draw from human team dynamics. For example, having agents play specific roles can break up the complexity of interaction with a single agent into simpler interactions with multiple agents. However, an interface or team integrating multiple agents could be complex based on expectations and the potential need to manage who knows what. If you tell a single agent something, do you expect all the agents to know? Do you need to tell each one? Raising questions like these, a multi-agent approach may connect to theory of mind.

8.2.4 Extending What You Can Create with Codi

Building out Codi's programming interface to support more mediums may inspire a diversity of projects that range in complexity and interests expressed. Codi projects currently only use audio output and control the agent itself. However, the ability to program physical objects gives children opportunities to shape and engage with their world [24, 42]. Codi's programming language could be extended to support programming physical devices like LEGO programmable bricks and micro:bit [14].

Codi's speech and music commands could also be improved. In Codi, children can make music using instructions that specifying instrument, note, and duration of note. Another way they can engage with music is to use and remix Codi's musical sound assets. We could add additional commands to Codi; children could use these commands to remix, break down, and build on songs they like. The speech commands children can use in Codi projects are limited to a small set of voices. Expanding this set would increase the expressiveness of speech commands in Codi and especially support storytelling projects in which a Child may want to have multiple characters speak in specific voices.

8.2.5 Extending Where You Can Create with Codi

Codi is a web application designed for a desktop experience even though it supports a screenless experience of creating programming projects. Since Codi has an interface that only requires a single key, a microphone, and a speaker, Codi could work well on mobile devices with small screens. Codi could also work well as an app for voice assistants in which certain wake words cue the voice assistants to begin processing and responding to user speech. Future work could bring Codi to a mobile platform that takes advantage of the inputs and outputs of phones and tablets, allowing people to program through conversation on the go.

8.2.6 Abstracting, Modularizing, and Compiling in Codi's Programming Language

To enable children to easily implement complex behaviors with Codi, we need to allow children to modularize projects and compose these modules. We can do this by treating each project as an abstraction of a behavior that can then be embedded as an instruction in a different project. See Section 5.5.3 for an example of what this looks like.

We should also enable children to create projects that take in arguments and give instructions that pass these arguments to the user. For example, a child might want to create a project "Give me jokes" where an argument can be passed to the project, "Give me three jokes." This would expand the creative possibilities for children and help them develop the practice of abstracting and modularizing their code to build systems of projects that are easy to debug.

Codi's natural language processing system involves compiling English instructions into program instructions. Currently, Codi only verifies arguments of high level commands, not arguments to project instructions. In order to make it harder to write programs with mistakes, Codi could detect when arguments to a particular command are invalid while parsing and evaluating semantic rules on the instruction. Then, Codi could let the child know that an argument to an instruction is invalid rather than letting them discover when the project gets played.

8.2.7 Making Codi More Conversational

We could make Codi's natural language interface more robust to the variations in expression enabled by the English grammar and idioms. We could also create a more flexible interface by defining syntactic and semantic rules for the voice user interface. These syntactic and semantic rules would build on English grammar rules, allowing

Codi to understand an instruction's intent by generating parse trees that denote words as noun phrases, verb phrases, direct objects, indirect objects, adjectives, adverbs, and so on. These parse trees would contain more informational parse trees than those parse trees generated by Codi's NLP, and the semantic rules ensure that valid sentences that represent the same concept or idea are understood to be the same. These English language rules would help Codi understand a child's intent even when the child uses pronouns like "it", "that", "them", and "this".

Future work could also make the interface more conversational by handling incomplete instructions. As mentioned in Section 6.5, the system expects instructions to be *complete*. However, a natural conversation with Codi might involve ill-formed instructions that that have filler words or are missing parts. Rather than requiring the child to repeat the entire instruction again, future work could enable Codi to identify what parts might be ambiguous or missing from the instruction. Then, Codi could ask the user for those parts or wait for the user to finish the instruction. On the other hand, if a child rephrases their instruction midway through their first attempt at specifying the instruction, Codi should be able to recognize and discard the initial incomplete instruction.

Appendix A

Tables

A.1 Codi Actions

Name	**Description**	**Ideal Trigger**
stopProject	stop playing a project	stop
editProject	edit the current project	what's inside
stopRecording	end the recording of a sound to use in projects	stop recording
startCues	start the audio cues	start audio cues
renameCurrentProject	rename the current project	call this project *
greet	greet me	hi
getCurrentProject	find out what project you are on	what project am I on
queryActionTypes	explore different kinds of actions	what kinds of things can I do
getProjectNames	hear a list of all the projects	what projects do I have
checkSound	check if there's a boing sound	do you have a boing sound?
queryActions	get a suggestion for what to try next	what can I do
stopBackground	stop the background music	stop the background music
listen	get me to start listening until you say 'hold on'	listen
play	play a project made by Tina called give me a compliment.	give me a compliment
getRecordings	hear me list all the recordings	list recordings
return	go back to the last state you were in	go back
createANewProject	create a new project	new project
startBackground	start the background music	start background music
createANewProjectCalled	create a new project with a name	new project called
playCurrentProject	play the current project	play project
renameRecording	rename recording	rename recording
deleteProject	delete a project	delete the * project
getProjectCount	hear how many projects there are	how many projects do I have
finishProject	leave the project	i'm done
recordASound	start recording a sound to use in projects	record a sound called *
holdOn	make me ignore you until you say 'listen'	hold on
stopCues	stop the audio cues	stop audio cues
editExistingProject	check out what the alarm project is made of	what's inside alarm
getSounds	discover what sounds there are	what sounds are there
goHome	go back to the home state	go home
playARecording	hear the last recording	hear recording
getNthProject	get the first project	what's project number one
queryState	figure out what state you are in	what state am I in
renameProject	change the name of one of your projects	call the * project * instead

A.2 Projects from March Test Sessions

Table A.1: Projects created during the March 5 test session. *P* stands for Participant and *F* stands for Facilitator.

Date	Maker	Project Name	Steps
3/5/19	P1	`What's on my calendar`	Say *what are my events.* Say *computer programming.* Say *dinner.* Say *maker club.*
3/5/19	P2	`The baby is crying`	Say *wah wah wah wah.* Say *it's ok, baby.*
3/5/19	P2	`Random project`	Say *it's morning time.* Play *bird chirping* sounds. Say *I hear birds chirping. It must be a sunny day.* Play the *tropical birds* sound.
3/5/19	P2	`The fun project`	Say *it's a sunny day outside.* Say *it might rain later though.* Say *maybe we can go to the park later today.* Say *it should be a great day. Wahoo.*

Table A.3: Projects created during the March 19 test session. *P* stands for Participant and *F* stands for Facilitator.

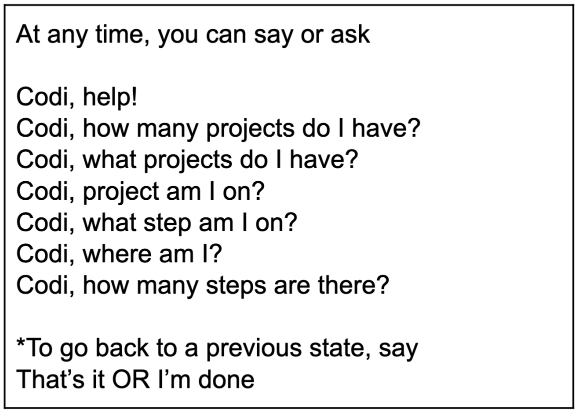
Date	Maker	Project Name	Steps
3/19/19	P1	`Rhino song`	Set instrument to *electric piano.* Play note *78* for *2* beats. Play note *94* for *one* beat. Play note *94* for *one* beat. Play note *87* for *1* beat. Play note *85* for *1* beat.
3/19/19	F1	`Sing with me`	Play the *tina singing* sound.
3/19/19	P2	`How are you doing?`	Say *I'm good thank you I hope you're having a great day*
3/19/19	P2	`Say something nice.`	Say *I think you're very beautiful today*
3/19/19	P2	`Talk to me`	Say *hi.* Whenever you hear *hello*, say *hi there* that's it. Listen and wait. Listen and wait. Whenever you hear *you look pretty today*, say *thank you* that's it. Listen and wait. Whenever you hear *whats for breakfast today*, say *I don't know* that's it. Listen and wait. Whenever you hear *are you excited for breakfast?*, say *I'm only a computer I don't eat breakfast* that's it. Whenever you hear *do you like chocolate*, say *I'm an AI assistant I don't have an opinion on that but thank you for asking* that's it. Whenever you hear *okay I'm headed in for breakfast*, say *I'll see you later* that's it. Listen and wait forever.

Appendix B

Figures

B.1 Learning Resources

B.1.1 Reference Cards from Wizard of Oz testing

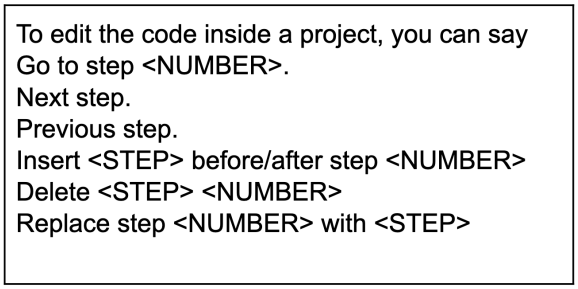


At any time, you can say or ask

Codi, help!
Codi, how many projects do I have?
Codi, what projects do I have?
Codi, project am I on?
Codi, what step am I on?
Codi, where am I?
Codi, how many steps are there?

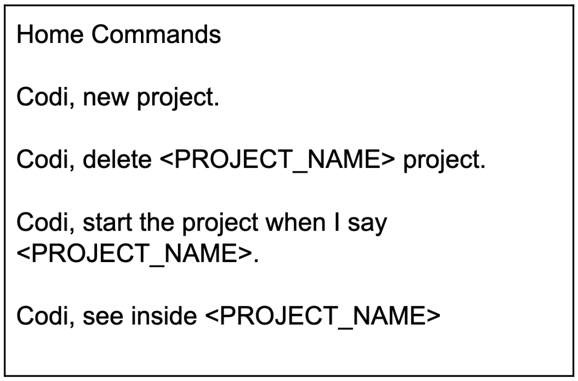
*To go back to a previous state, say
That's it OR I'm done

Figure B-1: Learning resource card from Wizard of Oz testing that lists commands you can say anytime.



To edit the code inside a project, you can say
Go to step <NUMBER>.
Next step.
Previous step.
Insert <STEP> before/after step <NUMBER>
Delete <STEP> <NUMBER>
Replace step <NUMBER> with <STEP>

Figure B-2: Learning resource card from Wizard of Oz testing that lists commands for editing projects.



Home Commands

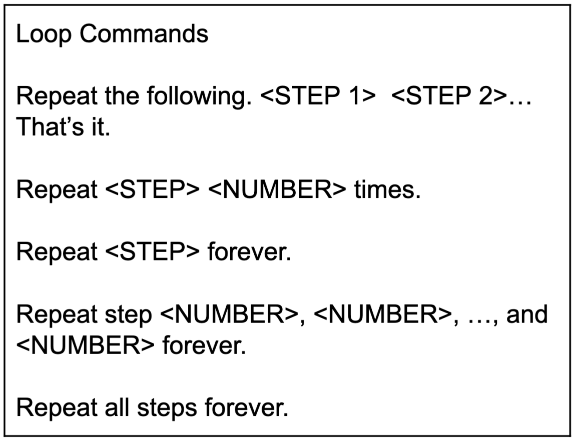
Codi, new project.

Codi, delete <PROJECT_NAME> project.

Codi, start the project when I say
<PROJECT_NAME>.

Codi, see inside <PROJECT_NAME>

Figure B-3: Learning resource card from Wizard of Oz testing that lists commands that work when you're in the "Home" state.



Loop Commands

Repeat the following. <STEP 1> <STEP 2>…
That's it.

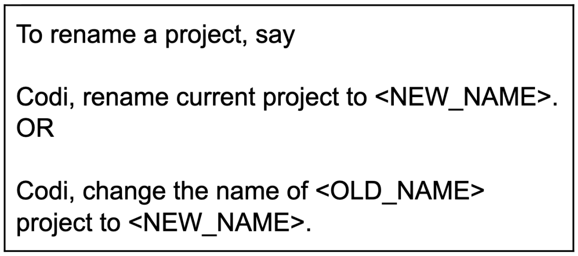
Repeat <STEP> <NUMBER> times.

Repeat <STEP> forever.

Repeat step <NUMBER>, <NUMBER>, …, and
<NUMBER> forever.

Repeat all steps forever.

Figure B-4: Learning resource card from Wizard of Oz testing that lists loop commands.

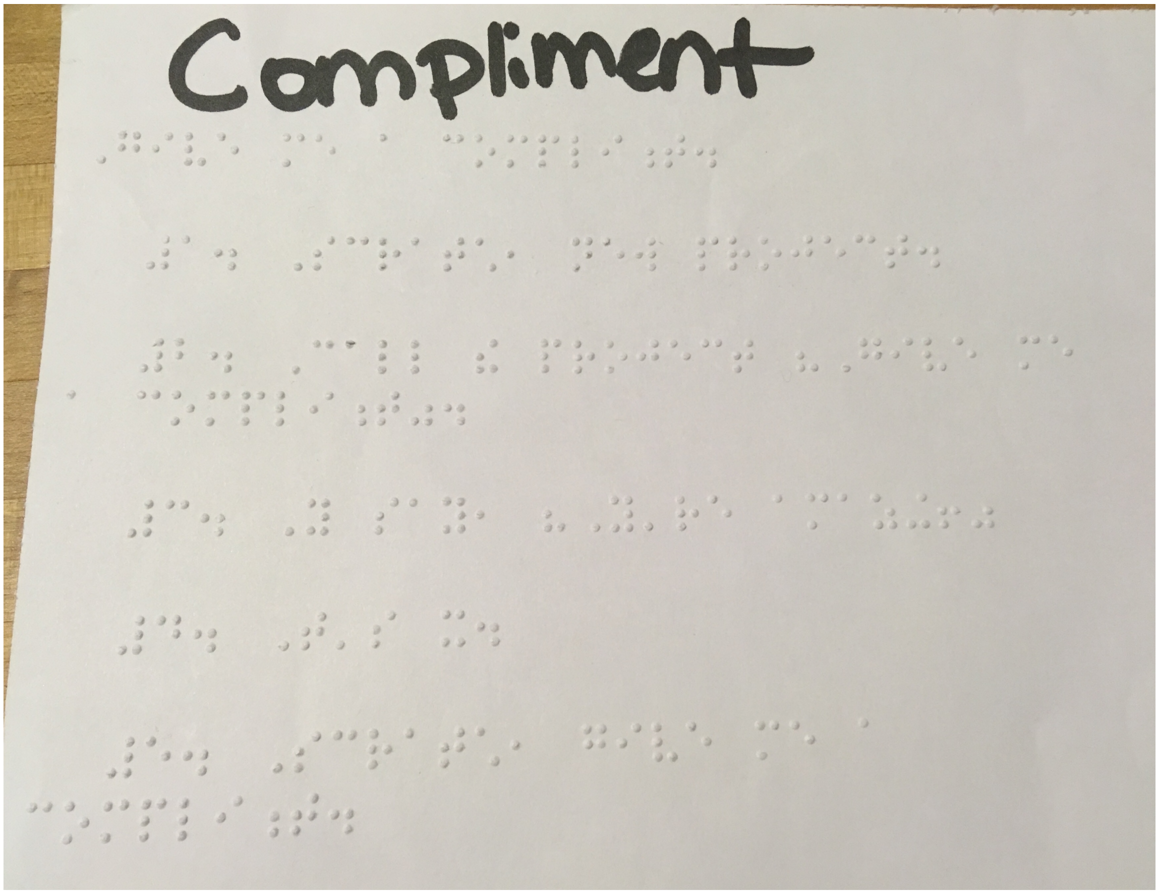


To rename a project, say

Codi, rename current project to <NEW_NAME>.
OR

Codi, change the name of <OLD_NAME>
project to <NEW_NAME>.

Figure B-5: Learning resource card from Wizard of Oz testing with instructions for renaming projects.



Compliment

Figure B-6: Braille learning resource card from Wizard of Oz testing with instructions for making the “Give me a compliment” project.

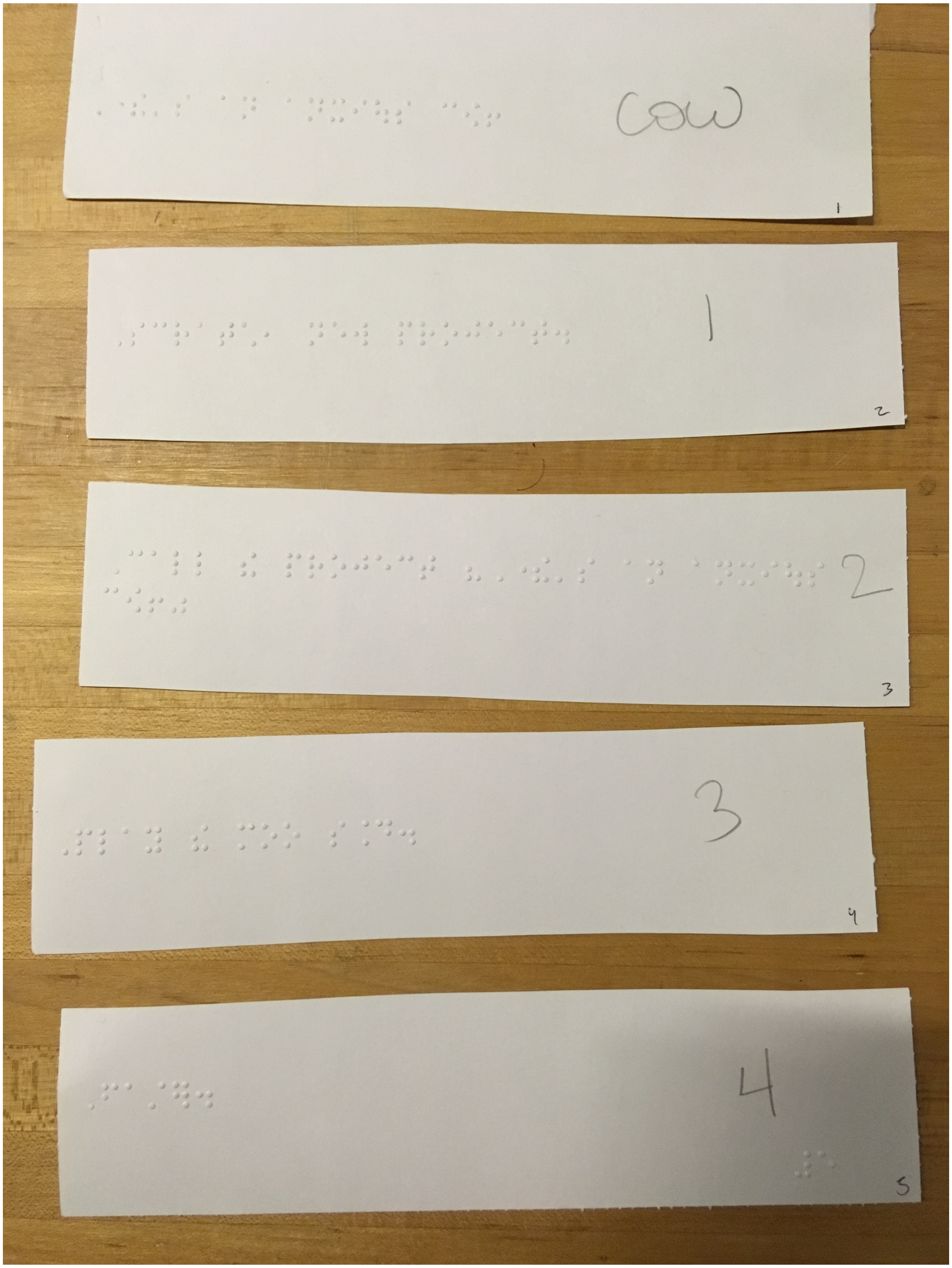


Figure B-7: Steps 1 through 4 of the “There’s an anxious cow” project as braille cards for the Wizard of Oz testing.

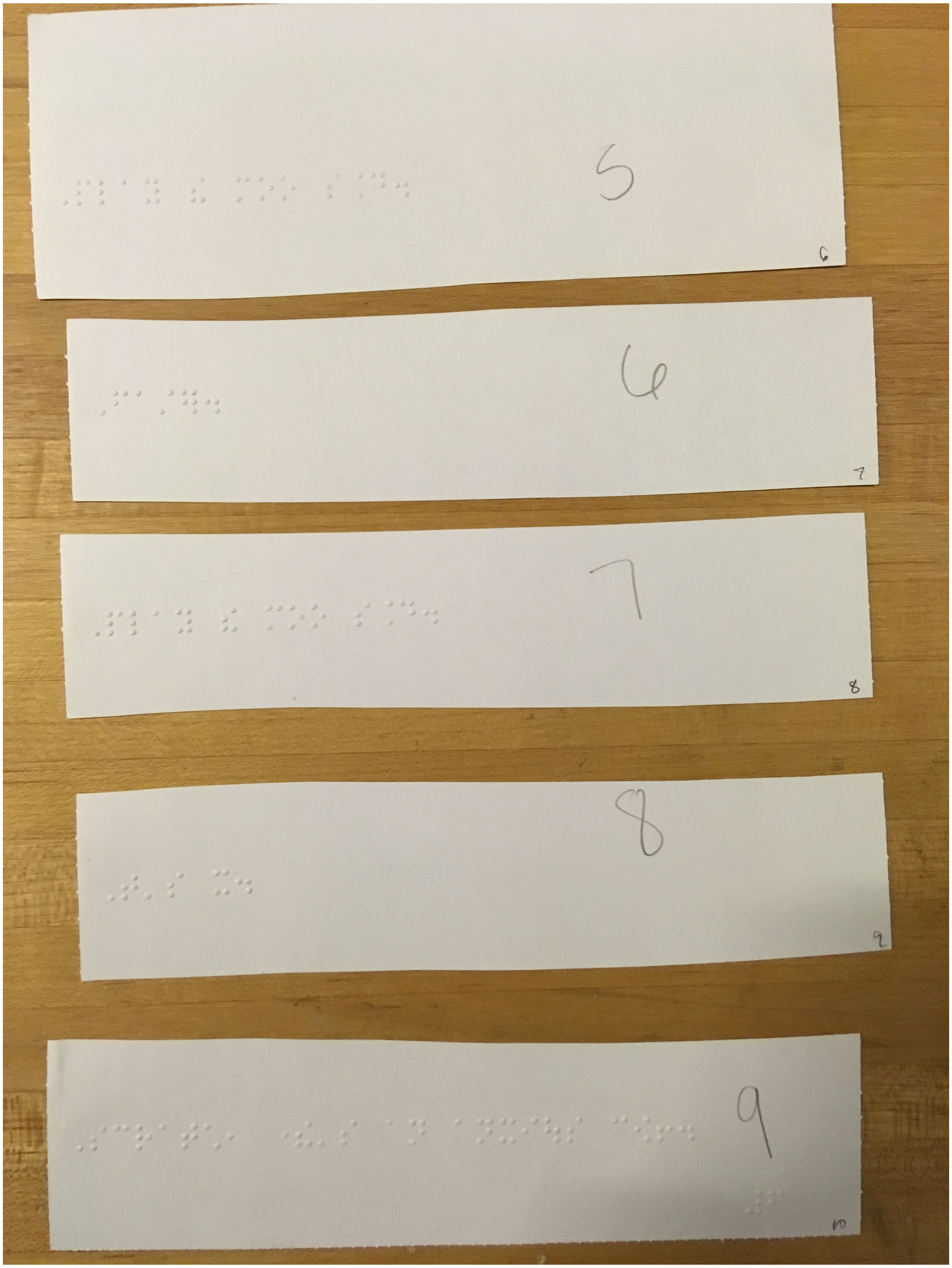


Figure B-8: Steps 5 through 9 of the "There's an anxious cow" project as braille cards for the Wizard of Oz testing.

B.2 Interview Questions

Background Questions

- What do you like to do for fun?
- Have you made or done something recently that you're proud of?
- What is your experience using computers? Have you ever coded before? What do you know about computer science?
- Have you used a voice assistant (Amazon Alexa, Google Home, or Siri) before? What did you use it for? What did you think of that experience?

Onboarding
What projects can you imagine making?

Projects

- How did you feel while working on the project?
- What made you feel that way?
- How did you feel after working on the project?
- What made you feel that way?
- What project was your favorite and why?
- What inspired you to create the projects? Why did you want to make projects?
- If we had more time to work on another project together, what would it be?
- Do you think you'd feel comfortable playing with this on your own?

Computational Thinking

- How does your project work?
- Simulation is a good way to gauge children's understanding without dealing with terms and definitions, specific jargon within coding or computer science. After posing a program or interaction with the system, facilitators can ask, "What will happen next?"

Usability & Tinkerability

- If you had to explain how the system works to a friend how would you do that?
- What do you want to build next? How do you think you would do that?
- Suggestions for improvements?

Bibliography

[1] About Charge: Overview. `www.chargesyndrome.org/about-charge/overview/`.

[2] Accessible Block-Based Programming Environments for the Visually Impaired. `www.se.rit.edu/research/accessible-block-based-programming-environments-visually-impaired`.

[3] Actions on Google. `developers.google.com/actions/`.

[4] Alexa Skills. `www.amazon.com/alexa-skills/b?ie=UTF8&node=13727921011`.

[5] Amazon Alexa — Skill Blueprints. `blueprints.amazon.com/`.

[6] ARIA. `developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA`.

[7] Autism Spectrum Disorder (ASD). `www.cdc.gov/ncbddd/cp/facts.html`.

[8] Blockly — Google Developers.

[9] Common types of additional disabilities for children with visual impairments. `www.familyconnect.org/info/multiple-disabilities/common-types-of-disabilities/13`.

[10] Cubetto: A toy robot teaching kids code & computer programming. `www.primotoys.com/`.

[11] Expanded Core Curriculum (ECC) Hub. `www.perkins.org/school/ecc`.

[12] LLK/scratch-vm. `github.com/LLK/scratch-vm`.

[13] Meet Cozmo. `www.anki.com/en-us/cozmo`.

[14] Meet micro:bit. `microbit.org/guide/`.

[15] Natural Language Toolkit. `www.nltk.org`.

[16] Osmo — play beyond the screen. `www.playosmo.com/en/`.

[17] Perkins School for the Blind. `www.perkins.org/school`.

[18] Screen Reader Training Services. `www.boundlessat.com/Blindness/Screen-Readers/Screen-Reader-Training-Services`.

[19] Slot Filling. `dialogflow.com/docs/concepts/slot-filling`.

[20] Sphero Edu. `edu.sphero.com/`.

[21] What is Cerebral Palsy? `www.cdc.gov/ncbddd/cp/facts.html`.

[22] Mark S Baldwin, Gillian R Hayes, Oliver L Haimson, Jennifer Mankoff, and Scott E Hudson. The tangible desktop: A multimodal approach to nonvisual computing. *ACM Transactions on Accessible Computing (TACCESS)*, 10(3):9, 2017.

[23] Andrew Begel and Susan L Graham. An assessment of a speech-based programming environment. In *Visual LAnguages and Human-Centric Computing (VL/HCC'06)*, pages 116–120. IEEE, 2006.

[24] Rahul Bhargava. *Designing a computational construction kit for the blind and visually impaired*. PhD thesis, Massachusetts Institute of Technology, 2002.

[25] Paulo Blikstein. Pre-college computer science education: A survey of the field. 2018.

[26] Jeffrey Bonar and Elliot Soloway. Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 10–13. ACM, 1983.

[27] Alice Boxhall, James Craig, Dominic. Mazzoni, and Alexander. Surkov. Accessibility Object Model. `github.com/WICG/aom/blob/gh-pages/explainer.md`.

[28] Karen Brennan and Mitchel Resnick. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*, volume 1, page 25, 2012.

[29] Justine Cassell. Embodied conversational agents: representation and intelligence in user interfaces. *AI magazine*, 22(4):67–67, 2001.

[30] Janghee Cho. Mental models and home virtual assistants (hvas). In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, page SRC05. ACM, 2018.

[31] Jan Cuny, Larry Snyder, and Jeannette M Wing. Demystifying computational thinking for non-computer scientists. *Unpublished manuscript in progress, referenced in www. cs. cmu. edu/˜ CompThink/resources/TheLinkWing. pdf*, 2010.

[32] Alain Désilets, David C Fox, and Stuart Norton. Voicecode: An innovative speech interface for programming-by-voice. In *CHI'06 Extended Abstracts on Human Factors in Computing Systems*, pages 239–242. ACM, 2006.

[33] Stefania Druga, Randi Williams, Cynthia Breazeal, and Mitchel Resnick. Hey google is it ok if i eat you?: Initial explorations in child-agent interaction. In *Proceedings of the 2017 Conference on Interaction Design and Children*, pages 595–600. ACM, 2017.

[34] Stefania Druga, Randi Williams, Hae Won Park, and Cynthia Breazeal. How smart are the smart toys?: children and parents' agent interaction and intelligence attribution. In *Proceedings of the 17th ACM Conference on Interaction Design and Children*, pages 231–240. ACM, 2018.

[35] Elizabeth Ellcessor. *Restricted access: Media, disability, and the politics of participation*. NYU Press, 2016.

[36] EndlessReverberation. r/Blind - JAWS Learning curve? `www.reddit.com/r/Blind/comments/8k63bi/jaws_learning_curve/`.

[37] Martin BH Everaert, Marinus AC Huybregts, Noam Chomsky, Robert C Berwick, and Johan J Bolhuis. Structures, not strings: linguistics as part of the cognitive sciences. *Trends in cognitive sciences*, 19(12):729–743, 2015.

[38] Judith Good and Kate Howland. Programming language, natural language? supporting the diverse computational activities of novice programmers. *Journal of Visual Languages & Computing*, 39:78–92, 2017.

[39] Benjamin M Gordon and George F Luger. Progress in spoken programming. In *8th Student Conference*, page 19, 2012.

[40] Lego Group. Mindstorms EV3. `www.lego.com/en-us/mindstorms/products/mindstorms-ev3-31313`.

[41] Adam J Hampton, Amanda N Fisher Boyd, and Susan Sprecher. Youre like me and i like you: Mediators of the similarity–liking link assessed before and after a getting-acquainted social interaction. *Journal of Social and Personal Relationships*, page 0265407518790411, 2018.

[42] Kreg Hanning. *Tinkering with ScratchBit: explorations in blended making.* PhD thesis, Massachusetts Institute of Technology, 2018.

[43] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, page 26, 2003.

[44] Yasmin B Kafai and Mitchel Resnick. *Constructionism in practice: Designing, thinking, and learning in a digital world.* Routledge, 2012.

[45] Shaun K Kane, Varsha Koushik, and Annika Muehlbradt. Bonk: accessible programming for accessible audio games. In *Proceedings of the 17th ACM Conference on Interaction Design and Children*, pages 132–142. ACM, 2018.

[46] Tom Kelley and David Kelley. *Creative confidence: Unleashing the creative potential within us all.* Currency, 2013.

[47] Jonathan Lazar, Aaron Allen, Jason Kleinman, and Chris Malarkey. What frustrates screen reader users on the web: A study of 100 blind users. *International Journal of human-computer interaction*, 22(3):247–269, 2007.

[48] Tuck Wah Leong, Frank Vetere, and Steve Howard. Randomness as a resource for design. In *Proceedings of the 6th conference on Designing Interactive systems*, pages 132–139. ACM, 2006.

[49] Lightningstep36. Make Scratch more accessible to blind and visually impaired users. `scratch.mit.edu/discuss/post/1478330/`.

[50] Stephanie Ludi. Position paper: Towards making block-based programming accessible for blind users. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 67–69. IEEE, 2015.

[51] Ewa Luger and Abigail Sellen. Like having a really bad pa: the gulf between user expectation and experience of conversational agents. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5286–5297. ACM, 2016.

[52] Patricia Yancey Martin and Barry A Turner. Grounded theory and organizational research. *The journal of applied behavioral science*, 22(2):141–157, 1986.

[53] meeow_me. r/Blind - JAWS Learning curve? `www.reddit.com/r/Blind/comments/8k63bi/jaws_learning_curve/`.

[54] Lauren R Milne and Richard E Ladner. Blocks4all: Overcoming accessibility barriers to blocks programming for children with visual impairments. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 69. ACM, 2018.

[55] Clifford Nass and Corina Yen. *The man who lied to his laptop: What we can learn about ourselves from our machines.* Penguin, 2010.

[56] Michael Nichols, Gopal Gupta, and Qian Wang. Voice-commanded scripting language for programming navigation strategies on-the-fly. In *Proceedings of the HCI International 2005*, 2005.

[57] Kurt Nørmark. Systematic unit testing in a read-eval-print loop. *J. UCS*, 16(2):296–314, 2010.

[58] Anna Nowogrodzki. Speaking in code: how to program by voice. *Nature*, 559(7712):141, 2018.

[59] Seymour Papert. Hard fun. `www.papert.org/articles/HardFun.html`.

[60] Hae Won Park, Rinat Rosenberg-Kima, Maor Rosenberg, Goren Gordon, and Cynthia Breazeal. Growing growth mindset with a social robot peer. In *2017 12th ACM/IEEE International Conference on Human-Robot Interaction (HRI*, pages 137–145. IEEE, 2017.

[61] Roy D Pea. Language-independent conceptual bugs in novice programming. *Journal of educational computing research*, 2(1):25–36, 1986.

[62] Rosalind W Picard. *Affective computing*. MIT press, 2000.

[63] Joseph Pistrui. The future of human work is imagination, creativity, and strategy. `hbr.org/2018/01/the-future-of-human-work-is-imagination-creativity-and-strategy`, Jan 2018.

[64] Anna Powers. Creativity is the skill of the future. `www.forbes.com/sites/annapowers/2018/04/30/creativity-is-the-skill-of-the-future/#2c655944fd48`, May 2018.

[65] Mitchel Resnick. All i really need to know (about creative thinking) i learned (by studying how children learn) in kindergarten. In *Proceedings of the 6th ACM SIGCHI conference on Creativity & cognition*, pages 1–6. ACM, 2007.

[66] Mitchel Resnick. Learn to code, code to learn. *EdSurge, May*, 54, 2013.

[67] Mitchel Resnick. Give ps a chance: Projects, peers, passion, play. In *Constructionism and creativity: Proceedings of the Third International Constructionism Conference. Austrian Computer Society, Vienna*, pages 13–20, 2014.

[68] Mitchel Resnick. The seeds that seymour sowed. *International Journal of Child-Computer*, 2017.

[69] Mitchel Resnick and Ken Robinson. *Lifelong kindergarten: Cultivating creativity through projects, passion, peers, and play.* MIT Press, 2017.

[70] Mitchel Resnick and Eric Rosenbaum. Designing for tinkerability. *Design, make, play: Growing the next generation of STEM innovators*, pages 163–181, 2013.

[71] E. Rosenbaum. Category color confusion. `github.com/LLK/scratch-blocks/issues/995`.

[72] sarhat. Is there ADA Accessibility to Scratch? I have a low vision student who is struggling to use Scratch. `scratch.mit.edu/discuss/topic/114554/?page=1#post-1008792`.

[73] Hannah Schwar and Rqayyah Moynhihan. There's a clever psychological reason why amazon gave alexa a female voice. 2018.

[74] Hannah Schwr and Ruqayyah Moynihan. There's a psychological reason why amazon gave alexa a female voice. `www.businessinsider.com/theres-psychological-reason-why-amazon-gave-alexa-a-female-voice-2018-9`.

[75] Alex Sciuto, Arnita Saini, Jodi Forlizzi, and Jason I Hong. Hey alexa, what's up?: A mixed-methods studies of in-home conversational agent usage. In *Proceedings of the 2018 on Designing Interactive Systems Conference 2018*, pages 857–868. ACM, 2018.

[76] Scratch. About. `scratch.mit.edu/about/`.

[77] ScratchTeacher404. Colour Blind Option. `scratch.mit.edu/discuss/topic/109778/`.

[78] Martin Seligman. Flourish: Positive psychology and positive interventions. *The Tanner lectures on human values*, 31, 2010.

[79] Jay Jay Saul Silver. *Lens x block: World as construction kit.* PhD thesis, Massachusetts Institute of Technology, 2014.

[80] Evropi Stefanidi, Maria Korozi, Asterios Leonidis, and Margherita Antona. Programming intelligent environments in natural language: An extensible interactive approach. In *Proceedings of the 11th PErvasive Technologies Related to Assistive Environments Conference*, pages 50–57. ACM, 2018.

[81] Meenakshi Swaminathan. Cortical visual impairment in childrena new challenge for the future? *Oman journal of ophthalmology*, 4(1):1, 2011.

[82] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285, 1988.

[83] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[84] Next Day Video. Using python to code by voice. `www.youtube.com/watch?v=8SkdfdXWYaI`, Mar 2013.

[85] Amber Wagner and Jeff Gray. An empirical evaluation of a vocal user interface for programming by voice. *International Journal of Information Technologies and Systems Approach (IJITSA)*, 8(2):47–63, 2015.

[86] Warthil. r/Blind - JAWS Learning curve? `www.reddit.com/r/Blind/comments/8k63bi/jaws_learning_curve/`.

[87] Randi Williams, Christian Vázquez Machado, Stefania Druga, Cynthia Breazeal, and Pattie Maes. My doll says it's ok: a study of children's conformity to a talking doll. In *Proceedings of the 17th ACM Conference on Interaction Design and Children*, pages 625–631. ACM, 2018.

[88] William E Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. 1990.