Shiro - A Language to Represent Alternatives

by

Jeffrey Robert Guenther

B.Sc., Liberty University, 2010

Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in the School of Interactive Arts and Technology Faculty of Communication, Art, and Technology

© Jeffrey Robert Guenther 2017
SIMON FRASER UNIVERSITY
Spring 2017

All rights reserved. However, in accordance with the Copyright Act of Canada, this work may be reproduced without authorization under the conditions for “Fair Dealing.” Therefore, limited reproduction of this work for the purposes of private study, research, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
Approval

Name: Jeffrey Robert Guenther
Degree: Doctor of Philosophy
Title: Shiro - A Language to Represent Alternatives
Examining Committee: Chair: Dr. Diane Gromala
                     Professor

Dr. Chris D. Shaw
Senior Supervisor
Professor

Dr. Robert Woodbury
Supervisor
University Professor

Dr. Brian Fisher
Internal Examiner
Professor

Dr. Ramesh Krishnamurti
External Examiner
Professor
School of Architecture
Carnegie Mellon University

Date Defended: 21 December 2016
Abstract

When people solve problems they explore a variety of potential solutions. Parametric systems have been added to the interaction models of design tools to make it easier to change a design. Just as a cell is changed in a spreadsheet, when a parameter is changed all parts of the design that depend on that parameter update. While tools with parametric systems are powerful, users are limited to single state solutions and are forced to use workarounds like using layers and file naming conventions. These improvisations are caused by tools whose user interface and document models only support a single state. Single-state document models are only designed to represent a single artifact.

In this work, I describe Shiro, a declarative, dataflow language for expressing alternatives in parametric systems. It provides a multi-state document model for parametric systems. To make this possible, I introduce the concept of subjunctive nodes, nodes that contain options. Options allow users to vary property values and computations. I demonstrate Shiro with a variety of examples from the designs of design and data analysis. Finally, I discuss what I learned while designing and implementing the language and provide a set of recommendations for future research.

Keywords: alternatives; parametric systems; multi-state document model
Dedication

To my dearest Julie, I’m done now. We can go outside and play!
The ability to complete a PhD is a tremendous privilege and requires the support of many people. Completing this work has been a test of will and endurance. Thank you to my wife Julie who has endured many evenings without me as I’ve completed this work. We made it! It’s time to play!! Thank you to my brother Daniel who let me take over his home office so I’d have a place to complete this manuscript and to Mom and Dad who kept me fed and watered for the month as I wrote. Thank you Chris for selecting me as your grad student and providing me with my research assistantship. I’ll always remember our late lunch talks in the cafeteria and the many hours we spent chatting at the Starbucks in the mall. Thank you Rob for introducing me to the world of computational design. You introduced me to a topic that has inspired me, challenged me, and provided a home for my wild ideas. I thoroughly enjoyed sparring with you. Thank you to Dr. Alissa Antle for introducing me to embodied cognition, a field which will forever influence the way I design user interfaces. Thank you to my CZSaw lab-mates Nazanin, Ankit, Saba, Eric, and Dustin for your camaraderie, friendship, and many great chats in the CZCave. What a ride this process has been!

—Soli Deo gloria
# Table of Contents

Approval ii  
Abstract iii  
Dedication iv  
Acknowledgements v  
Table of Contents vi  
List of Tables ix  
List of Figures x  
List of Listings xii

1 Introduction 1  

2 Related Work 6  
2.1 Human Problem Solving ........................................ 6  
2.2 Single-state Document Model .................................. 8  
2.3 Interfaces for Alternatives .................................... 13  
2.4 Subjunctive Interfaces in Parametric Design Tools ............. 17  
2.5 Alternatives in Text Authoring Tools .......................... 20  
2.6 Support for Alternatives in Commercially Available Tools ........ 21  
2.6.1 Version Control Systems ................................... 22  
2.6.2 Microsoft Excel .............................................. 23  
2.6.3 Photoshop ..................................................... 23  
2.6.4 VisTrails ..................................................... 24  
2.6.5 Grasshopper ................................................... 25  
2.7 Languages .......................................................... 26  
2.8 Towards a Multi-state Document Model ......................... 29

3 Shiro 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Shiro at a Glance</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Language Definition</td>
<td>42</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Node Definitions</td>
<td>43</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Using Nodes</td>
<td>45</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Options and States</td>
<td>51</td>
</tr>
<tr>
<td>3.3</td>
<td>Evaluation</td>
<td>53</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Dependency Resolution</td>
<td>53</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Collect Definitions</td>
<td>53</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Realize States</td>
<td>54</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Recursion</td>
<td>54</td>
</tr>
<tr>
<td>3.4</td>
<td>Idioms for Describing Alternatives</td>
<td>56</td>
</tr>
<tr>
<td>3.5</td>
<td>Shiro Playground</td>
<td>57</td>
</tr>
<tr>
<td>3.6</td>
<td>Implications for History and Undo</td>
<td>59</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Represent Time as States</td>
<td>60</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Tradeoffs to Representing Time as States</td>
<td>66</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Representing Time as Transactions</td>
<td>68</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Compare Representing Time as States and Transactions</td>
<td>70</td>
</tr>
<tr>
<td>3.7</td>
<td>A Multi-state Data Model</td>
<td>72</td>
</tr>
<tr>
<td>4</td>
<td>Evaluating Shiro</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Designing a Logo</td>
<td>74</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Problem Statement</td>
<td>74</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Solution</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>Developing Color Schemes</td>
<td>82</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Problem Statement</td>
<td>82</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Solution</td>
<td>83</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Completed Code</td>
<td>86</td>
</tr>
<tr>
<td>4.3</td>
<td>A Basic Interest Calculation</td>
<td>90</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Problem Statement</td>
<td>90</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Solution</td>
<td>90</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Completed Code</td>
<td>93</td>
</tr>
<tr>
<td>4.4</td>
<td>Costume Selection</td>
<td>95</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Problem Statement</td>
<td>95</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Solution</td>
<td>95</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Completed Code</td>
<td>97</td>
</tr>
<tr>
<td>4.5</td>
<td>Analysing Poverty Data</td>
<td>97</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Problem Statement</td>
<td>97</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Solution</td>
<td>98</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Completed Code</td>
<td>100</td>
</tr>
</tbody>
</table>
4.6 Evaluation ......................................................... 101
  4.6.1 Ecological Validity ........................................... 103
  4.6.2 External Validity ............................................ 104
  4.6.3 Internal Validity ............................................. 105
  4.6.4 Replication and Alternatives .............................. 105

5 Future Work ...................................................... 106
  5.1 Integrated Text, Graphical Programming, and Direct Manipulation Interfaces 106
  5.2 Event Driven Execution ....................................... 107
  5.3 Improving Performance ....................................... 108
    5.3.1 Parallel Execution and Caching .......................... 108
  5.4 Type System .................................................... 109
  5.5 Treat Solution Space as Database ........................... 112
  5.6 Generating Alternatives ..................................... 113
  5.7 Reconcile History and Alternatives ........................ 114
  5.8 Build a Tool .................................................. 114

6 Conclusions ....................................................... 116

Bibliography ........................................................ 117

Appendix A Shiro ANTLR Grammar ................................ 122
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Return of Product A on $5000 without additional contributions</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Return of Product A with Leslie’s $100/month contribution</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Return of Product A with Leslie’s $120/month contribution</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Return of Product B on $5000</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>Result of Product B with Leslie’s $120/month contribution</td>
<td>3</td>
</tr>
<tr>
<td>4.1</td>
<td>Sample of the CSV data</td>
<td>98</td>
</tr>
</tbody>
</table>
# List of Figures

| Figure 2.1 | Using layers to represent alternatives in Pixelmator | 10 |
| Figure 2.2 | Alternative business cards designs on a single canvas. Design by Mig Reyes at Basecamp. Used with permission. | 11 |
| Figure 2.3 | A Numbers spreadsheet where cells A1 and B1 are added in C1 | 12 |
| Figure 2.4 | The dependency graph created by the constraint system. Arrows point in the direction of the dependency | 13 |
| Figure 2.5 | NodeBox user interface | 19 |
| Figure 2.6 | GEM-NI showing three alternatives | 20 |
| Figure 2.7 | Alt.Text displaying a document | 22 |
| Figure 2.8 | Excel’s Scenario Manager | 23 |
| Figure 2.9 | Photoshop Layer Comp controlling visibility | 24 |
| Figure 2.10 | The VisTrails interface | 25 |
| Figure 2.11 | Explore parameter values | 26 |
| Figure 2.12 | Custom replication strategy dialog | 27 |
| Figure 2.13 | Manipulating states in Grasshopper | 27 |
| Figure 3.1 | Dependency graph for code in Listing 3.19 | 43 |
| Figure 3.2 | Path resolution of b.y in Listing 3.32 starts in the default graph named ^_. The first segment of the path matches the node instance a (1) in the scope of ^_. Search continues for the rest of the path in scope of a. x is the next path segment and matches the port x in a (2). Since x is the last path segment, a reference to the port is returned, completing the search. | 50 |
| Figure 3.3 | Fibonacci recursively evaluates a detached instance of the node. | 56 |
| Figure 3.4 | A single alternative can be shown in the Canvas by selecting a single state. (a), (b), (c) and (d) each show one such state. | 58 |
| Figure 3.5 | Selecting multiple states in the Canvas superimposes their views into a single view. This figure shows states (a), (b) and (c) from Figure 3.4. | 58 |
| Figure 3.6 | The LightTable defaults to arranging states in a grid. | 59 |
| Figure 3.7 | Users can manually configure the LightTable to any layout of states. | 59 |
| Figure 3.8 | History | 60 |
| Figure 3.9 | QDraw canvas | 68 |
Figure 3.10  **QDraw** code editor  

Figure 4.1  Logo options on white background  

Figure 4.2  Draw the background ellipse with a midline.  

Figure 4.3  Rough in text box  

Figure 4.4  Draw lines above and below the logotype.  

Figure 4.5  Draw lines above and below the logotype.  

Figure 4.6  Draw lines above and below the logotype.  

Figure 4.7  Draw a palette in color and monochrome.  

Figure 4.8  Draw the base character  

Figure 4.9  Draw the character with the two costume options  

Figure 4.10 Render the list of countries in 2010  

Figure 4.11 Render the list of countries in 2010 with below average poverty
List of Listings

3.1 Example Listing .................................................. 33
3.2 Literal integer 3 .................................................. 33
3.3 Literals .......................................................... 33
3.4 Named Port ....................................................... 34
3.5 Using a named port in an expression .......................... 34
3.6 Explicit dataflow form of Listing 3.5 ........................ 34
3.7 Declaration of dataflow using infix expressions ............ 34
3.8 Explicit declaration of dataflow ............................... 34
3.9 Node declaration ................................................ 35
3.10 Node defined with input, output, and private ports ....... 35
3.11 Node defined with input and output ports .................. 35
3.12 Declaring an instance of Box ................................. 36
3.13 Instantiating a node with arguments ......................... 36
3.14 Calculating the cost of Boxes of different volumes ...... 37
3.15 Box and Cost definitions ...................................... 38
3.16 Define layout node ............................................. 38
3.17 Add Pool definition ............................................ 40
3.18 Define graph and states ....................................... 40
3.19 Explicitly defined expression ................................. 43
3.20 A Node or function definition ................................. 43
3.21 Paths created by an instance of Listing 3.20 ............... 44
3.22 Using an instance of AreaSquare in a graph ............... 45
3.23 Define a graph .................................................. 45
3.24 Nested node definition ........................................ 46
3.25 Create instances of nested nodes in the default graph ... 46
3.26 Create an instance of a nested node inside of its containing node ................................. 47
3.27 Syntax error in a nested node definition ....................... 47
3.28 Correctly connected ports inside a nested node definition ...... 47
3.29 Define a node to calculate volume with a magic factor .... 48
3.30 Read the first value of length in Listing 3.29 ............. 48
3.31 Set the port length in Listing 3.29 ........................................ 49
3.32 Code used to illustrate path resolution .................................. 49
3.33 Passing a path selector to a function ...................................... 50
3.34 Passing a reference path to a function .................................... 51
3.35 Declare two ports as optional .............................................. 51
3.36 Define three options .......................................................... 52
3.37 The state definitions generated by the runtime for Listing 3.36 ..... 53
3.38 Naïve Calculation of a Fibonacci number ................................. 54
3.39 Use tail-recursion to calculate a Fibonacci number .................. 55
3.40 Property Variation ............................................................. 56
3.41 Object Variation ............................................................... 57
3.42 A subjunctive node representing values at \( t_0 \) and \( t_1 \) ........ 60
3.43 Add states to Listing 3.42 to represent \( t_0 \) and \( t_1 \) ............... 60
3.44 Use a naming convention to identify the states to evaluate for \( t_n \) ..... 61
3.45 Represent changes to a graph definition .................................. 62
3.46 Recall, the definition of \( C \) ................................................... 64
3.47 Redeclarer \( C \) with options for small .................................... 64
3.48 Update states to handle nest option small ............................... 64
3.49 Current definition of artifact ................................................ 67
3.50 Transactions Statements ..................................................... 67
3.51 QDraw statements ................................................................ 69
3.52 QDraw transaction with the resulting state cache .................... 70
3.53 State with multiple time steps references ................................. 70
4.1 Draw an ellipse as the background ....................................... 74
4.2 Add the text ................................................................. 75
4.3 Add decorative lines ........................................................... 76
4.4 Refactor drawing into named graph ......................................... 77
4.5 Add optional typeface values ............................................... 79
4.6 Define remaining optional nodes ......................................... 81
4.7 Node definition for the first value of the scale. ......................... 83
4.8 Node to convert colors to grayscale ....................................... 84
4.9 Palette node. ................................................................. 84
4.10 Complete code for color scheme .......................................... 86
4.11 Nodes to calculate interest. ............................................... 91
4.12 Represent the investment ................................................... 91
4.13 Investment values .......................................................... 91
4.14 States configuring the investment products ............................. 92
4.15 Completed Investment ..................................................... 93
4.16 Render the base character .................................................. 95
Chapter 1

Introduction

Exploration of alternatives plays an important role in how people solve problems. Problems are often ill-defined [45, 46] and require people to *satisfice*, or to seek a good enough solution rather than the “perfect” solution. To choose a solution, people make trade-offs between the properties of different solutions.

Consider the following scenarios: Markus receives a brief from the creative director at his design firm detailing a request from EverClean, a new natural cleaning products company, to design a logo. The owners of EverClean want their logo to represent the company’s commitment to providing customers with non-toxic alternatives to chemical based cleaners. The brief reads as follows:

*EverClean is a new entry into the natural cleaning products market. We want to establish the EverClean brand as a brand people look to when they need to clean the dirtiest of things and want to do it without using harmful chemicals. We value people and the environment. We believe you can clean using a completely natural product. We want a logo that presents our values of environmental sustainability, care for people, and a product that works. EverClear, our flagship product, is an all natural, all-purpose cleaner perfect for use around the home.*

Markus opens Photoshop and begins to brainstorm. He starts by examining his collection of typefaces to see if any might be suitable for the logotype. He creates a text box and types in the company’s name – EverClean. Starting with some familiar typefaces, he begins to try different options. Each time Markus finds a typeface he likes, he copies and pastes the text box to save a record of the typeface. After a while, he has a screen filled with different typefaces. He organizes the text boxes according to how well he thinks the typefaces communicate the brand. Modern, powerful typefaces are grouped in one part of the screen. Vibrant, natural typefaces are grouped in another. Several candidates emerge from his sort.
Then Markus thinks to himself, “I wonder if the client will let me change up the capitalization and spacing of the text. Even if they don’t, I want to try a couple things…” To try his new ideas, Markus selects all of the text boxes and pastes them on a new canvas. To see the new text in each of the typefaces, he selects and edits each text box and makes the change. Each time Markus wants to try different text for the typefaces, he selects each of the text boxes and edits the text. After trying several different combinations of font and color, Markus decides on four options to show the owners of EverClean. He will see which designs they like best and continue from there.

Designers are not the only people who explore alternatives. People in many domains explore alternatives in their work. Consider Leslie’s desire to understand how much she will earn if she chooses one of the investment products her financial planner presented to her. She has $5000 to invest. Her financial planner presented her with the following options:

- Product A earns 12% interest compounded annually for a three year term, with the ability to contribute annually
- Product B earns 5% compounded annually for a five year term, with the ability to contribute annually

She knows she can save $100 a month to contribute to the investment, given her current income. She wonders to herself how much money she will earn if she lets her $5000 grow for the term of Product A. To help her do the math, she creates a spreadsheet.

<table>
<thead>
<tr>
<th>Year</th>
<th>Principal</th>
<th>Interest Rate</th>
<th>Earnings</th>
<th>Investment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$5,000.00</td>
<td>12%</td>
<td>$600.00</td>
<td>$5,600.00</td>
</tr>
<tr>
<td>2</td>
<td>$5,600.00</td>
<td>12%</td>
<td>$672.00</td>
<td>$6,272.00</td>
</tr>
<tr>
<td>3</td>
<td>$6,272.00</td>
<td>12%</td>
<td>$752.64</td>
<td>$7,024.64</td>
</tr>
</tbody>
</table>

Table 1.1: Return of Product A on $5000 without additional contributions

She discovers her investment will be worth $7,024.64 after three years if she makes no annual contributions. If she saves $100 per month to contribute to the investment, for a total contribution of $1,200 per year, her investment will grow to $11,073.92 in three years (See Table 1.2).

<table>
<thead>
<tr>
<th>Year</th>
<th>Principal</th>
<th>Interest Rate</th>
<th>Earnings</th>
<th>Investment</th>
<th>Contribution</th>
<th>Yearly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$5,000.00</td>
<td>12%</td>
<td>$600.00</td>
<td>$5,600.00</td>
<td>$100.00</td>
<td>$1,200.00</td>
</tr>
<tr>
<td>2</td>
<td>$6,800.00</td>
<td>12%</td>
<td>$816.00</td>
<td>$7,616.00</td>
<td>$100.00</td>
<td>$1,200.00</td>
</tr>
<tr>
<td>3</td>
<td>$8,816.00</td>
<td>12%</td>
<td>$1,057.92</td>
<td>$9,873.92</td>
<td>$100.00</td>
<td>$1,200.00</td>
</tr>
</tbody>
</table>

Table 1.2: Return of Product A with Leslie’s $100/month contribution
She wonders what her return will be if she contributes $120 per month instead. To compare the differences, she copies the spreadsheet and changes her monthly contribution to $120 (See Table 1.3).

<table>
<thead>
<tr>
<th>Year</th>
<th>Principal</th>
<th>Interest Rate</th>
<th>Earnings</th>
<th>Investment</th>
<th>Contribution</th>
<th>Yearly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$5,000.00</td>
<td>12%</td>
<td>$600.00</td>
<td>$5,600.00</td>
<td>$120.00</td>
<td>$7,040.00</td>
</tr>
<tr>
<td>2</td>
<td>$7,040.00</td>
<td>12%</td>
<td>$844.80</td>
<td>$7,884.80</td>
<td>$120.00</td>
<td>$9,324.80</td>
</tr>
<tr>
<td>3</td>
<td>$9,324.80</td>
<td>12%</td>
<td>$1118.98</td>
<td>$10,443.78</td>
<td>$120.00</td>
<td>$11,883.78</td>
</tr>
</tbody>
</table>

Table 1.3: Return of Product A with Leslie’s $120/month contribution

By contributing $120 per month the value of her investment will be $11,883.78, which is $809.86 more than if she contributes $100 per month. Satisfied for the moment with what she has learned, she begins to investigate Product B. She creates a set of similar spreadsheets to the ones she used to evaluate Product A (See Table 1.4).

<table>
<thead>
<tr>
<th>Year</th>
<th>Principal</th>
<th>Interest Rate</th>
<th>Earnings</th>
<th>Contribution</th>
<th>Yearly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$5,000.00</td>
<td>5%</td>
<td>$250.00</td>
<td>$120.00</td>
<td>$6,690.00</td>
</tr>
<tr>
<td>2</td>
<td>$5,250.00</td>
<td>5%</td>
<td>$262.50</td>
<td>$120.00</td>
<td>$8,464.50</td>
</tr>
<tr>
<td>3</td>
<td>$5,512.50</td>
<td>5%</td>
<td>$275.63</td>
<td>$120.00</td>
<td>$10,327.73</td>
</tr>
<tr>
<td>4</td>
<td>$5,788.13</td>
<td>5%</td>
<td>$289.41</td>
<td>$120.00</td>
<td>$12,284.11</td>
</tr>
<tr>
<td>5</td>
<td>$6,077.53</td>
<td>5%</td>
<td>$303.88</td>
<td>$120.00</td>
<td>$14,338.32</td>
</tr>
</tbody>
</table>

Table 1.4: Return of Product B on $5000

If she contributes $100 per month, her investment will be worth $9,571.13 after three years and $13,012.17 by the end of the term. Contributing $120 per month provides an even better return of $14,338.32 at the end of five years.

Table 1.5: Result of Product B with Leslie’s $120/month contribution

Provided Leslie is able to keep her money invested for the five year term and is able to make a yearly contribution, Product B is a better investment. If she is not able to make a yearly contribution, Product A is the better investment, because, using the numbers from Table 1.1 and Table 1.4, Product A earns $7,024.64 − $6,381.41 = $643.23 more interest.

Markus and Leslie complete their tasks by exploring alternatives. They use existing software tools to help them, but neither tool is designed to make it easy. Our scenario with
Markus illustrates how designers explore different options as they work. Exploring different typefaces, colors, and layouts is fundamental to design. It also illustrates a fundamental weakness of design tools like Photoshop. (Other design tools like Illustrator\(^1\), Pixelmator\(^2\), and Inkscape\(^3\) share the same weakness as they employ a similar interaction model.) These tools provide inadequate support for exploring alternatives. Being required to manually select and change each text box each time Markus wants to change the text of the logo for each possible typeface is tedious and time consuming.

To make matters worse, the problem quickly becomes even more complex and more time consuming if Markus wants to vary more than the text of the logo. With the addition of each varied parameter, the amount of labor required to render the different designs increases. For example, if he wishes to explore 2 text options, 5 typefaces, 3 colors, and 2 different weights, he will have to make 60 \((2 \times 5 \times 3 \times 2 = 60)\) designs. The number of clicks to complete this work will likely be many times more. As a result, much effort is required to create and to review variations of a design.

Leslie faces a similar challenge as she evaluates the two investment products. To learn how much more she would earn if she contributed $120 per month instead of $100, she created a copy of her calculations. Because spreadsheets automatically update when a cell is changed, she could change the value of her contribution and the spreadsheet would be recalculated; however, doing so would erase her previous work. To preserve the two results, she must create a copy of the spreadsheet and then change the amount of her contribution. If Leslie wants to change the amount of her principal, she needs to change it on both spreadsheets. Admittedly, she could place the investment amount in a separate cell and use it to drive the calculations, causing any change to the principal to be propagated to the two other spreadsheets. While adding a cell to represent the principal minimizes some of the work required to test different principals, unfortunately, she faces the same problem she did when comparing the returns based on the amount of her monthly contribution. Whenever she changes the value of the principal, she loses the previous result. To compare the results, she needs to save the them by creating more copies of the spreadsheets. Leslie’s use of a spreadsheet allowed her to create calculations based on other cells, but fails to alleviate the labour necessary to compare multiple monthly contributions.

Fortunately for Leslie’s work, she was able to create calculations based on the value of other cells. Markus was not so fortunate. Design tools, like Photoshop, do not allow designers to create relationships between parts of their design. For example, if Markus decides to use a circle as the background of the logo, each time the length of the text box changes he would need to adjust the circle’s radius so it remains properly positioned. A large amount of work is needed if he wants to update all of his alternatives. It benefits

\(^{1}\)http://www.adobe.com/ca/products/illustrator.html
\(^{2}\)http://www.pixilemator.com/
\(^{3}\)https://inkscape.org/en/
designers if they are able to relate objects in their design to each other, so that changes to one part of the design affect the rest of the design. Furthermore, to make it easier for people to explore alternatives while solving problems using software tools, a multi-state document model is needed.

These scenarios demonstrate some of the difficulties people face when they use existing software tools to explore alternatives. In this work, I summarize problem solving theory’s account (Chapter 2) of how people explore alternatives to solve problems. I describe how tools employing a single state document model make it difficult for designers to represent alternatives and discuss some of the workarounds people use to deal with this difficulty. I present previous attempts to create tools, in particular parametric systems, to explore alternatives and show why a better representation is needed to adequately support alternatives in parametric systems. In Chapter 3, I describe Shiro, a declarative, dataflow programming language for building parametric systems. Shiro provides language constructs to describe alternatives in a parametric system and provides tools with a multi-state document model. In Chapter 4, I demonstrate how Shiro can be used to solve problems in finance, graphic design, and data analysis. Finally, in Chapter 5 I discuss some of the ways this work can be continued.
Chapter 2

Related Work

Having seen how Markus and Leslie solved their problems by exploring alternatives, in this section I argue that current software tools hinder problem solvers work because of their single-state document model. To make the case for a multi-state document model, I describe the different types of problems people encounter and how exploring alternatives is a perquisite to solving creative problems. I discuss the ways people improvise with single-state tools to workaround their limitations. To demonstrate the need for a multi-state representation to aid problem solvers, I summarize efforts made by researchers and industry to support alternatives and demonstrate that current tools and research focus on the user interface aspects of the problem and have yet to provide a description of a multi-state document model. Finally, I make the case for developing a multi-state representation as a programming language by recounting how languages have been developed by researchers to help problem solvers by providing them with a computational representation of their problem.

2.1 Human Problem Solving

Efforts [45, 46] have been made in the field of artificial intelligence to create systems capable of solving any problem they are given. In the pursuit of this goal, researchers in the field have sought to determine the properties of the different types of problems for which a person might seek a computer’s assistance.

The most basic type of problem is one that easily fits into what Simon [45] defines as a well-structured problem. According to Simon, a well-structured problem must meet the following criteria:

1. There is definite criteria for testing a proposed solution and there must be a mechanical process to the apply the criteria.
2. The goals, initial problem, goal state, and the states encountered during solving can be represented in a problem space.

3. All transitions from one state can be represented in the problem space be they legal or not.

4. All knowledge about the problem can be represented in the problem space.

5. If the state changes the physical world, the necessary natural laws can be represented in a problem space accessible to the solver.

6. The above criteria can be met with practicable amounts of computation and resources access with practicable amounts of search. In other words, it’s possible to compute them.

An example of well-structured problem that meet all of these criteria are optimization, or simulation problems solvable with linear programming or another numerical method.

Some problems people face are well-defined [45, 46], having clearly defined criteria by which to judge a possible solution. In these cases, optimization techniques such as linear programming can be used to arrive at a solution.

Depending on how a problem is defined it may not be well-structured. Instead, it might be ill-structured. Simon [45] defines an ill-structured problem as one that is not well-defined. He gives two examples. Proving a theorem is an ill-structured problem because its solution requires the introduction of new information into the solving process. Playing a complete game of chess is also ill-structured because it requires the solver to continually redefine the problem as it responds to the moves of its opponent. While a single move of a chess game is a well-structured problem, playing an entire game is ill-structured.

Simon goes on to give an account of a ship design process. In the large, the problem is ill-structured. As teams and sub-trades develop the system, they solve well-structured problems and the ill-structured nature of the design challenge becomes evident as the sub-systems are assembled into the final ship. During this process the tradeoffs between needs and capabilities of subsystems are reconciled by the leaders in charge of the overall project.

Goel and Pirolli [15] describe an ill-defined problem as a problem where there is no limit to the knowledge used to frame it and there is no set of legal operations to be used to arrive at a solution.

Simon [45] illustrates this particular type of ill-structured problem when he describes the work of an architect to design a house. A client may provide a basic set of desires for the structure, but the work to design the house is bound only by the architect’s creativity. There is no definitive set of criteria for evaluation and given that new information can introduced at any moment by the architect’s inspiration, the rest of Simon’s criteria for well-structured problems are not satisfied. Design tasks by the nature of how new information and new alternative solutions can be introduced at anytime are ill-structured.
Rittel and Webber [41] further develop the concept of an ill-structured problem when they describe a wicked problem. A *wicked* problem is one without a definite description because it is impossible to provide all the information a solver needs ahead of time. This makes specifying the problem identical to solving it. Wicked problems have no clear stopping state, or moment where the solution is determined completed. Rather, the learning from the latest iteration is used to define the next solution. If work on the problem is stopped it is because of an external factor, like having run out of time, money, or interest. Similar to having no stopping state, wicked problems have no right or wrong answer. Instead, alternative solutions are examined and qualified by the solver. Wicked problems also do not have a finite set of alternatives to be considered. The potential solutions are infinite. In the context of social change, Rittel and Webber note that the cost of choosing a potential solution and carrying it out so it can be evaluated is very expensive as it can take years to be able to see the result. For example, a crime reduction strategy may require several years before the results become obvious. In other fields like graphic design, the cycle is not long.

Thus, to attempt to solve an ill-structured or wicked problem requires the ability to specify multiple solutions based on the solver’s current understanding of the problem so that the solutions can be examined, evaluated, reflected upon, and new solutions formulated as resources allow.

Markus’ task to design a logo by is an example of a wicked problem. Though the client may provide a clear description of the kind of design they seek, there are many ways a designer can achieve the design. Each proposed solution will have strengths and weaknesses and the client and the designer will negotiate to determine which of the designs best matches the client’s desires. The designer and the client will arrive at the final solution by *satisficing* [57], when a solution to a problem is chosen because it satisfies the criteria rather than being optimal.

Bradner et al. [7] discovered that in some cases, where parts of the problem can be described as a system to be optimized, designers used the optimal result as a starting point for other parts of the design process. They give an example for a building design where the result of a structural optimization completed by the engineering team provided the basis for the aesthetic design of the architectural team. In this case, both optimization and *satisficing* were used to reach a satisfactory solution. This reinforces Simon’s argument that specific parts of a ill-structured problem are well-structured and are prime fodder for computational solvers. It also illustrates how computational tools can be used to assist people as they solve ill-structured problems.

### 2.2 Single-state Document Model

With powerful computing and rendering abilities, creativity support tools [43] make it possible to create, view, and analyze solutions. While current software tools make many
problems possible to solve, they have one primary weakness; most software tools are not
designed to help people explore multiple alternatives at a time.

Software tends to be designed with single-state document models [50, 51]. A single-state
document model means the software is designed to work on a single artifact at a time. An
artifact is the thing the software is being used to create. In Photoshop, the artifact is a
bitmap. In Microsoft Word, the artifact is a document. In SolidWorks, it is a 3D model.
Some tools do allow multiple documents to be open at a time with what is called a Multiple
Document Interface(MDI) [36]. For example, Microsoft Word allows a writer to have more
than one document open at time, but only one document can be active at a time. You
cannot simultaneously edit multiple essays at a time—even if they are drafts of what will
become one final document. In addition, a single-state document model will often isolate
the resources of the artifact like styles, or color palettes to the document model currently
being edited. Resources are often not shared across artifacts.

Software developers have conceived a single artifact as the thing their software operates
on and was a necessary first step to develop working tools; however, better support for
alternatives is needed. As illustrated in Chapter 1, use of a single-state document model
has a negative consequence. It requires extra labor to explore multiple alternatives. The
limitations of software have not stopped people from finding ingenious ways to cope with
the limitations of their tools. People have developed a number of workarounds to help them
create and manipulate alternatives.

Terry and Mynatt [49] observe some of the ways that people improvise to create and
manipulate alternatives are to 1) create and toggle layers; 2) to use a large canvas for
multiple designs, and 3) to create file naming conventions. Cycling between two states using
Undo and Redo is also common [49]. The use of layers (See Figure 2.1) is fundamental to
creating digital images and graphic designers have adopted this technique to support their
need to explore alternatives. Designers create layers to represent the part of their design
they want to vary. Each time they want to create a variation they create a copy of the layer
and make their changes. To control which alternative is shown at a given time, graphic
designers toggle the layer’s visibility. If there are several aspects of the design they wish to
vary, background color and typeface for example, the graphic designer can create a group
of layers for each aspect of the design and toggle the appropriate layer’s visibility.

Another approach is to use a large, common canvas for all alternatives. With this
approach designers create all of their alternatives on a large canvas. Designers are able
to see all of their alternatives simultaneously. It allows designers to spatially group their
alternatives.

To represent multiple alternatives in a single view, one seminal work in this area is Marks
et al.’s Design Gallery. Design Gallery is a prototype application used to generate values
for a series of properties used to render a 3D scene. Properties of the generated image,
such as luminance, are used to compare the images and ensure enough of the parameter
space is covered. Once enough images are generated, they are organized into a gallery. Design Gallery provides two gallery layouts. One layout of the gallery is determined by organizing images into a graph based on their inputs. The graph is partitioned and three rows are shown, each corresponding to a row in the graph. A second layout is generated by multi-dimensional scaling each image’s output vector, which represents properties of the rendered image. Thumbnails are shown in the display. Full size versions of the image can be seen by clicking a thumbnail. Design Gallery allows an artist to examine alternative lighting scenarios by navigating through the solutions space for the image.

Smith et al. [47] studied how different layouts affect the problem solving process as they studied the use of tabs, a spatial map, and layers to represent alternatives in a logo design task completed by 18 participants. They found that spatial maps were used the most for reflective activities, analyzing existing designs and deciding what to do next. No other significant differences between interface types were found. The other outcome of the study was a set of recommendations on how interfaces can be designed to better support alternatives. They recommend tool developers make it easy for designers to see multiple designs at a time and to make it easy to share work between the designs. To make it easier to keep track of designs, the authors, based on participant feedback, recommended that tool developers add ways to organize and label designs. The authors conclude that tabs, layers, and a spatial layout of alternatives each have their own merits and should be provided as options for people to choose from as they design.

Designers also create file naming conventions to distinguish between alternatives. For tools, or domains that don’t have a notion of layers, this is the primary way that alternatives are stored. It allows alternatives to be identified in a tool with an MDI. One of the most
serious drawbacks of using a file naming convention is that it isolates each alternative into a single file. Where the layer approach allows common aspects of the design to be shared by multiple alternatives, the use of separate files for each alternative results in each alternative being completely separate. If designers want to change an aspect of the design common to all alternatives, designers must copy their work to all of the alternatives.

In some cases *Undo* is used to toggle between two alternative designs. For alternatives that only involve a single change, *Undo* and *Redo* are used to toggle between two alternatives. *Undo* is the weakest workaround to view an alternative as the alternative only exists in the undo stack. Once additional work is done and more operations are stored on the undo stack it becomes difficult, if not impossible, to retrieve the alternative. New operations added to the stack bury the operation representing the alternative and most tools do not provide a graphical list of the operations in the undo stack. (Photoshop does.)

Another workaround to single-state document models observed by Hartmann et al. [17] is how programmers use code comments to try different program configurations. A programmer creates alternate implementations of parts of a program’s code and use code comments
to toggle which implementation is active. The drawback of this workaround is that a programmer can only run a single configuration of the program at a time, which makes it difficult to compare the runtime behavior of the two implementations.

One of the ways that data can be represented in a single-state document model is with a parametric system. A parametric system is a type of constraint system where computation is represented as a graph. The vertices of the graph are referred to as nodes and represent operations on data. The edges of the graph represent the dependencies of one operation on another and indicate how data will flow. One of the most recognizable examples of a parametric system is a spreadsheet. In Figure 2.3, cells A1 and B1 are added together in cell C1.

![Figure 2.3: A Numbers spreadsheet where cells A1 and B1 are added in C1](image)

The benefit of representing a tool’s document model as a parametric system is it allows for parts of an artifact to be built with respect to one another. If A1 is changed to 10, when the parametric system evaluates, the result of C1 will be 17, which reduces the amount of work required to change the artifact.

Users of single-state parametric systems can use a Recorder Pattern [56, p. 278-280] to watch and record snippets of designer action. This requires sub-models to be configured in a very specific way. The Recorder Pattern requires a special type of node to record parameter values each time the system updates, the recorder node. The recorder node may have a parameter that determines if the values are recorded. Each time the parametric system evaluates the value of the parameter it is appended to an array of previous values. The array of previous values persists between evaluations of the parametric system. Multiple values can be recorded by storing an array of values per recorded parameter. It also is worth observing that a node like this is not purely functional as it stores state for the next run. The Recorder Pattern is another type of workaround as it requires the parametric system to store state between evaluations. It also does not allow variations in graph structure to be represented.
2.3 Interfaces for Alternatives

A variety of tools and user interface techniques have been developed to allow people to explore alternatives. Work has been done to make it easier to use a spreadsheet to explore what-if scenarios. Truvé [53] presents a spreadsheet with two widgets called a slidercell and a micrograph in its cells. The slidercell is used to select the value of the cell. The micrograph displays the graph of the formula. Rather use than guess and check, Truvé’s approach allows a person to move through a range of values using the slidercell. The micrograph allows a person to view the result of this exploration by seeing it in the context of the graph of the formula. The author argues that this approach might allow a person to understand better the behavior of a formula than if a solution is found with a goal seeker.

Combining a parametric system and a view that allows a person to see multiple alternatives simultaneously, Aran Lunzer [27, 28] introduced the concept of a subjunctive interface as a way to make it easier to explore alternative scenarios. A subjunctive interface is an interface that simultaneously embodies all the “what-if” situations the user wishes to explore [27]. Non-subjunctive interfaces limit users’ exploration to a single set of parameters at a time and make it difficult to manage multiple scenarios. In tasks like planning a trip, or designing a piece of furniture, it is desirable for the users to keep their options open and to easily view and compare a range of different outcomes. The goals of a subjunctive interface are to provide overviews of different scenarios, a means of comparing the the scenarios, and to help people reduce the number of alternatives they need to consider by filtering out those not of interest. To achieve these goals, a tool must support multiple realities or alternative states and allow the alternative states to modified in parallel.
To explore these ideas, Lunzer and his colleagues built a number of prototypes that implemented the principles of a subjunctive user interface. Some of these prototypes are an application to explore census data, C3W, an application to connect web resources [29], and RecipeSheet [30, 26], a platform for creating spreadsheet-like connections between calculations. RecipeSheet embodies the most complete implementation of a subjunctive interface in Lunzer’s work and was used to implement domain specific versions of the tool. One example is OncoRecipeSheet [25], an application that provides cancer drug simulation functions in RecipeSheet. RecipeSheet allows users to create recipes from predefined functions or functions written by the user in Squeak, a SmallTalk implementation. Recipes take inputs, called ingredients, and provide outputs called results. Networks of recipes can be combined to create nested recipes. When recipes are nested in a recipe, the outputs of the nested recipe are those involved in the node’s calculations. Unused outputs are not exposed by the nested recipe. Connected recipes form a directed acyclic graph. Evaluation of the graph is invoked by the user making changes to parameter values via the UI, or by pressing a button to explicitly request evaluation. Calculation of multiple scenarios is supported by recipes that accept lists of ingredients. If a recipe has an input that accepts a list of ingredients, each of its outputs will produce a list of values, the result of computing the cross product of the inputs lists. This approach is similar to replication [4], a feature of parametric CAD tools, except replication has a more general notion of a replication strategy. A cross product is one possible implementation of a replication strategy.

Replication is a technique employed in parametric CAD tools to allow functions to handle collections. It also serves as a form of implicit iteration. Iteration cannot be explicitly represented in a parametric system without making the graph cyclic, which makes the graph evaluation algorithm more complicated to implement and the graph more difficult to understand. Replication allows nodes in parametric systems to operate on both lists and single values with a single node definition as a single value is represented as a list with one element. *Shortest List, Longest List, and Cartesian Product* are some strategies for computing replication. Replication strategies determine how an function handles a list of elements. For example, let \( a = \{3, 4, 9\} \) and \( b = \{12, -2, 5, 6\} \). If the function we want to apply is binary addition, for *Shortest List* the result would be \( \text{result}_{\text{shortest}} = \{15, 2, 14\} \). For *Longest List* the result would be \( \text{result}_{\text{longest}} = \{15, 2, 14, 6\} \). For the Cartesian product, the result would be \( \text{result}_{\text{cartesian}} = \{15, 1, 8, 9, 16, 2, 10, 21, 7, 14\} \).

After exploring the concept with a number of prototypes, Lunzer and Hornbæk [31] conducted three studies to compare a simple interface and a subjunctive interface of a census browser. In Study 1, Lunzer and Hornbæk had 20 participants completed a series of comparison tasks. The experiment used a within-subjects design, where each participant completed nine tasks with each interface. The independent variable was the type of interface used and the dependent variables were accuracy, number of marks made on paper, task completion time, a user satisfaction questionnaire, and a mental workload assessment. They
found participants had higher satisfaction when using the subjunctive interface and relied less on making notes on paper to complete the task.

In Study 2, the authors re-examined the time it took participants to complete the task because in Study 1, it was not clear if the subjunctive interface decreased participants task completion time. Participants completed some task more quickly and some more slowly. In the feedback portions of the study, participants told the authors they would like to have more time to learn the subjunctive interface. The authors agreed because they observed participants having trouble developing and executing strategies to complete the task. The authors then provided nine more tasks to the participants that they completed in four sessions separated by a day. For the fifth session, participants returned to the lab and completed the tasks and satisfaction questionnaire completed in Study 1. When given more exposure to the subjunctive interface, participants completed tasks 27% more quickly with the subjunctive interface.

In Study 3, to learn more about how participants use the subjunctive interface, the authors asked seven participants to complete two sessions. In the first session, participants completed twelve tasks and were allowed to ask questions. In the second session, participants were asked to complete two tasks and to think aloud as they did them. The participants’ work was video recorded. To analyze the video, the authors transcribed and coded the activity. The authors observed that subjects used the ability to create scenarios as a way to hold information as they examined other questions on their way to completing their task.

In addition to identifying the problem of single-state document models, Michael Terry explored interface designs that simultaneously allowed multiple alternatives to be viewed and edited. Terry et al. [52] present an interaction model called Parallel Paths and makes variations first-class objects in the system. Terry et al. define variations as “alternative solutions to a given problem at a point in time” [52, p. 713]. They distinguish a variation from an iteration arguing an iteration represents the same solution to a problem at a given point in revision, whereas a variation is a separate solution.

Parallel Paths is a graphical editor augmented with tools to manage variations. To create variations with Parallel Paths (PP), the document is duplicated for each variation, and is superimposed within the same workspace. Each variation manages a record of the commands applied to it. To move through the record of variations, the authors introduce the term skating. Skating is moving a variation to a previous state without the use of undo. To use undo, the user risks losing a variation. Using a radial divider with a movable center, multiple variations can be shown in the same workspace. Moving the center and radial controls how much of each variation is shown. They also added support for editing one or more variation at a time.

Terry and Mynatt [50] describe Side Views, a user interface component based on a tooltip designed to allow people to see the results of a command before they commit to executing it. Side Views were designed to help people explore different parameter settings.
for commands as they work on an open-ended or ill-defined problem. Side Views appear like a tooltip when the user’s mouse hovers over a menu item. The result of the command is displayed in the window. What makes a Side View different from a tooltip is that it can be made persistent. A tooltip disappears after the mouse moves off its triggering item. The user can choose to retain a Side View after moving off a command. The Side View will give a preview of the command for whatever data is selected. A series of Side Views can be strung together to show the result of a series of commands. The parameters of each Side View can be adjusted and each intermediate result is displayed. Multiple Side Views can be opened to allow a user to compare different parameter settings. When a command has parameters, a parameter spectrum is used to display snapshots of the result for a given value. Side Views provide a way for users to preview and explore a variety of commands and parameters settings.

Side Views were implemented in a rich text editor and in GIMP. The authors used the tooltip events of the applications to popup Side Views. Because of the volume of information in a Side View, the authors displayed it for 10 seconds instead of the normal 7 seconds. Also, when a person’s cursor enters a Side View, the timer is paused until it leaves. Side Views also linger longer after the cursor is move away from the command. These timing adjustments were done to make it possible for a person to have time to process the increased information the panels provide.

Side Views are an interesting approach to displaying alternatives. They are a flexible tool. You can make as many of them as you want. You can link them together to create little programs. Parameter spectrums add an additional level of preview for a command by displaying a sample of what the design would look like with a range of paramater values applied. With all this help, people should find it easier to explore data and generate solutions they like, or match their criteria. In the midst of the parameters and previews, artists are likely to see a result they did not intentionally consider. Side Views provides a means for people to preview many different parameter settings.

Hailpern et al. [16] describe Team Storm, a sketching tool to support collaborative and parallel work. The authors conducted a study to discover the requirements for supporting this type of work and built a prototype to meet the needs they discovered. Team Storm supports co-located, collaborative, and parallel work via connected workspaces. Each person’s sketch tool provides a view of their own private workspace and the group workspace. The group workspace might be displayed on a screen in the room, but it is not a requirement. People make and arranges sketches in their private workspaces. A workspace is an infinite canvas where sketches can be placed. When they are ready, they can share their work with the rest of the group by dragging the sketch to a group workspace. When the sketch is shared with the group, they are permanently shared. The sketch’s owner can choose whether to allow others to edit the sketch or to only allow them to see it. Alternatives are represented by unique sketches positioned in the workspace. Team members can further de-
velop a sketch shared with the group by copying it to their own personal workspace. Team members can jointly work on a sketch shared with edit permissions by selecting the sketch in the group workspace to make it the active sketch in their tool. Their strokes are shared to all group members as they are made. Team Storm allows groups to create alternatives by creating additional images in a workspace and to share alternative designs with a group workspace.

2.4 Subjunctive Interfaces in Parametric Design Tools

The concept of subjunctive interfaces has been explored in parametric design. Chen [11] presents a formalism to describe variations. Variations are collections of different values for a set of properties in a parametric system. Chen views variations as a means to represent alternatives. To represent variations, Chen describes a variation head and a variation space. A variation head is the set of nodes that do not depend on any other nodes called graph independent nodes. Each property of a node in the variation head has a concrete value and does not include an expression. A variation head does not have to contain all of the graph independent nodes in a parametric system. It can contain some. The key insight of this work is there is a group of nodes whose input values control the behavior of the rest of the parametric system. A variation head is a sort of “interface” to the parametric system.

Variation heads are collected into groups called a variation space. A variation space is a list of variation heads. He treats the list of variation heads like an array in a programming language like Java, or C. A variation space is then combined with an operation called unification to provide a means to generate additional variations. Unification is an operation that determines how two objects can be combined. Chen defines unification in terms of subsumption saying that an object can be unified with another and the result is the subsuming object.

Chen formally describes a series of operators to generate additional variations. The variations are represented in terms of variation spaces and variation heads. The two objects become the fundamental elements by which one can construct alternatives. What is interesting about this approach is that it provides a basic specification of a vocabulary for expressing alternatives.

The weakness of the formalism is it does not take into account how differences in the parametric structure can also be used to represent alternatives. His description is limited to the inputs given to a parametric system. With these weaknesses, the work is an excellent start and needs to be expanded to more fully describe alternatives.

Kolarić et al.[24] describe a series of design principles for building tools with support for alternatives based on an expert review of their prototype vector drawing tool called CAMBRIA. It allows designers to create and edit collections of alternatives. Alternatives can be displayed in a grid, or user-defined layout. Edits to alternatives can be made in a
juxtaposition view, where designs are shown beside one another and a superposition view, where designs are shown on top of each other. CAMBRIA introduces two operations to make it easier to manipulate alternatives: pass variable allows designers to move/copy an element into other alternatives; pass value allows designers to propagate a property value to alternatives with a similarly named element. As a result of their study, Kolarić et al. recommend that tool implementers make it easy for designers to create collections of alternatives and be able to label them for later reference. Nested collections would provide a more robust structure of organization. CAMBRIA only supported collections of alternatives one level deep. They also recommend tool builders make it easy to edit an alternative, or a set of selected alternatives simultaneously.

Shireen et al. [42] present a prototype design for a subjunctive parametric editor. To solve the problems described by Terry and Mynatt[51] with single state tools, the authors present a subjunctive parametric editor to support simultaneous editing of alternatives in parametric systems. The prototype focuses on the design of an interface for graphical programming environments like those present in Generative Components(GC) or Grasshopper. The goals of the design are to enable the creation of alternatives, to be able to combine different parts of different designs, to reuse components, and to allow variations to be manipulated simultaneously.

In their proposed design, parallel editing is made possible by adding a panel below the graph view of a parametric system. Each alternative is represented by a panel. If a designer wants to change a node’s values for a specific alternative, the node can be added to the panel. Changes to these nodes are local to the alternative. Designs can be changed by changing nodes shared by all the variations, or by changing nodes specific to the alternative. Nodes common across alternatives can be brushed. To view an alternative, a single alternative can be selected and only its 3D view and parametric graph is shown. When the tool is in single alternative view, the complete parametric graph for the alternative state is displayed. Shared nodes and local nodes are merged to create a graph.

Implementing a similar design to the previous paper, Zaman et al. [58] describe GEM-NI, a design tool that extends NodeBox [12] to support alternatives. NodeBox is a parametric and generative design tool. It allows a designer to draw images by creating and connecting nodes, representing values and drawing functions, in a graph. NodeBox provides three main views: a result view, a graph view, and a parameter view in a workspace (See Figure 2.5. GEM-NI extends the NodeBox interface by supporting multiple parallel workspaces. Each NodeBox workspace is organized vertically with the result view on the top and graph view below. The parallel workspaces are spread out horizontally and use up as much screen space as the application is given.

GEM-NI allows users to edit designs in parallel. The properties and positions of node in the alternative graphs are kept in sync. User can sandbox a design so that changes to it are not shared with the other alternatives. Similarly, the design can mark an alternative as
idle so changes made to the other alternatives won’t affect it. To support the generation of alternatives, GEM-NI provides merging and branching operations. When a user decides to combine two alternatives, the graphs of the two alternatives are merged. Absented nodes are created and connections are recreated as intelligently as possible, though some merges may require the user to fix parts of the graph not functioning after the merge. GEM-NI maintains the undo stack for each alternative. When an alternative is cloned, its undo stack is copied as well. Because the undo stacks are preserved for each alternative it is possible to *skate* back in time and generate a new a alternative, a branch in history. GEM-NI also allows users to generate alternatives in a design gallery. Users choose two alternatives and select the properties they want to explore. GEM-NI uses this selection to compute the Cartesian product of the designs. Users can navigate through the generated designs and select the ones they like to create instantiated as alternatives in the workspace.

Hartmann et al. [18] report the result of two studies that compare users use of d.note and freeform sketches to suggest modifications to a user interface design. d.note is a user interface prototyping tool that allows designers to express user interfaces in control flow diagrams. Interfaces are described in terms of states and transitions. The goal of the study was to compare how designers made revisions in d.note in comparison to only being able to suggest changes by sketching or writing on images. A criticism of digital tools is that they do not allow designers to rapidly make changes, as significant development skills are required to change buttons, or screens. d.note has tools to allow designers to add sketches
alternative designs by sketching. Alternatives states can be represented in the diagram alongside the original state. Designers can then choose the state alternative they want by selecting a radio button. In addition to sketching changes, designers can refactor the state diagram and edit images directly in d.note.

Hartmann et al. [18]'s first study compared the authoring of revisions with and without d.note. There were 12 participants. Each participant was asked to do two revision tasks. The without d.note task allowed them to use Sketchbook, a drawing tool, to annotate an image of a state diagram. Without d.note participants wrote more comments, and did few deletions. In their post-study questionnaire, the most frequently mentioned advantages of d.note were the ability to make functional changes and to see the result immediately. The second study investigated how designers interpreted the results of revised designs. Eight participants were shown eight annotated designs from the previous study and were asked to list all the revisions they understood and questions they had. Participants asked for few clarifications using d.note, but had a difficult time understanding why changes were needed.

2.5 Alternatives in Text Authoring Tools

Support for alternatives has been explored in development tools. Hartmann et al. [17] present Juxtapose, a tool designed to help interaction designers build and test multiple alternatives in one development environment. Juxtapose consists of a source code editor that allows simultaneous edits to alternative source code, a runtime environment to allow the results to be viewed in real-time, and a hardware tuning device. The authors developed
three types of runtime environments: one for the desktop, the mobile phone, and one for micro-controllers.

In a series of exploratory interviews, the authors discovered that designers use a couple of workarounds to explore different solutions. Some designs require careful tuning of parameter values to give the desired effect. To solve this problem, designers expose the parameters as GUI controls to allow them to change the values in real-time. Other designs require playing with different ways to write an algorithm. Designers comment and uncomment code as needed to test different approaches. These needs resulted in the linked source editor and the hardware interface for tuning parameters. The runtime environment was necessary to see the results of the alternative blocks of code.

18 participants with HCI backgrounds were asked to do three tasks that required them to manipulate alternative designs. All participants were able to use the linked source editor without issue. Participants were faster with Juxtapose in a task to find the correct parameters to match a supplied design. Participants explored an order of magnitude more parameters values with the alternatives tools than without. The study indicates the benefit of design/development tools with support for alternatives.

Elkhaldi and Woodbury [13] describe Alt. Text a text editor designed to support alternatives in document hierarchies. With Alt. Text writers explore the space of possible document structures by creating alternative document outlines, sections, and passages. Documents are composed of outlines of sections and sections consist of passages. Writers control how the alternative is created by choosing which sections to include in the document outline and which passages to include in the section.

To assist in the writing and document generation Alt. Text provides a graph-based document outline tool and text editors. The graph-based document outliner allows writers to specify the flow of their document through sections by drawing lines between graph nodes representing the sections. A writer can also open multiple text editors to view any combination of passages, sections, or documents. Shown in the editor is a document outline that allows a writer to configure the passages and sections they want shown.

2.6 Support for Alternatives in Commercially Available Tools

A variety of existing commercial software products provide limited support for alternatives. Source code version control tools provide some of the most robust support for alternatives of the products I surveyed. Even with their robust representations of the changes to code over time, source code version control systems are still limited to single state data models. Productivity software like Excel and design tools like Photoshop and Grasshopper also have support for alternatives, but are limited by their single state data models.
2.6.1 Version Control Systems

*Git* [2] is a distributed version control system used to manage changes to source code. *Git* represents changes to source code as a directed acyclic graph of commits. Each commit represents a snapshot of the source code at the point in time it is made.

When a programmer makes a commit, *Git* computes the SHA1 hash of each of the files and stores a reference to the file tree containing the state of the files. The file tree consists of references to *blobs*, which can represent files, or directories. To save space, *Git* only stores a file once and identifies files, existing or new, by their hash. That *Git* stores a copy of every state of a file in a source tree is where it differs from version control technologies like SVN [1], which store only a single copy of a file and generate the most current version by applying a sequence of deltas. Commits store references to the current state of the source files at the time they were created and to their parent commits. If there is more than one commit, the commit is the result of a merge.

While *Git* represents the evolution of source code over time and uses branches to represent variations of the source code, only one branch can be checked out at a time, making it functionally equivalent to a single-state model. Having access of multiple branches simultaneously requires the repository to be cloned for access to each branch.
2.6.2 Microsoft Excel

Microsoft Excel provides basic support for alternatives with a feature called *Scenario Manager*. *Scenario Manager* allows an Excel user to create sets of cell values to explore, called Scenarios, alternative spreadsheet calculations. In the Scenario Editor, a user selects cells from the spreadsheet he wishes to explore (See Figure 2.8a) and provides a new value for the cell. A formula can also be entered as a cell value, with its result being set as the cell value. The formula is not stored; only its value is stored. As a result, it is not possible to store alternative computation in a scenario, only alternative values.

To show the results of the scenario, the user clicks “Show” and the results are shown on a combined spreadsheet, as seen in Figure 2.8b. The *Scenario Manager* provides features to edit, add, delete, and merge scenarios.

2.6.3 Photoshop

Photoshop provides Layer Comps [3] and the ability to save states of a design to support the exploration of alternatives. By creating a *Layer Comp*, a designer can record the visibility, position, or effect of a layer. A designer can give each *Layer Comp* a name and *Layer Comps* are saved as part of the file. A designer can also capture a snapshot of the current state of a design in the history panel. Capturing a snapshot of the design’s state allows the designer to recall the state from the history panel without replaying each step between it and the current state. It allows the designer to jump to the state. The drawback of these
states is they only last as long as the file is open. As soon as the file is closed, history is deleted and the states are lost.

Figure 2.9: Photoshop Layer Comp controlling visibility

2.6.4 VisTrails

VisTrails [5, 10, 9, 14, 44], is a scientific visualization and workflow management tool. Scientists create visualizations by creating pipelines of modules to specify how their data should be transformed and rendered. The output from a pipeline is shown in the visualization spreadsheet.

Alternative visualizations of a pipeline are created in the explore workspace. In the explore workspace, users can specify parameters they want to explore and determine how they want the variations displayed. To choose a parameter to explore, a user drags the method from the Pipeline methods pane to the explore canvas. In Figure 2.11, four visualizations with values ranging between 30.0 and 75.0 are created for the `vtkContourFilter` values. The visualizations are shown as horizontal row.

The user can choose between linear interpolation and a user-defined function (implemented in python) to create the values in the range, or a list can be provided as the set of values to explore.
At any point a user can store all of the parameters for a pipeline by giving an exploration a name and saving it. VisTrails also records all of the changes made to a pipeline (in manner similar to SVN) and provides a view for comparing pipelines.

In the development version of VisTrails (2.2 at the time of writing)\(^1\), the developers have begun to explore the implications of replication on VisTrails. They have added support for custom replication strategies (See Figure 2.12).

VisTrails provides support for exploring visualization parameters and records the changes users make to their visualizations pipelines. VisTrails has strong support for alternatives, but it does not allow scientists to create alternative structures of their pipelines without creating a second version of the pipeline. It only allows scientists to vary the values of module parameters. This limitation prevents the structure of pipelines from being explored in the same way as module parameters.

### 2.6.5 Grasshopper

Grasshopper for Rhino provides rudimentary support for alternatives. It allows designers to save input parameters in a manner similar to Excel. In Figure 2.13a, the designer stores values for the height and radius parameters of a cylinder. To use the state manager, the designer must create an input slider for each of the values, he wants to store. To save a

\(^{1}\)http://www.vistrails.org/usersguide/dev/html/list_handling.html
state, the designer makes changes in the graphical programming view and opens the State Manager. In the State Manager, he can choose the values he wants to save and can provide a name for the state.

Restoring and deleting a state is done through a drop-down menu (See Figure 2.13. Once states are created, they cannot be edited. To make a change the designer must restore the design to the state he wants to modify, delete the state, and save a new state with the same name.

2.7 Languages

To support alternatives, application developers need a way to describe what alternatives to compute. People use textual programming languages to represent designs and computation. The primary focus of this thesis is alternatives as they relate to parametric systems and since a parametric system is a rudimentary form of a constraint system, I surveyed several constraint based programming languages. I also surveyed Worlds, another language-based approach to representing alternatives developed by colleagues of Aran Lunzer at Viewpoints Research Institute.

One approach researchers have taken to try to better support problem solvers is to develop constraint systems and languages that compute solutions to problems. One of the earliest attempts to use a constraint system to support design was Sutherland's seminal
Sketchpad [48]. In 1964, Sutherland built Sketchpad, a drawing tool that used a constraint system to help reduce the amount of time it took to make a drawing. Sutherland’s vision was for tools like Sketchpad to provide programmers with a different means of expressing programs than by typing code. He wanted people to be able to do things like draw to make their programs. Users of Sketchpad used a light pen to draw shapes and to create constraints between them. A drawing could be used to create additional instances. Changes made to the master (original) drawing would affect all of its children. Other supported operations included recursive delete, and recursive merges. These operations provided users with the tools to create and manipulate drawings composed of other drawings and their instances.

Borning’s ThingLab [6] is a language to describe simulations with constraints. Borning’s goal was to extend the idea demonstrated by Sutherland in Sketchpad that people could
program by manipulating graphical representations of their problems. ThingLab is an extension of Smalltalk that added constraint solving, multiple inheritance, prototypes, and symbolic paths. His goal was to allow people to build domain specific building blocks for simulations and then to use these building blocks to specify simulations.

In ThingLab, Objects were built from primitives and other parts. Each part “knew” its name, its type, the constraints that applied to it, and the merges it participated in. New types of objects could be created with inheritance. When an object inherited another it gained access to the properties of its superclass. Objects and constraints are created in ThingLab by direct manipulation. For example, lines are drawn to create a quadrilateral. Additional lines and constraints are added to create a parallelogram inscribed in the quadrilateral. Because of the constraints, no matter how the points of the quadrilateral are moved, the sides of the parallelogram remain parallel.

Constraints satisfaction happens in three steps: first, an object is sent a message plan, or a description of the change to be made. When the object receives the message, it attempts to complete the request by using a single pass ordering of the constraints. If a solution can be found, the changes are made instantly. If not, the kernel reverts to using relaxation to satisfy the constraints. Relaxation is an approach to constraint satisfaction where the solver tries a series of values for a set of constraints. It continues until the error has been minimized. This approach only works for constraint systems of numerical values.

Piela et al. [40] developed ASCEND, an engineering design language to model systems in chemical engineering. ASCEND is designed to help engineers specify large systems of equations, and is domain independent, as it does not contain any domain specific constructs. It is a strongly-typed language that is both declarative and procedural. Piela et al. make a case for the addition of the procedural constructs by arguing they are helpful to set values and initial conditions. It implements object-oriented programming concepts like inheritance. Objects are called Models and are composed of inputs and outputs, called Atoms. Atoms contain values and are often given units. With strong-typing and unit analysis, the compiler provides a means of verifying whether the system is correctly specified.

ASCEND is not an executable language; rather, it is a language for specifying a data structure to be manipulated by other tools. Its purpose is to be the lingua franca of a constellation of simulation tools. Individual tools provide features specific to one part of working with the language. Some of these tools include a Library for creating, viewing, and manipulating hierarchies, Sims that are created from the models, a Browser for exploring objects in a sim, and a constraint solver that computes the result of a sim. When the model is converted into a sim, the model is converted into a database of equations. All of the tools are kept in sync with the database, so if changes are made in one view, all other views are updated to reflect the change.

Warth et al. [55] describe Worlds a programming language construct that captures program state in an imperative programming language like Javascript or Smalltalk. Each
program provides a root world (referenced with `thisWorld`) from which new worlds can be created with the `sprout` command. Values are resolved by searching the world hierarchy to find the first world where the property is defined. When `sprout` is called a new world is created. This new world can be used as the execution context for code via an `in {...}` block. Changes in the sprouted world can be written back to the parent world with the `commit` command. `commit` effectively clears any data stored in the child world and stores it in its parent. The construct allows computation to be completed in isolation from the root symbol table.

Worlds make it easy to implement `Undo` and `Redo`. Before completing a command, a new world is sprouted and then command is issued in the context of the new world. The newly modified world is saved in a stack. `Undo` is completed by popping the previous world from the stack, so it’s no longer referenced and is deleted in the next garbage collection run. This returns the symbol table to its previous state. This pattern makes it possible to implement tree-undo and cleaner exception handling. Worlds can also function as a sort of module by encapsulating object implementations. Code that needs access to objects with certain functions can be executed in the context of a world with the correct implementation, making it possible to monkey-patch, a programming technique where functions are added to an object after its initial definition, internal implementations and not mess up other code that might depend on the previous implementation.

2.8 Towards a Multi-state Document Model

Solving ill-defined or wicked problems requires that people be able to explore the space of alternatives. To explore alternatives with most design software available today requires workarounds such as toggling layer visibility, placing all alternatives on a common canvas, and toggling undo and redo. In parametric systems, designers can use the Recorder pattern to store and recall the values of parameters between executions of the system, a technique which violates the purely functional nature of the parametric system and forces alternatives to be represented outside the system. To avoid forcing people to use workarounds, better support for alternatives in creativity support tools is needed.

To this end, researchers have been aware of the need to support alternatives in tools for a while now and some support does exist in commercially available tools. For most commercial tools, supporting alternatives is an afterthought and is not a prominent or well designed feature of the tool. What should be clear from the research I have summarized is that exploring alternatives is a fundamental part of human problem solving, be it a problem in design, data analysis, or some other domain. The research I reviewed in this chapter shows the need for and promise of tools that provide ways for people to create and manage alternatives. Current research indicates parallel manipulation and the ability to easily compare alternatives will reduce the amount of effort it takes to work with alternatives. This chapter
summarizes the research done to build user interfaces with support for alternatives, yet the implementation of a multi-state document model to support this interaction has received little attention in the literature. Most of the papers I reviewed focus on the user interface tools needed to support alternatives and the details of how the user interface interactions interact with the document model is left to the reader to guess. The problem this lack of examination presents is it limits future exploration of user interface designs to clever hacks and use of “Wizard of Oz” implementations. There is no multi-state document model that interface tool builders can leverage in their designs. To fill this gap, the remainder of thesis describes a multi-state, parametric document model.
Chapter 3

Shiro

To better support the exploration of alternatives, tools need access to a multi-state document model. Future user interaction research depends on the availability of a computational foundation on which to continue to develop interface techniques. Current user interface research is limited by the lack of access to an easy-to-use, multi-state document model. Researchers use prototyping techniques like “Wizard of Oz” simulations or modify existing tools to meet their needs. These approaches allow researchers to explore new user interface ideas and spend little time building infrastructure. The challenge this ad hoc approach presents is that it limits researchers to what they can improvise with existing tools. Often, their work lacks integration with a multi-state document model. To gain the knowledge to build tools that support alternatives, the computational aspects of the problem also need to be examined.

In an ideal world, research in user interface techniques and multi-state document models would happen in collaboration, so that work to understand the user interface techniques needed to support alternatives informs the underlying computational model and knowledge of the computational requirements of a multi-state model would inform the design of the user interface by making designers aware of possibilities that might not be obvious. For example, the cost of computing an alternative will have an effect on how many alternatives a user interface encourages a user to make. Computing an alternative might be inexpensive and thus a user interface design can afford the regular creation and destruction of alternatives. If alternatives are expensive to compute, user interface designers will have to be intentional about how easy they make it for new alternatives to be created. To ground user interface work in reality, a multi-state computational model should be used. Since exploring alternatives is such a fundamental part of human problem solving, it only makes sense that our tools support it and therefore work needs to be done to solve both aspects of the problem. Given the variety and volume of research done on user interfaces, I think developing a more robust computational model is the next step towards putting tools with support for alternatives into the hands of the general public.
The impetus for this work could come from any number of fields. My interest in the topic is inspired by the fields of generative design and visual analytics as the result of participating in the collaboration between the visual analytics research and computational design research groups at Simon Fraser University. As a result, the examples and scenarios I consider are chiefly from those two fields, though other domains may also provide excellent subject matter for examples. This scholarly heritage has also prompted me to explore a multi-state document model from the point of view of how to support alternatives in parametric systems. Parametric systems are fundamental to generative design tools like Grasshopper and Generative Components and have also been applied to visual analytics in tools like CZSaw [20] and VisTrails [5] and form my primary focus. Even with this focus, the technology and issues I discuss in this chapter can be applied to any other domain where it’s natural to express computations using graphs of functions. Thus, the scope of my work is limited to exploring multi-state document models in the context of parametric systems as applied to generative design and visual analytics.

In this chapter, I address the question, “Is it possible to extend a parametric system to make it possible to represent multiple alternatives in a single system definition?” With the introduction of a new language construct, the subjunctive node, I describe the design and implementation of language for describing multi-state parametric systems.

3.1 Shiro at a Glance

Shiro is a declarative language for specifying parametric systems containing alternatives and provides a multi-state document model. Shiro is implemented as a language with a runtime that exposes an API for modifying it’s current state rather than as a library of code because having a human readable, human writable representation makes it accessible to both designers and tool builders. A textual language provides the most basic of user interfaces for the system. Designers comfortable writing code can use the language to specify designs with alternatives. It will even support the improvisations of clever researchers if they modify existing parametric tools to output Shiro code. Tool builders can integrate the runtime in their software and use its API to manipulate the parametric system and alternatives via user interfaces they design. Beginning with a language also forces me to focus on the computation required to evaluate the language instead of domain specific ideas that would need to be present in an end-user application. It is declarative so a designer only has to describe the computational relationships between parts of their design rather than how the design is computed. It is up to the Shiro runtime to manage the computations expressed in the relationships of the design. Informed by the user interaction research I have previously summarized, this work is a bottom-up, first principles-based approach to supporting alternatives in a parametric system. In this chapter, I describe the syntax, implementation of the Shiro runtime, and share some of the implications of my approach.
Before diving into the formal description of the language, I want to give you a taste of Shiro. Shiro is designed to allow you to easily express a basic solution. As your solution becomes more complex, it provides you with tools to structure your work so it is easier to understand.

Code in this section will be shown in listings to make it easier to read. The numbers on the left side of the black line are line numbers. When a program is built over multiple listings, the code added to the previous listing will be highlighted in yellow as seen on line 5 in the listing below.

```
1 // code goes here
2 ...
3 ...
4
5 // code added in this step will be highlighted
6
7 // more code
8 ...
9 ...
```

Listing 3.1: Example Listing

The result of running the program is shown in console output form. The following output reads, “The value of foo, which is of type Power, is 144.0.”

foo#<Power results:[144.0]>

With literals, expressions, named ports, and nodes, you define Shiro programs. Let’s start with the simplest statement in Shiro - the literal.

```
1 3
```

Listing 3.2: Literal integer 3

On line 1 of Listing 3.3 is the literal representing the integer 3. Shiro has literals for doubles, integers, booleans, and strings.

```
1 3 // positive literal integer
2 -4 // negative literal integer
3 3.3434 // positive literal double
4 -4.33 // negative literal double
5 'Hello World!' // literal string
6 true // literal boolean value
7 false // literal boolean value
```

Listing 3.3: Literals

Each of these literals is a function that returns itself. You can create a variable, or a named port by declaring the function with a name.
This statement creates a port named “a” of type `Double` with the value 11.0. This statement is similar writing `100.0` except the `Double` instance is named “a”. With literals and named ports, you can write expressions.

```
1 a Double(11.0)
2 a + 7
```

Listing 3.5: Using a named port in an expression

```
#<Add results:[18.0]>
```

Because the infix + is used, the `Add` port does not have a name, so there is nothing before the # in the output. The unnamed `Add` function’s value is 18.0. Expressions can be written using infix notation or in explicit dataflow form.

```
1 a Double(11.0)
2 sum Add(a, 7) // sum is 18.0
```

Listing 3.6: Explicit dataflow form of Listing 3.5

```
sum#<Add results:[18.0]>
```

These expressions are equivalent with the difference being to declare the `Add` function requires me to give it a name. I chose `sum`. Being able to write expressions in infix notation is helpful when you want to use an expression to modify the value of an argument without explicitly declaring the ports in the dataflow. Here is another example:

```
1 a Double(11.0)
2 r Power(a + 1, 2) // r is 144.0
```

Listing 3.7: Declaration of dataflow using infix expressions

```
is equivalent to
```

```
1 a Double(11.0)
2 b Add(a, 1)
3 r Power(b, 2) // r is 144.0
```

Listing 3.8: Explicit declaration of dataflow

The value of `result` is:

```
r#<Power results:[144.0]>
```
Expressions follow the standard BEDMAS\(^1\) rules with one exception; there is no exponent operator. \(12^2\) is not a valid expression in Shiro. You must explicitly use the `Power` function.

When you write literals, expressions, and named ports, you can graphs of functions. All of the examples I have shown so far are examples of how to define an *implicit graph*. Internally, all of these listings are statements in the graph named `^`, a name that can be used to reference the graph if necessary. The runtime was implemented this way to reduce the amount of code you need to write to get started. You can treat Shiro like a scripting environment and progressively add more structure as your solution requires it.

Rather than write your code in one giant block, which quickly becomes tedious as the program grows, nodes provide you with a way to modularize your code. Nodes allow you to encapsulate subgraphs in named containers. The name of a node must start with a capital letter. For example, a box is defined:

```plaintext
node Box begin
  input length Double
  input width Double
  input height Double
end
```

Listing 3.9: Node declaration

In this statement a node `Box` is defined. It declares three ports each marked as an input. The keyword `input` is an access modifier. There are three access modifiers in Shiro: `input`, which allows the port to be read from and set, `output`, which allows the port only to be read from, and a port declaration without any modifier that makes it private and only accessible within the node.

```plaintext
node ShortBox begin
  input length Double
  input width Double
  height Double (1.0)
  output volume Double (length * width * height)
end
```

Listing 3.10: Node defined with `input`, `output`, and private ports

```plaintext
node Box begin
  input length Double
  input width Double
  input height Double
  output volume Double (length * width * height)
end
```

Listing 3.11: Node defined with `input` and `output` ports

\(^1\)BEDMAS defines the order mathematical operations should be evaluated and stands for brackets, exponents, division, multiplication, addition, and subtraction.
Box is only a node definition, similar to a class definition in object-oriented programming. To use it we must create an instance. To create an instance, we use the same syntax as we do when we declare an instance of a port as both ports and nodes are functions. A node defines a function in terms of ports and other nodes. The statement `b Box` creates an instance of Box called `b`.

```
1 b Box
2 b. length (100.0)
3 b. width (20.0)
4 b. height (7.0)
```

Listing 3.12: Declaring an instance of Box

To assign values to each of b’s input ports, the ports are accessed using dot notation. We refer to this sequence of dots and port identifiers as a path. Paths are resolved by the interpreter to the correct port during execution. Path resolution plays an important role in how Shiro is evaluated, which we will examine in more detail in Section 3.3.3.

It is also possible to instantiate and initialize a node’s ports in one line. Values can be set using named arguments or as a list of arguments. When a list of arguments is provided, values are assigned to the input ports in the order the ports are declared in the node definition.

```
1 b Box (width: 20.0, length: 100.0, height: 7.0) // notice width is provided before length
2 b Box (100.0, 20.0, 7.0)
3 // these lines are equivalent
```

Listing 3.13: Instantiating a node with arguments

Let’s finish our example by creating an instance of Box:

```
node Box begin
  input length Double
  input width Double
  input height Double
  output volume Double (length * width * height)
end

8 b Box (width: 20.0, length: 100.0, height: 7.0)
9 cost Double (b. volume * 12.0)
```

The value of cost is:

```
cost#<Double results:[168000.0]>
```

We create an instance of Box and use its volume to calculate the cost for the box. Now, imagine a scenario where you want to calculate the cost of digging a swimming pool and the excavation company charges you by the volume of dirt they dig. You want to calculate
the cost of pools of three different depths. You are certain of the pool’s length and width, as you are limited by the space in your backyard.

```
node Box begin
  input length Double(20.0)
  input width Double(36.5)
  input height Double
  output volume Double( length * width * height )
end

node Cost begin
  input volume Double
  output cost Double(volume * 12.0)
end

heights List(3.0 , 10.0 , 15.0)

volumes Map(heights , Box , @height , @volume)

costs Map(volumes , Cost , @volume , @cost)
```

Listing 3.14: Calculating the cost of Boxes of different volumes

We modify our Box definition by setting values for `length` and `width` when we declare them. We call the values given to a node’s ports at declaration *default values*.

Next, create a list of input values called `heights`. We use `Map` to compute the new list. `Map` is a function provided by the standard library that takes four arguments. The first argument is the list to process. The second is the reference to the node to use in the computation. `Box` is a *literal reference* and tells `Map` to create an instance of `Box` and, for each element in the list, use it compute the new value for the element. The next two arguments(`@height` and `@volume`) are *literal selectors* and tell `Map` which input of `Box` to assign the list value. Here the list element is being assigned to the port `height`. The value of the output `volume` is stored in the list after evaluation. This approach allows the programmer to control which value of the node is used as the output if the node has more than one output. `Map` computes the value of the node named by the literal reference for each value in the list. The value of `cost` after running the program is a list with the transformed values:

```
costs#<Map results:[[2190.0, 7300.0, 10950.0]]>
```

I have described how to create a simple parametric system. With the introduction of `Map`, you have seen how Shiro extends a traditional parametric system to support functions passed as arguments, a feature common in many programming languages such as Ruby and Javascript.

With this foundation, we can explore what makes Shiro different from other parametric systems. All of the examples we have worked through so far have had one thing in common;
they specify a single solution. Even though the programs may have multiple outputs, or
lists as results, they represent the result of a single set of parameters and connections, a
single state. Shiro allows you to represent multiple sets of parameters and connections, or
states in the same program.

Recall in our previous example how we said you could only use one footprint for your
pool. Let’s lift this constraint and explore several different lengths and widths for the pool.
Instead, this time we’ll constrain the depth of the pool to 10.0, deep enough for diving. We
start with the definitions of Box and Cost:

```plaintext
1 node Box begin
2   input length Double
3   input width Double
4   input height Double
5   output volume Double(length * width * height)
6 end

7 node Cost begin
8   input volume Double
9   output cost Double(volume * 12.0)
10 end
```

**Listing 3.15: Box and Cost definitions**

We will use Box to calculate the volume of the pool and Cost to determine the cost of
the excavation. Because the layout of the pool is what we are exploring, let’s use a node to
encapsulate the layout of a pool.

```plaintext
1 node Layout begin
2   input width Double
3   input length Double
4 end

5 node Box begin
6   input length Double
7   input width Double
8   input height Double
9   output volume Double(length * width * height)
10 end

11 node Cost begin
12   input volume Double
13   output cost Double(volume * 12.0)
14 end
```

**Listing 3.16: Define layout node**

Next, we’ll need a way to represent the pool layouts we want to explore. We could
represent the layouts with a list as we did with the pool depths earlier; however, this time
we want our program to represent the calculations for a single pool, not a collection of pools. To do this we need to add a node with options, also referred to as a subjunctive node, where the options are called subjuncts. The term subjunctive node is derived from Aran Lunzer’s [27] concept of a subjunctive interface.
1 node Layout begin
2   input width Double
3   input length Double
4 end

5 node Pool begin
6   option singleLane Layout(width: 10.0, length: 30.0)
7   option basin Layout(width: 20.0, length: 20.0)
8   option olympic Layout(width: 25.0, length: 50.0)
9 end

10 node Box begin
11   input length Double
12   input width Double
13   input height Double
14   output volume Double(length * width * height)
15 end

16 node Cost begin
17   input volume Double
18   output cost Double(volume * 12.0)
19 end

Listing 3.17: Add Pool definition

We define the node Pool to represent the pool in our program. What is a pool? It’s a layout. When we define options, we are saying to the interpreter, “I have an object called ‘Pool’ and it can have one of three states, each a different layout.” This allows us to write our program with respect to a single pool and still make it easy to determine which layout we want to calculate.

We instantiate a Layout node for each pool configuration. Instantiating a node inside of a node creates a nested node. Pool[basin] sets the default option for Pool as the option basin. Defining a default option informs the interpreter which option to choose if none is specified in a state declaration.

In previous examples, we use the default graph, the graph the interpreter uses when no graph statement wraps the commands to build a graph. In Listing 3.18, we use an explicit graph statement graph poolCosts begin ... end to make it easier to refer to the graph in the state statements.

We define three states, one for each pool layout. We tell each state which graph to use and set the active option for each node with options using the syntax <port-name>[<name>]. When the state is evaluated, the paths pool.active.length and pool.active.width are resolved to the corresponding port in the active option. Altogether, we have a program to compute the cost of a single pool layout that specifies three possible pool designs.
Listing 3.18: Define graph and states

The cost, or the value of \texttt{c.cost} in state \texttt{singleLane} is

\texttt{c.cost\#<Double results:[36000.0]>}

The cost, or the value of \texttt{c.cost} in state \texttt{basin} is
The cost, or the value of `c.cost` in state `olympic` is

```
c.cost#<Double results:[48000.0]>
```

In this section, I started from a basic program with only ports and built upon it to provide an overview of the main language features of Shiro and those that distinguish it, namely subjunctive nodes and state definitions. With this introduction, I’ve laid the foundation for the in-depth discussion of Shiro syntax and semantics that follows.

### 3.2 Language Definition

Shiro is a declarative, functional, dataflow language and here I present the syntax and semantics of the language. Shiro is an extension of the parametric language described in Elements of Parametric Design [56].

A function is the underlying representation in Shiro. The function takes a tuple with named elements as its argument and returns a tuple with named elements. The name of the function defines the type of the object. Formally, the function \( f \) is defined as follows:

Let \( I = \langle \text{name}_i : \text{value}_i, \text{name}_{i+1} : \text{value}_{i+1}, \ldots, \text{name}_m : \text{value}_m \rangle \) be the tuple with named elements of input values and let \( O = \langle \text{name}_j : \text{value}_j, \text{name}_{j+1} : \text{value}_{j+1}, \ldots, \text{name}_n : \text{value}_n \rangle \) be the tuple with named elements of output values, where \( i \) is the index of the value and \( n \) is the number of elements. When no name is given to the element, the name is left off. For example a tuple with one, unnamed element, the notation is \( \langle \text{value}_0 \rangle \). As a concrete example, if the value is 0, then notation would read \( \langle 0 \rangle \). \( f \) is the mapping between the input tuple and the output tuple.

\[
f : I \rightarrow O
\]

Functions have defined tuples of inputs and outputs. The union of these two tuples defines tuple of readable values for the function. If \( R \) is the tuple of readable values, it is defined formally as:

\[
R = I \cup O
\]

Literals like doubles, integers, and strings are represented by identity functions. These are functions that return themselves and have no name. For example, the integer 0 is represented as:

\[
f_0 : \langle 0 \rangle \rightarrow \langle 0 \rangle
\]
Functions can be named. a Double(2.0) defines a double with the value of 2.0 named a. Functions can be combined to create expressions. These expressions form graphs of functions, so a Shiro program can be defined as the directed acyclic graph S:

\[ S(F, D) \]

where \( F \) is the set of functions and \( D \) is the set of dependencies between the functions.

The functions’ names are used to connect functions to one another and result in the dependency graph shown in Figure 3.1. a refers to the function Double(2.0). b refers to the function Double(3.0). sum depends on a and b. The arguments of a function are passed by reference for simplicity of the implementation, but continue to be pure functions as all functions provided by the runtime only read from the reference.

```
1 a Double (2.0)
2 b Double (3.0)
3 sum Add(a, b)
```

Listing 3.19: Explicitly defined expression

![Figure 3.1: Dependency graph for code in Listing 3.19](image)

3.2.1 Node Definitions

Functions are assembled into containers called nodes. Nodes are similar to classes in object oriented programming languages and their names must start with a capital letter. Shiro programmers use them to encapsulate a computation, or graph of functions. When a function is used in a node it is referred to as a port. For example, here is a node calculates the area of a square.

```
1 node AreaSquare
2 | input length Double
```

43
Instances of a node are created in the same way as instances of a function. Here I create and instance of \texttt{AreaSquare} named \texttt{a}.

\begin{verbatim}
  a AreaSquare
\end{verbatim}

In addition to being reusable graphs, nodes are also functions. The arguments of the function are determined by the set of input ports. Access to a port is controlled using an access modifier, which is either \texttt{input}, or \texttt{output}. The purpose of these access modifiers is to allow the programmer to determine the input and output interface of the node. Declaring a port as an \texttt{output} prevents a port from being modified outside of the node definition, as is the case with ports declared as \texttt{input}. Ports without an access modifier are only accessible inside the node definition. Shiro programmers must explicitly declare whether a port should be accessible from outside the node.

- \texttt{input} can be read from or written to outside the node
- \texttt{output} can only be read from outside the node

From the perspective of a node as a function, Listing 3.20 is defined as \( f_{\text{AreaSquare}} : (\text{length, width}) \rightarrow (\text{area}) \). The position of a port in the list of port definitions determines its position in the argument list and result tuple. When the instance of \texttt{AreaSquare} named \texttt{a} is created the following paths become usable.

\begin{verbatim}
  a . length  
  a . width   
  a . area    
\end{verbatim}

\begin{verbatim}
Listing 3.21: Paths created by an instance of Listing 3.20
\end{verbatim}

It is possible to set a node’s ports at the time the instance is created using the argument list, named arguments, or direct assignment:

Argument List

\begin{verbatim}
  a AreaSquare (2.455)
\end{verbatim}

Named Arguments

\begin{verbatim}
  a AreaSquare (length : 2.455)
\end{verbatim}

Direct assignment

\begin{verbatim}
  a AreaSquare
  a . length (2.455)
\end{verbatim}
3.2.2 Using Nodes

With nodes and graphs, computations like Listing 3.22 can be specified.

```plaintext
1 node AreaSquare
2   input length Double
3   input width Double
4   output area Multiply(length, width)
5 end

7 length Double(200.0)
8 a AreaSquare(length: length)
```

Listing 3.22: Using an instance of AreaSquare in a graph

Listing 3.22 uses the default graph, whose name is \(^\_\). When ports are connected outside of a node or a graph definition, the statements are defined inside the default graph. The default graph is a graph the runtime maintains to allow this style of input. A programmer might want to create multiple graphs to represent different approaches to a problem. To do so, each graph is given a name and provides the namespace for the computation. For example, in the following program, the fully qualified name of the port length is AreaCalculation.length and the input port width in a is AreaCalculation.a.width.

```plaintext
1 node AreaSquare
2   input length Double
3   input width Double
4   output area Multiply(length, width)
5 end

7 graph AreaCalculation
8   length Double(200.0)
9   a AreaSquare(length: length)
10 end
```

Listing 3.23: Define a graph

The node is defined outside the graph so its definition can be reused in any graph. It is a syntax error to define a node inside a graph definition. The port length is instantiated in the graph AreaCalculation and is passed to the length port of AreaSquare.

Some readers will recognize the similarity between graphs and nodes. Both are named graphs of ports. The distinction between graphs and nodes is how they are intended to be used. A node is meant as a construct to enable encapsulation and code reuse and is intended to represent a subgraph. A graph represents one computation and node definitions represent the available building blocks. The graph construct is meant to act as an entry point for computation and provides the root scope for paths defined within. A graph is analogous to the main method in languages like C++ and Java.
The definition of a node creates a namespace for the ports it contains. Node definitions may be nested to provide more specific namespaces.

```
node A
  input alpha Double
  input beta Double
  node AA
    input Double alpha
  end
  output result Add(alpha, beta)
end
```

Listing 3.24: Nested node definition

When used in a graph definition, a nested node definition requires the programmer to write out the fully qualified path to create an instance.

```
a A
aa A.AA
```

Listing 3.25: Create instances of nested nodes in the default graph
A nested definition can be used within the nested node:

```plaintext
node A
  input alpha Double
  input beta Double
node AA
  input alpha Double
end

aa AA(1.1)
output result Add(alpha + aa.alpha, beta)
end
```

Listing 3.26: Create an instance of a nested node inside of its containing node

Though a node definition can be nested inside another, it is an isolated definition within the namespace. The nested definition does not have access to any of the parent definition’s ports. It would be a syntax error to write:

```plaintext
node A
  input alpha Double
  input beta Double
node AA
  input alpha Double
end

aa AA(beta)
output result Add(alpha + aa.alpha, beta)
end
```

Listing 3.27: Syntax error in a nested node definition

The `beta` is not defined in the scope of `AA` and none of the ports defined in `A` are accessible in the nested definition. An instance *must* be created and the ports connected in order `AA.alpha` to have access to `A.beta`. The need to create an instance is because a node definition represents an unrealized subgraph. It is simply a definition with the scope of another, a namespaced definition. Because the definition is not realized, there is no way for the ports in the parent definition to be connected with those of the nested node. They don’t exist. Once the node instantiated, the runtime is able to resolve the paths to connect the ports.

```plaintext
node A
  input alpha Double
  input beta Double
node AA
  input alpha Double
end

aa AA(beta)
```
Paths

During the graph building phase of evaluation (Described in Section 3.3.3), paths are resolved to values. So far I have shown the way to connect ports is to pass the name of the referenced port as an argument. In Shiro, these identifiers are called paths and what I have shown so far is the simplest form of a path. Recall that a function can return multiple values. A path is used to refer to those values. To make this description concrete, I will reference this node definition:

```
node MagicVolume begin
  input length Double
  input width Double
  input height Double
  factor Double (2.2)
  output vol Double (length \cdot width \cdot height \cdot factor)
end

mv MagicVolume (30.0, 40.0, 50.0)
```

Listing 3.29: Define a node to calculate volume with a magic factor

The simplest type of path in Shiro is to use dot notation to reference a port, similar to what is used in other languages like C++ and Java to access the properties of an object. `mv.length` resolves to the `length` port in the instance of `MagicVolume` named `mv`. For convenience, this statement accesses the first item in the result tuple of the `Double`, the output port `vol`. The elements of the result tuple can also be accessed with an index or with a key. Each function provides access to tuples containing its inputs and the outputs (results). This allows you to reference a value passed into a function directly from the inputs tuple.

```
mv.length.outputs[0]
mv.length.outputs[vol]
```

Recall that nodes are also functions so this syntax applies to them as well. The port `length` can be also be accessed with the `inputs` keyword like `mv.inputs[0]` or `mv.inputs[length]`. Similarly, the `outputs` keyword can be used like `mv.inputs[0]` or `mv.outputs[vol]`.

Listing 3.29 provides a number of equivalent paths to read from `length`:

```
mv.length
mv.length.outputs[0]
mv.inputs[0]
```
Each of these paths resolves to first output value of the port `length`. Use of the input and output accessors can also be done inside a node definition.

```plaintext
node MagicVolume begin
  input length Double
  input width Double
  input height Double
  factor Double (2.2)
end
```

The internal references to `length`, `width`, and `height` can be replaced with the inputs accessors.

Some of these paths can be used to set `length`:

```plaintext
mv. length (100.0)
mv. inputs[0](100.0)
mv. inputs[length](100.0)
```

Paths are resolved by looking for a function in the existing scope that matches the first segment in the path and searching down the scope tree. If function is not found, the system throws a runtime error. Here I define two nodes, A and B each with a single property. Then I create instances of the two nodes and try to access `a.x`.

```plaintext
node A begin
  input x Double (13.2)
end

node B begin
  input y Double (183.2)
end

a A
b B
a.x
```

Path selectors and references are two other atoms in Shiro. A `path selector` is a path prefixed with the sigil `@`. A path selector is a literal path and is not resolved to a port’s value by the runtime. It is used to specify on which port a function should operate. This is
Figure 3.2: Path resolution of b.y in Listing 3.32 starts in the default graph named ^1. The first segment of the path matches the node instance a (1) in the scope of ^1. Search continues for the rest of the path in scope of a. x is the next path segment and matches the port x in a (2). Since x is the last path segment, a reference to the port is returned, completing the search.

similar to the selector construct in Objective-C, which allows a programmer to dynamically call a method on an object\(^2\). It provides the same function as Ruby’s send method, which dynamically calls a method on an object.

For example, it is used to select an input and output port in Map.

```
node DoubleIt begin
  input v Double
  output newValue Double (v * 2)
end

l List (1.0, 2.0, 3.0)
doubled Map(l, DoubleIt, @v, @newValue)
```

Listing 3.33: Passing a path selector to a function

doubled#<Map results:[[2.0, 4.0, 6.0]]>

Map is a function that takes as its arguments a list of objects to transform, a type literal, an input selector, and an output selector and returns a list as its first argument. The list’s elements are the function indicated by the type literal being applied to each element of the input list. When the function is applied, the value from the initial list is passed to the port identified by the path selector in argument two. The value of the output identified by argument three is returned and add to the list returned in the first output of Map. The path selectors allows the programmer to choose to which argument the list element is passed and from which output value to retrieve the value, a necessity since functions can have multiple return values.

Thus, passing @v to Map tells the runtime which port of DoubleIt to assign the list
element to as it iterates through the list. Passing @newValue tells Map which value of
DoubleIt to use as the output and store in the new list.

References provide a way to access a function instead of its outputs. A literal reference
is the name of node or function and must start with capital letter. In Listing 3.33, DoubleIt
is a literal reference and is passed to Map as the function to apply to each element in the
list l.

When paths are resolved they return the value of the first element in the result tuple. To
retrieve a reference to the function to which the path refers, rather than the output value,
we use the path reference operator ~. For example, ~a resolves to the function named a.
Paths prefixed with the operator are referred to as reference paths. The reference operator
essentially does the opposite of the de-reference operator in C/C++, where it returns the
value at the address held by the pointer. In Shiro, it returns a reference to the function.
Listing 3.33 can be rewritten to use a reference path instead of a literal reference.

```
1 node DoubleIt begin
2   input v Double
3   output newValue Double (v * 2)
4 end
5
6 di DoubleIt
7 l List(1.0, 2.0, 3.0)
8 doubled Map(1, ~di, @v, @newValue)
```

Listing 3.34: Passing a reference path to a function

In this case, an instance of DoubleIt is created named di and a reference path is used
to pass it to Map.

### 3.2.3 Options and States

To make it possible to represent alternatives in a parametric system, we add the concept
of an optional port to the language. With the keyword option, a port declaration can be
marked as an alternative value or computation.

```
1 node Numbers [one]
2   option input one Double(110.0)
3   option input two Double(100.0)
4 end
```

We use Numbers[<option >] to define a default option for the node. To allow a pro-
grammer to retrieve the active option from the node, we use the keyword active. It allows
a programmer to refer to the active port in the current states and allows me to write a
program that is generic with respect to its state.
The keyword `active` exists to provide a way to reference an active object at runtime. Because each option is named, it is possible to reference the port directly by name. However, as will be seen when I introduce state definitions, referencing the port by name causes the graph to only compute when the mentioned port is the active option. The programmer needs a way to generically refer to the active option of a node, otherwise a single graph cannot be used to represent multiple states. The `active` keyword allows a graph to be reused by multiple state definitions where the only difference between states is the options that are set active. The `active` keyword allows Shiro code to refer in general to the currently active option in a node.

The final construct in Shiro is the state definition. A state definition represents, given a graph and set of options to activate, a single state in the solution space. Because solutions spaces are often large, state definitions provide a way to specify single states to reduce the number of states the runtime must evaluate.

```
state Sum begin
  graph math // define the graph to evaluate
  n[one] // select option for the node
end
```

Each state definition must be given a unique name. It defines a graph and set of node → option mappings. To handle the simple parametric case, where there are no options and no states defined, the runtime provides a default state called `$`. When there are options defined in the node, but no states given, the runtime generates states for all combinations of options in the graph. For example, defining

```
node Character
  option output jeff String('Jeff')
  option output joey String('Joey')
  option output bob String('Bob')
end
	node EmployeeBadge
  input name String
  ...
end
```
Listing 3.36: Define three options

causes the runtime to generate three state definitions. The names are of the states are
determined by the runtime and default to state#.

```ruby
1 state state1
2  graph ^
3  c[jeff]
4 end
5
6 state state2
7  graph ^
8  c[joey]
9 end
10
11 state state3
12  graph ^
13  c[bob]
14 end

Listing 3.37: The state definitions generated by the runtime for Listing 3.36

Shiro programmers should beware having the runtime automatically generate all states
can quickly generate hundreds of states because the number of states is combinatorial to
the number of options specified by the nodes in the graph.

3.3 Evaluation

Because Shiro is declarative and the order of a program’s code does not matter, the runtime
needs a number of passes to evaluate a program.

3.3.1 Dependency Resolution

To begin evaluation, the runtime lexes and parses each file referenced by the import state-
ment. The parse trees are cached for later passes. The runtime creates a dependency graph
of imported files and processes them in topological order.

3.3.2 Collect Definitions

Once the dependencies are parsed, the node definitions are collected and stored as parse
trees. The parse trees are processed later by the graph building pass to create instances of
nodes.
3.3.3 Realize States

Each graph defined in the Shiro file being run is realized by instantiating the nodes declared within it. To instantiate a node, a node’s parse tree is walked twice. The first pass creates instances of any nodes within it. After the nodes are instantiated, the port assignment statements are processed to connect the nodes together.

To evaluate the graphs, a state must exist to specify which options are to be activated. If the program does not specify any states, the runtime generates states for each combination of options. If there are no options specified in any of the nodes in the graph, a single state without an option selection block is generated.

After the states are generated the runtime begins evaluating states. To evaluate a state, the active options specified in the state definition are activated and the graph’s dependencies are resolved. At this time paths are resolved to ports and a direct acyclic graph, or dependency graph of the graph’s ports is created. Once the dependency graph is created, it is topologically sorted and each port is evaluated. This process of option activation, dependency resolution, and dependency graph evaluation is completed for each state referencing the graph. If a state references a different graph, the graph is instantiated and evaluated as described.

3.3.4 Recursion

Shiro supports recursion by allowing a node to be passed as a reference to its definition. For example, here is the classic example of recursion—calculating a fibonacci number.

```shiro
node Fibonacci begin
  input n Integer

  zeroFilter ConditionalReturn(n == 0, 0, 0)
  oneFilter ConditionalReturn(n == 1, 1, 0)
  nMinus1 ConditionalReturnNode(n > 1, n - 1, Fibonacci, 0)
  nMinus2 ConditionalReturnNode(n > 1, n - 2, Fibonacci, 0)

  // Sums the arguments passed
  values List(zeroFilter, oneFilter, nMinus1, nMinus2)
  output sum Sum(values)
end

graph fib5 begin
  fib Fibonacci(n: 7)
  // fib.sum is 13
end
```

Listing 3.38: Naïve Calculation of a Fibonacci number

Listing 3.38 shows how a programmer might implement a node to calculate a Fibonacci number. `ConditionalReturn` is a function that returns the value in argument one if argu-
ment zero is true. If argument zero is false, the last value is returned. `ConditionalReturnNode` provides similar functionality, but allows a node reference to be passed that determines the value returned if argument zero is true. The `Sum` function is used to sum the values provided by the conditional return functions.

Listing 3.38 demonstrates a naïve implementation to calculate a Fibonacci number. A more efficient approach is to use tail-recursion.

```plaintext
node Fibonacci begin
  input n Integer
  input a Integer
  input b Integer

  zeroFilter ConditionalReturn(n == 0, a, 0)
  oneFilter ConditionalReturn(n == 1, b, 0)
  // if n is greater than 1, recursively calculate the Fibonacci number,
  // otherwise return 0
  onePlusFilter ConditionalReturnValues(n > 1, ~Fibonacci(n - 1, b, a + b), 0)

  // Sums the arguments passed
  values List(zeroFilter, oneFilter, onePlusFilter)
  output sum Sum(values)
end

graph fib7 begin
  fib Fibonacci(7, 0, 1)
  // fib.sum is 13
end
```

Listing 3.39: Use tail-recursion to calculate a Fibonacci number

n is the Fibonacci number to calculate. a and b are the two base cases, where \( \text{fib}(0) \) is 0 and \( \text{fib}(1) \) is one. Instead of calculating the needed Fibonacci number for each sum, the previous two numbers are passed into the node.

**Evaluating Recursive Node Definitions**

Recursive node definitions are evaluated in a similar way to other functions. The passed reference is used to create an instance of the node local to the function. The node instance acts as a detached graph that is evaluated whenever the containing function is evaluated. See Figure 3.3 for an illustration of the graphs created by Listing 3.39.

When a function is passed as an anonymous reference, its arguments are resolved in the scope of its instantiation. This is how line 10 of Listing 3.39 passes the arguments of Fibonacci are passed into `ConditionalReturnValues`. 
3.4 Idioms for Describing Alternatives

In the process of writing Shiro programs during development of the language, the idioms of Property Variation and Object Variation emerged. Property Variation is a pattern of programming where a node is created to hold the alternative values for a property.

```
1 node InterestRates[low] begin
2   option output low Double(0.01)
3   option output medium Double(0.05)
4   option output high Double(0.19)
5 end
6
7 node Interest
8   input principal Double
9   input rate Double
10  input time Double
11  output newValue Double(principal * rate * time)
12 end
13
14 rates InterestRates
15 i Interest
16 i.principal(1000.0)
17 i.rate(rates.active)
18 i.time(1)
```

Listing 3.40: Property Variation
In Listing 3.40, the node *InterestRates* is used to represent the set of interest values to be used in the interest calculation.

With Object Variation, the programmer creates an instance of the node to be varied for each alternative he wants to create. To allow the options to be selected via state definitions, the options are placed in a node. Doing so allows the node to be used in graphs without depending on the selected option. The options do not need to be of the same type. They only need to have a port with the same name, so the path remains valid irrespective of which option is selected. The result is the ability to access a common “interface” in a manner reminiscent of polymorphism in object-orient programming.

```plaintext
1 node Interest
2  input rate Double
3  input principal Double(20000.0)
4  input time Double(1.0)
5  output newValue Double(principal * rate * time)
6 end

7
8 node Investment [lowRate]
9  option lowRate Interest(rate: 0.01)
10 option mediumRate Interest(rate: 0.05)
11 option highRate Interest(rate: 0.19)
12 end

13 investment Investment
14 investment . active . newValue
```

Listing 3.41: Object Variation

In Listing 3.41, three instances of *Interest* are created to represent the three investment configurations. They are wrapped in *Investment* to enable their ports to be accessed in the graph with the path *investment.active.newValue*.

### 3.5 Shiro Playground

Being able to run Shiro code is an important part of designing a language. Inspired by editors like the Processing editor and XCode’s new Playground, I implemented a basic editor prototype for Shiro called *Shiro Playground*. *Shiro Playground* provides a text editor for editing Shiro code and two visualization tools for viewing results: the Canvas (which provides superimposed views) (See Figures 3.4 and 3.5) for viewing alternatives singly and superimposed, and the LightTable (which provides juxtaposed views) (See Figure 3.6) for arranging multiple alternatives in grids or user-defined spatial layouts. To use the prototype, a designer can type code into the editor, or load code from a file. Once opened, the code can be run by pressing the Run button.
Figure 3.4: A single alternative can be shown in the Canvas by selecting a single state. (a), (b), (c) and (d) each show one such state.

The result of the code is shown in the Gallery tab. The designer selects, from the list of alternatives on the right, the alternative they want to view. Selecting multiple alternatives causes the alternatives to be superimposed.

Figure 3.5: Selecting multiple states in the Canvas superimposes their views into a single view. This figure shows states (a), (b) and (c) from Figure 3.4.

On the LightTable tab, the designer can choose between a grid view and a freeform view. The freeform view allows the designer to organize the alternatives as he might if he were to print out all the images and tack them to a bulletin board. Because I believe the best design tools are mixed-initiative [19], where computers tools augment the creativity and work of the human designer, I believe that designers will want to choose a layout that meets their needs. I support this need by providing two of many possible layout tools. The
grid provides an automatic layout of the designs and the freeform layout allows the designer the flexibility to organize the alternatives as he desires.

3.6 Implications for History and Undo

In the previous sections, I describe how Shiro can be used to simultaneously represent multiple states of an artifact. However, the work of people to define and to refine an artifact also happens over time. Sometimes, in the process of solving a problem a user wants to recall a state of the artifact from a prior point in the day; from history. History refers to a time ordered sequence of operations stored by a tool. The most common implementation of history is Undo/Redo, where a tool records a stack of commands and requires each command to be reversible [37]. If a command is Add Rectangle, it can be reversed to remove the rectangle. Undo is used to move backwards through the stack and Redo is used to move forward. To move to a previous state of the artifact, the reverse of each previous command must be executed. To access a previous state of an artifact, a tool needs to provide a mechanism for recording the changes made over time, and while this work deals primarily with the simultaneous representation of alternatives in a parametric system, I also considered the problem of representing alternatives over time as I designed Shiro.
3.6.1 Represent Time as States

Shiro was designed for the purpose of simultaneously representing multiple alternatives in a parametric system. With the ability to represent multiple states, one might wonder if it is possible to represent history with just the constructs available in Shiro. The answer is “almost.” To support history using states and subjunctive nodes requires a change to the runtime. The runtime must know which state to evaluate for a moment in time, \( t_n \).

To create a record of the history of an artifact, a representation must be able to represent changes to its elements over time. In the context of Shiro, the question is whether states can be used to represent changes to node definitions, input data, and graph structure. By using subjunctive nodes and state definitions, I can improvise a record of changes to a property over time. Consider the following code:

```plaintext
1 node B begin
2  option input t0 Double (3.455)
3  option input t1 Double (333.4)
4 end
```

Listing 3.42: A subjunctive node representing values at \( t_0 \) and \( t_1 \)

We can represent the changes to a property by defining a node \( B \) to encapsulate its different values. We give each option (\( t_0 \) and \( t_1 \) in this case) a name to represent when the option is to be selected. I write state definitions to represent those moments in time.

```plaintext
1 node B begin
2  option input t0 Double (3.455)
3  option input t1 Double (333.4)
4 end
5
6 node A begin
7  input x Double
8 end
9
10 a A(b-active)
11 b B
12
13 state t0 begin
14  graph ^
```

Figure 3.8: History
Listing 3.43: Add states to Listing 3.42 to represent $t_0$ and $t_1$

In Listing 3.43, I create a simple program that passes the current value of $B$ to an instance of $A$. This approach will only work if modifications are made to the Shiro runtime. To execute the correct state for the correct moment in time, the runtime needs to be modified to recognize a naming convention that will identify the state to evaluate that represents the point in time. To evaluate the parametric system at $t_0$, the runtime must evaluate the state $t_0$ so that the option $t_0$ is selected for $B$.

To retain the ability to simultaneously represent multiple states of an artifact, the reason why Shiro was designed in the first place, further modification to the runtime is needed. The runtime needs to be modified to allow multiple states to be associated with a point in time. One approach might be to use a naming convention for states that allows the runtime to identify the states to evaluate at a given point in time. For example:

```
1 node B begin
2   option input t0 Double(3.455)
3   option input t1 Double(333.4)
4 end

5 node C begin
6   option input small Double(13.2)
7   option input big Double(26.4)
8 end

11 node A begin
12   input x Double
13 end

15 c C
16 b B
17 a A(b.active * c.active)

19 state t0_small begin
20   graph ~
21   b[t0]
22   c[small]
23 end

25 state t0_big begin
```
States \(t_0_{\text{\text{big}}}\) and \(t_0_{\text{\text{small}}}\) are used to represent the multiple states of \(C\) in \(t_0\). The same pattern is applied to \(t_1\). With this workaround, the remaining pieces are to track changes to graph and node definitions.
In Listing 3.45, I represent three time steps. In $t_0$, the nodes $A$, $B$, and $C$ are defined and the graph $t_0$ is defined. Because $C$ has two options, two states are created so the runtime will evaluate each of the alternatives. In $t_1$, the value of $B$ is changed to 333.4 and two states are created to represent the two alternatives at that point in time. In $t_2$, the graph is modified to use addition instead of multiplication, so a separate graph definition is needed to define the states $t2\_big$ and $t2\_small$.

The changes to a node over time can also be represented in a similar manner. In $t_3$, I want to change the definition of $C\_small$. 13.2 is too small. 15.2 would be better. To make this change, I factor out the definition of $small$ into its own node.
Because I changed the definition of $C$, I need to update the existing states to account for the nested option in $C$. 

```
node B begin
  option input t0 Double(3.455)
  option input t1 Double(333.4)
end
	node C_small begin
  option input t0 Double(13.2)
  option input t3 Double(15.2)
end

node C begin
  option output small C_small
  option input big Double(26.4)
end
```
b B
a A(b.active + c.active)
end

state t0_small begin
graph t0
b[t0]
c[small]
small[t0]
end
end

state t0_big begin
graph t0
b[t0]
c[big]
end

state t1_small begin
graph t0
b[t1]
c[small]
small[t0]
end
end

state t1_big begin
graph t0
b[t1]
c[big]
end

state t2_small begin
graph t2
b[t2]
c[small]
small[t0]
end
end

state t2_big begin
graph t2
b[t2]
c[big]
end

state t3_small begin
graph t2
3.6.2 Tradeoffs to Representing Time as States

If the runtime is modified so that it can identify the states to evaluate for each moment in time, it’s possible to represent history using Shiro’s subjunctive node, option, and graph constructs; however, this might not be the best approach for representing the changes to a declarative representation of an artifact as it mixes the concerns of representing the artifact and representing changes over time. Shiro is meant to represent an artifact with multiple alternatives states at a given point in time. The use of subjunctive nodes and state definitions introduces a lot of complexity to a program. It is no longer a concise definition of an artifact as it introduces much duplicate code. For example, the graph definitions in Listing 3.48 differ only in one line (Compare lines 23 and 29), but two declarations are needed. Similarly, this approach requires a state definition for each time/alternative combination. For artifacts with many alternative states, many state definitions will be required. Rather, than having to redefine a state for each time step, a number which will grow with each addition step the user takes, time can be factored out of the representation.

By factoring time out of the representation of the artifact, a Shiro program is simplified back to its original intent—to represent a snapshot of an artifact with multiple states. If we factor out time, we need a construct to represent time. A transaction is such a construct. A transaction is an object that represents the transition between two definitions of an artifact and contains a set of commands to apply to the runtime. In a transactional approach to history, the artifact definition only needs to represent the current state of the artifact (a state which might contain many alternatives) and the transaction represents the changes to the artifact’s definition.

Because Shiro is a declarative language, a transactional system is straightforward to implement. Each change to the parametric system can be represented by an extended set of Shiro statements since Shiro code represents the relationships between nodes and ports and the runtime determines the evaluation order. The whole system does not need to be
redefined, or reloaded. Changes can be made by executing statements in the context of the current runtime. All that remains is to re-evaluate the parametric system.

Motivation to add the ability to execute partial model definitions is high as a scripting interface, which will require incremental execution of statements, is a common requirement for parametric design tools. Thus adding incremental execution to the runtime does not add an extra requirement to the implementation. A tool should allow a designer to make incremental changes to the model by writing Shiro statements. For example, if a programmer wanted to add a node definition and use it in the graph, the following can be done:

```shiro
node InterestRates [low] begin
  option output low Double (0.01)
  option output medium Double (0.05)
  option output high Double (0.19)
end

node Interest
  input principal Double
  input rate Double
  input time Double
  output newValue Double (principal * rate * time)
end

rates InterestRates
i Interest
i.principal (1000.0)
i.rate (rates.active)
i.time (1)
```

Listing 3.49: Current definition of artifact

A transaction containing the following statement can be applied to redefine the principal value:

```shiro
i.principal (5000.0)
```

Listing 3.50: Transactions Statements

A small vocabulary of additional transaction-specific statements are needed to represent operations such as delete and rename nodes and ports as these operations are not possible to do by redeclaring a statement as I did in Listing 3.49. This similar to SQL, which is a composite of three languages: the data definition language, the data modification language, and the query language. In this vein, a Shiro definition acts as the data definition language and the transaction language would act as the data modification language.

With the ability to represent both a complete and partial state of the runtime it becomes straightforward to implement linear history and undo. To jump to a point in time, the Shiro code for that point in time can executed bringing the tool to that state. To step forward and backward through time, the transactions can be applied or reversed, accordingly. Further
optimizations can be made by implementing a cache to store the results of ports. Given that ports are functions, a port will have one and only one result for a set of inputs. Movement through the timeline will require the runtime to re-evaluate each state and having a cache will reduce the amount of computation each re-evaluation causes. The more complicated the computation required to execute a port, the more the runtime will benefit from the cache. That Shiro is a declarative language makes it straightforward to implement history and undo in design tools because of the low cost to store and recover states.

3.6.3 Representing Time as Transactions

For the sake of keeping the scope of this work manageable, I did not implement support for history in the Shiro runtime. Instead, to demonstrate these ideas I built a prototype drawing application called QDraw that executes a simple, declarative, Shiro-like language to explore the integration of a transactional, script-driven interface with a direct manipulation interface.

![Figure 3.9: QDraw canvas](image)
**QDraw** allows users to create simple drawings using the language or a direct manipulation interface. Two statements are possible in **QDraw**, one to define a rectangle and one to define an ellipse:

```plaintext
1 rect0 rect(100.0, 100.0, 100.0, 100.0, "GREEN")
2 ellipse0 ellipse(10.0, 10.0, 10.0, 10.0, "GREEN")
```

**Listing 3.51: QDraw statements**

The first statement defines a rectangle named `rect0` and pass as its arguments (in order left to right) the x position, y position, width, height, and the color of the shape. Three color values are possible: "GREEN", "BLUE", and "YELLOW". Similarly, the second statement defines an ellipse named `ellipse0` and its arguments are the x position, y position, x diameter, y diameter, and the color of the shape.

Statements are entered into the text area on the Editor tab. If the “Treat as REPL” checkbox is checked, the statements are evaluated in the context of the current state when “Run” is pressed. If the “Treat as REPL” checkbox is not checked, statements representing the current state of the program are appended to the text area. When “Run” is pressed, the system executes the code in the editor from scratch.

![QDraw code editor](image)

**Figure 3.10: QDraw code editor**

The prototype also allows users to create rectangles and ellipses using their mouse (See Figure 3.9. To create a rectangle, a user clicks the “R” button and a rectangle is created at the origin of the canvas. Once the rectangle is created, a properties panel on the side is shown. The properties panel allows the user to adjust the properties of the rectangle. Changes are made by pressing “Update.” The “S” (Select) and “M” (Move) buttons allow the user to select a shape and move it.

As the user makes changes to the rectangle, the changes are captured as transactions. Each transaction contains a set of statements representing the changes, the *forward statements*, and set of statements representing their reverse, the *reverse statements*. The transaction language differs slightly from the state declaration because it needs to deal with cases
when a statement is applied for the first time. In this case, a `delete` statement is generated as the reverse statement. As transactions are generated, they are applied to the current state of the system to modify the drawing. Also captured in a transaction is the resulting state of the system as statements. The state declaration acts as a cache for the runtime state at the point in time where the transaction is created. As changes are made to the drawing, transactions are appended to a list and represent the history of the drawing.

```plaintext
1 transaction {
2   ellipse0 ellipse (10.0, 10.0, 10.0, 10.0, 'GREEN')
3
4   state {
5     rect0 rect (100.0, 100.0, 100.0, 100.0, 'GREEN')
6     ellipse0 ellipse (10.0, 10.0, 10.0, 10.0, 'GREEN')
7   }
8 }
```

Listing 3.52: QDraw transaction with the resulting state cache

A slider along the bottom of the window allows the user to `skate` through history by applying the forward or reverse statements. The code representing the result of a transaction is captured to make it easier to jump to a particular state. If a state is in the distant past or the distant future, I can save time by jumping to the state and executing it rather than executing a long series of transactions.

With QDraw I learned what was required to build a transactional history system on top of a declarative language. It requires support for incremental evaluation, and the addition of a data modification language.

### 3.6.4 Compare Representing Time as States and Transactions

A convention of state definitions or transactions can be used to represent the history of a multi-state graph. Both approaches require modification of the Shiro runtime as state definitions are only meant to represent the alternatives at a given point in time. Using state definitions to represent history has several drawbacks. It embeds history into the multi-state document model rather than treating it as a separate concern, which results in a complex and brittle representation of history. It is complex because both time and parallel states are represented in the file. Also, as evidenced by Listing 3.45, the code becomes quite difficult to follow. To determine code for a particular space of artifacts, the programmer has to keep track of what time step he’s at, which can be tricky some alternative selections persist between multiple time steps. For example:

```plaintext
1 state t1_big begin
2   graph t0
3     b [ t1 ]
4     c [ big ]
```

30
As a reader, the programmer has to know that the graph in $t_1$ is the same as $t_0$ and the only that changed is the option selected in C. And things only become more complex when you consider tracking changes to a node definition as I showed in Listing 3.48.

The representation is brittle because it depends on a naming scheme that a programmer will have to explicitly follow. If the programmer deviates, the runtime will not be able to determine what states to render for given time. Use of a naming convention also makes it more difficult for a programmer give nodes names that make sense with respect to the problem being solved. Instead, names have to be chosen to keep track of time and to represent the problem, adding cognitive overhead. It does have the advantage of storing both history and alternatives in a single representation at the cost of significantly increased complexity.

A transactional approach, in comparison, is much simpler. Though it requires the addition of a new construct, the transaction, it respects the original intent of Shiro—to provide a clear, compact representation of multiple states in a parametric system. With transactions, there is a clear separation between representing changes to a multi-state model and representing the model. Transactions and the base model declarations leverage a common linguistic foundation so that the programmer only has small amount to learn to author changes to the model. I argue that the cost of learning these constructs is much smaller than having to manage a mental model of the combine representation of time and multiple artifacts, a mental model that will change with each problem. Wherease, with transactions, once the programmer has learned the data modification language, they can apply the same concepts to each problem and handoff the bookkeeping to the Shiro runtime. This is ideal as the reason for creating Shiro is to reduce the difficulty of managing multiple states of an artifact.

A transactional representation makes it easier to build applications that convert user interactions into a record of history. With the state-based approach, the client application must perform complex refactoring operations each time a property is modified, or a graph structure is changed. With a transactional approach, the only thing that’s needed is the redefinition of the portion of the graph that’s changed. There is no need to manage structural changes or have to create extra nodes, or subjunctive nodes. If a property changes, the client application needs to inform the runtime to change a property to a new value. If a graph is changed, the runtime is informed that arguments of a particular part have changed. The Shiro program representing the alternative versions of the artifact remains consistent with the domain.

Both approaches make it possible to represent linear history, a time ordered sequence of changes to an artifact. Only a transactional approach is easily extended to support branch-
ing history. Rather than representing history as a list of transactions, which yields a linear history, a graph can be used. By using a graph transactions, it’s possible to represent the branching activity of problem solvers as they revisit previous solutions and modify them to explore other parts of the problem space. It should be noted, that with a graph representation of problem solver activity, the representation is no longer strictly chronologically ordered. The graph is a structural representation of how an artifact is transformed. To view a chronological record of the problem solving activity the branches of the graph must be flattened into time order, so that moments of backtracking are shown at the proper point in time.

For these reasons, the transactional approach to representing history seems best. It allows a clear, domain-specific representation of multiple alternatives of an artifact at the small cost of requiring the programmer to learn a new construct.

3.7 A Multi-state Data Model

Shiro provides a declarative, multi-state document model for describing parametric systems. Subjunctive nodes, containing options, make it possible to express both alternative values for properties and alternative computations. Shiro also allows graphs to contain recursive node definitions. The state construct allows specific alternatives of artifact to be specified and named. Because Shiro is declarative, it is easy to extend to support transactional history as I demonstrated with QDraw. The Shiro Playground demonstrates a basic editor and provides a reference implementation of a tool for displaying and manipulating the alternatives with Shiro. The Shiro runtime provides a platform on which researchers and tool implementers can build tools to support alternatives.
Chapter 4

Evaluating Shiro

The purpose of this work is to explore the question, “Is it possible to extend a parametric system to make it possible to represent multiple alternatives in a single system definition?” Having described the syntax and main concepts of Shiro, in this chapter I evaluate the design of Shiro.

In much of the previous work on the topic, summarized in Chapter 2, the focus of researchers has been on designing user interfaces to help people manage and explore alternatives. The research I have summarized covers a variety of user interface designs and this research has been evaluated using methods appropriate for user interfaces with the goal of determining if the affordances of the design effectively support an aspect of problem solving.

My work, however, breaks from this pattern. Instead of building a graphical tool, I designed and implemented a programming language. As I argue in Chapter 3, there is a lack in research on how to represent multiple alternatives in a manner that tools can use the representation. This purpose of this work is to fill this gap.

As a result, my method is to design and build a working system in order to bring into existence a system that allows problem solvers to represent alternatives in parametric systems and show how it can be used. I have implemented a Shiro interpreter and runtime. A prototype implementation is available on Github\(^1\). Shiro is a working proof-of-concept. It provides means by which a multi-state representation can be explored in the future. The object or phenomenon must first exist before it can be studied. Given large amount of labor that goes into building a language, my effort was focused on bringing multi-state parametric system into existence.

To show that Shiro is indeed a capable representation, I describe the steps to solve problems in the domains of design, finance, and data analysis. The purpose of these case studies is to demonstrate the utility of Shiro in a variety of different domains. Following the case studies, I discuss this work with respect to common research practice address the question, “How do I know that what I’ve built is any good?”

\(^1\)https://github.com/jrguenther/shiro
4.1 Designing a Logo

4.1.1 Problem Statement

In Chapter 1, I began by describing a fictional, yet realistic scenario of a designer tasked with designing a logo for EverClean, a natural cleaning products company. In this example, I demonstrate how to use Shiro to complete a similar task. I want the logo to look like the simple, retro, and geometric logos that have become popular in the last few years.

In this section, we’ll recreate the following logos created manually in a graphics tool in Shiro.

4.1.2 Solution

To begin, I create a graph and add an ellipse to act as the logo background. Shape nodes are found in file named “geom” provided by the runtime. Nodes provided by the runtime are considered part of the standard library. I use the include statement to import them.

By default the ellipse has a white fill and a black stroke. I position the center of the ellipse at (200.0, 200.0). I’ve also drawn a line across the middle of the circle for reference as I position elements later.

```cpp
#include "geom"

radius Double(200.0)
stroke ColorFromRGB(10, 10, 10, 1.0)
red ColorFromRGB(255, 10, 10, 1.0)
whiteFill ColorFromRGB(255, 255, 255, 1.0)
```
Next, I add a text box to display the company’s name. I’m just roughing in the shapes, so I won’t worry about styling anything yet. The default styling of the text is shown.

```groovy
include 'geom'
radius Double(200.0)
stroke ColorFromRGB(10, 10, 1.0)
red ColorFromRGB(255, 10, 1.0)
whiteFill ColorFromRGB(255, 255, 1.0)
```
Next, I add the lines above and below the the logotype as decorations.
With the basic shape of the logo roughed in, I start exploring colors for the typeface and background ellipse. To explore these properties, I create nodes to encapsulate the properties I want to explore. I need to define a node to represent the color of the type when the background changes. I also give the graph a name so it can be referenced more easily later.
Figure 4.4: Draw lines above and below the logotype.

```latex

\begin{verbatim}
graph everGreenLogo begin
  radius Double(200.0)
  stroke ColorFromRGB(10, 10, 10, 1.0)
  red ColorFromRGB(255, 10, 10, 1.0)
  whiteFill ColorFromRGB(255, 255, 255, 1.0)
  fontSize Double(56.0)
  bg Ellipse(centerX: radius, centerY: radius, fill: whiteFill)
  bg.stroke(stroke)
  bg.radiusX(radius)
  bg.radiusY(radius)
  mid Line(startX: 0.0, startY: radius)
  mid.stroke(red)
  mid.endX(2 * radius)
  mid.endY(radius)
  logotype Text(text: 'EverClean', font: 'Times New Roman')
  logotype.fill(stroke)
  logotype.size(fontSize)
  logotype.originX(radius - 125)
  logotype.originY(radius + 20.0)
\end{verbatim}

```
Now that the drawing is a named graph, we add nodes to represent the options. I start with the typeface node. It’s easy to add more variation after I’m confident the graph is setup properly.

```plaintext
include 'geom'

node Typefaces begin
    option output pacifico String("Pacifi co")
    option output baskerville String("Baskerville")
end

graph everGreenLogo begin
    typeface Typefaces

    radius Double(200.0)
    stroke ColorFromRGB(10, 10, 10, 1.0)
    whiteFill ColorFromRGB(255, 255, 255, 1.0)
    fontSize Double(56.0)

    bg Ellipse(centerX: radius, centerY: radius, fill: whiteFill)
    bg.stroke(stroke)
    bg.radiusX(radius)
    bg.radiusY(radius)

    logotype Text(text: "EverClean", font: typeface.active)
    logotype.fill(stroke)
    logotype.size(fontSize)
    logotype.originX(radius - 125)
    logotype.originY(radius + 20.0)
```

Listing 4.4: Refactor drawing into named graph
Listing 4.5: Add optional typeface values

Figure 4.5: Draw lines above and below the logotype.

After defining the nodes I instantiate the option nodes and use the `active` keyword to pass their values to the `Ellipse` and `Text` nodes. Because I want to control which options I see in the gallery, I specified the states I wanted to see. Recall, the background color and the text color need to change in tandem.
include 'geom'

node Typefaces begin
  option output pacifico String('Pacifico')
  option output baskerville String('Baskerville')
end

node BackgroundColors [white] begin
  option output black ColorFromRGB(0, 0, 0, 1.0)
  option output white ColorFromRGB(255, 255, 255, 1.0)
end

node TypeColors [grey] begin
  option output grey ColorFromRGB(10, 10, 10, 1.0)
  option output white ColorFromRGB(255, 255, 255, 1.0)
end

graph everGreenLogo begin
  typeface Typefaces
  bgColors BackgroundColors
  typeColors TypeColors

  radius Double(200.0)
  stroke ColorFromRGB(10, 10, 10, 1.0)
  whiteFill ColorFromRGB(255, 255, 255, 1.0)
  fontSize Double(56.0)

  bg Ellipse(centerX: radius, centerY: radius, fill: bgColors.active)
  bg.stroke(stroke)
  bg.radiusX(radius)
  bg.radiusY(radius)

  logotype Text(text: "EverClean", font: typeface.active)
  logotype.fill(typeColors.active)
  logotype.size(fontSize)
  logotype.originX(radius - 125)
  logotype.originY(radius + 20.0)

  top Line(startX: radius - 130, startY: radius - 37.0)
  top.stroke(typeColors.active)
  top.strokeWeight(3.0)
  top.endX((2 * radius) - 65)
  top.endY(radius - 37.0)

  bottom Line(startX: radius - 130, startY: radius + 37.0)
  bottom.stroke(typeColors.active)
  bottom.strokeWeight(3.0)
  bottom.endX((2 * radius) - 65)
4.2 Developing Color Schemes

4.2.1 Problem Statement

A common task when doing information visualization is to create color scales that work in both black and white and color. To create the scales, the designer must choose colors with the correct luminance. To do so, it helps to be able to preview a color in both black and white and color. Using a tool like Photoshop, a designer might create a series of rectangles filled with the colors of the scale. In this example, I take a similar approach and use Shiro to define swatches, a palette, and the different options for the color value.
4.2.2 Solution

I solve the problem by creating rectangles to act as swatches. One node is created to represent each value and the colors mapped to the value are represented as options. The graph colors is created to compute the palettes. Finally, two states are defined to show the palettes.

The color scale we are creating contains five steps. I create a node to represent each value named Value*. Two options are created in each node, one for each of the color schemes. The ColorFromRGB multi-function creates a color value from three integers (red, green, and blue, respectively).

```
1 node Value1 begin
2   r ColorFromRGB(22, 0, 0)
3   option output red Color(r)
4
5   g ColorFromRGB(0, 22, 0)
6   option output green color(g)
7 end
```
Listing 4.7: Node definition for the first value of the scale.
To convert the colors into the greyscale equivalent, I create a node BlackAndWhite which uses the Grayscale function. Grayscale takes a color as an input and returns on its first output the grayscale equivalent.

```
1 node BlackAndWhite begin
2 input color Color
3 corrector Grayscale(color)
4 output bw Color(corrector)
5 end
```

Listing 4.8: Node to convert colors to grayscale.

To represent the color scale, I create a palette composed of rectangles positioned in a row, one for each value on the scale. Input ports are created to provide the colors and the palette’s position.

```
1 node Palette begin
2 stroke ColorFromRGB(0, 0, 0, 1.0)
3
4 input sideLength Double(50.0)
5 input x Double
6 input y Double
7
8 r ColorFromRGB(150, 0, 0, 1.0)
9 input color1 Color
10 input color2 Color
11 input color3 Color
12 input color4 Color
13
14 // create the swatches
15 r1 Rectangle
16 r1.originX(0.0)
17 r1.originY(0.0)
18 r1.width(sideLength)
19 r1.height(sideLength)
20 r1.stroke(stroke)
21 r1.fill(color1)
22
23 r2 Rectangle
24 r2.originX(50.0)
25 r2.originY(0.0)
26 r2.width(sideLength)
27 r2.height(sideLength)
28 r2.stroke(stroke)
29 r2.fill(color2)
30
31 r3 Rectangle
32 r3.originX(100.0)
33 r3.originY(0.0)
```
After defining the necessary nodes, I create the graph to render the palettes. In colors, I create instances of each of the Value* nodes. Their active options are provided as inputs to the color palette and four instances of BlackAndWhite to convert the colors for the black and white palette.

```plaintext
graph colors begin
  bw1 BlackAndWhite
  bw2 BlackAndWhite
  bw3 BlackAndWhite
  bw4 BlackAndWhite
  colorPalette Palette
  bwPalette Palette
  value1 Value1
  value2 Value2
  value3 Value3
  value4 Value4
  colorPalette.x(0.0)
  colorPalette.y(0.0)
  colorPalette.color1(value1.active)
  colorPalette.color2(value2.active)
  colorPalette.color3(value3.active)
  colorPalette.color4(value4.active)
  // convert the colors to black and white
  bw1.color(value1.active)
  bw2.color(value2.active)
end
```

Listing 4.9: Palette node.
When executed in the Shiro Playground, the two palettes are shown. I created two distinct color schemes of green and red. If a designer wanted to explore several options of red, he could add options with those colors to the appropriate value node.

### 4.2.3 Completed Code

Putting together the definitions and the graph, I get:

```python
include 'geom'

node Value1 begin
  r ColorFromRGB(22, 0, 0, 1.0)
  option output red Color(r)
  g ColorFromRGB(0, 22, 0, 1.0)
  option output green Color(g)
end

node Value2 begin
  r ColorFromRGB(100, 0, 0, 1.0)
  option output red Color(r)
  g ColorFromRGB(0, 100, 0, 1.0)
  option output green Color(g)
end

node Value3 begin
  r ColorFromRGB(150, 0, 0, 1.0)
  option output red Color(r)
  g ColorFromRGB(0, 150, 0, 1.0)
  option output green Color(g)
end

node Value4 begin
  r ColorFromRGB(200, 0, 0, 1.0)
```
option output red Color(r)

ColorFromRGB(0, 200, 0, 1.0)
option output green Color(g)
end

node BlackAndWhite begin
  input color Color
  corrector ColorToGrayscale(color)
  output bw Color(corrector)
end

node Palette begin
  stroke ColorFromRGB(0, 0, 0, 1.0)

  input sideLength Double(50.0)
  input x Double
  input y Double

  ColorFromRGB(150, 0, 0, 1.0)
  input color1 Color
  input color2 Color
  input color3 Color
  input color4 Color

  // create the swatches
  r1 Rectangle
  r1.originX(0.0)
  r1.originY(0.0)
  r1.width(sideLength)
  r1.height(sideLength)
  r1.stroke(stroke)
  r1.fill(color1)

  r2 Rectangle
  r2.originX(50.0)
  r2.originY(0.0)
  r2.width(sideLength)
  r2.height(sideLength)
  r2.stroke(stroke)
  r2.fill(color2)

  r3 Rectangle
  r3.originX(100.0)
  r3.originY(0.0)
  r3.width(sideLength)
  r3.height(sideLength)
  r3.stroke(stroke)
rectangle originX (150.0) originY (0.0) width (sideLength) height (sideLength) stroke (stroke) fill (color4)

// put all the rectangles in a group so they can moved as a unit
Group originX (x) originY (y) children (~r1.rect, ~r2.rect, ~r3.rect, ~r4.rect)
end

// Compute the colors
graph colors begin
  bw1 BlackAndWhite
  bw2 BlackAndWhite
  bw3 BlackAndWhite
  bw4 BlackAndWhite
  colorPalette Palette
  bwPalette Palette
  value1 Value1
  value2 Value2
  value3 Value3
  value4 Value4
  colorPalette.x(0.0)
  colorPalette.y(0.0)
  colorPalette.color1(value1.active)
  colorPalette.color2(value2.active)
  colorPalette.color3(value3.active)
  colorPalette.color4(value4.active)

  // convert the colors to black and white
  bw1.color(value1.active)
  bw2.color(value2.active)
  bw3.color(value3.active)
  bw4.color(value4.active)

  // show the BW colors on their own palette
  bwPalette.x(0.0)
  bwPalette.y(50.0)
  bwPalette.color1(bw1.bw)
  bwPalette.color2(bw2.bw)
bwPalette.color3(bw3.bw)
bwPalette.color4(bw4.bw)
end

date redPalette begin
graph colors
value1 [red]
value2 [red]
value3 [red]
value4 [red]
end

date greenPalette begin
graph colors
value1 [green]
value2 [green]
value3 [green]
value4 [green]
end

Listing 4.10: Complete code for color scheme
4.3 A Basic Interest Calculation

4.3.1 Problem Statement

In Chapter 1, I described the challenges Leslie faced when exploring which of two investment products would give her the best return for her investment of $10,000. In this scenario, the investment products we are to compare are:

- Product A earns 12% interest compounded annually for a three year term
- Product B earns 5% interest compounded annually for a five year term
- Product C earns 7% interest at the end of a two year term

4.3.2 Solution

To write a Shiro program to calculate the returns on the two products, I start by looking for alternatives in the problem description. I notice the principal being invested is constant
in both products. The interest rates, how the interest is calculated, and terms of the investment are different. A node is created for each of the properties of the investment that vary. To handle the interest that is compounded monthly, I created nodes to represent the interest calculations.

```plaintext
node SimpleInterest begin
  input rate Double
  input principal Double
  input time Double
  output newValue Double(principal + (principal * rate * time ))
end

node CompoundInterest begin
  input principal Double
  input compoundFreq Double
  input time Double
  input rate Double
  interestFactor Power(1 + (rate/compoundFreq), compoundFreq * time)
  output newValue Double(principal * interestFactor)
end
```

Listing 4.11: Nodes to calculate interest.

Then I instantiate SimpleInterest and CompoundInterest in Investment as options so I can use a state definition to control which interest calculation is used. In previous examples, I have used options to represent variations in data and use that approach to represent the length of the investment (InvestmentLength), and the interest rate. In Investment, I use options represent variations in computation instead.

```plaintext
node Investment begin
  input principal Double
  input rate InterestRate
  input time Double
  option compound CompoundInterest(principal, rate.frequency, time, rate .rate)
  option simple SimpleInterest(rate: rate.rate, principal: principal, time: time)
  output newValue Double(active.newValue)
end
```

Listing 4.12: Represent the investment

With the investment represented as a node, I create two nodes to represent the alternative lengths of the investment and the interest rates.
Then I use these nodes to create a graph to calculate the value of the investment.

Because we want to investigate the three product configurations, I create three state definitions to generate the results. I select the active options for each of the option nodes.
4.3.3 Completed Code

Bringing together my definitions, the code to explore the two investment products:

```java
node SimpleInterest begin
  input rate Double
  input principal Double
  input time Double
  output newValue Double(principal + (principal * rate * time))
end

node CompoundInterest begin
  input principal Double
  input compoundFreq Double
  input time Double
  input rate Double
  interestFactor Power(1 + (rate/compoundFreq), compoundFreq * time)
  output newValue Double(principal * interestFactor)
end

node InterestRate begin
  input rate Double
  input frequency Double
end

node Investment begin
  input principal Double
  input rate InterestRate
  input time Double
  option compound CompoundInterest(principal, rate.frequency, time, rate.rate)
  option simple SimpleInterest(rate: rate.rate, principal: principal, time: time)
  output newValue Double(active.newValue)
end

node InvestmentLengths[threeYear] begin
  option output threeYear Double(3.0)
  option output fiveYear Double(5.0)
  option output twoYear Double(2.0)

Listing 4.14: States configuring the investment products
node InterestRates begin
  option output productA InterestRate(0.12, 1.0)
  option output productB InterestRate(0.05, 12.0)
  option output productC InterestRate(0.07, 1.0)
end

graph earnMoney begin
  times InvestmentLengths
  rates InterestRates
  retirement Investment

  retirement . rate . rate (rates . active . rate)
  retirement . rate . frequency (rates . active . frequency)
  retirement . principal (10000.0)
  retirement . time (times . active)

  balance Double (retirement . newValue)
end

state productA begin
  graph earnMoney
  retirement [compound]
  times [threeYear]
  rates [productA]
end

state productB begin
  graph earnMoney
  retirement [compound]
  times [fiveYear]
  rates [productB]
end

state productC begin
  graph earnMoney
  retirement [simple]
  times [twoYear]
  rates [productC]
end

Listing 4.15: Completed Investment
4.4 Costume Selection

4.4.1 Problem Statement

Choosing what a character wears is part of the character design process for movies and video games, a process that involves considering a variety of options. In this next example, I characterize how Shiro can be used to support such a design task. In the code that follows, I show how two costumes can be tried. If a designer had a costume with multiple parts, he could use a similar technique to mix and match wardrobe items.

4.4.2 Solution

Character designs are often sketched in a tool like Photoshop, or Illustrator. Both of these tools provide excellent toolsets for drawing images. I start with a base character design created in Illustrator. Using a node to load an image from a file, I render the base character design.

```
1 include 'geom'
2
3 graph costumeSelection begin
4 character Image(originX: 0.0, originY: 0.0, file: 'file://costume_selection/character.png')
5
6 g Group
7 g.children(~character)
8 end
```

Listing 4.16: Render the base character

To show the alternative costumes designs for the character, who bears an odd resemblance to the author, I create **Costumes** to represent the costumes.

```
1 node Costumes begin
2 option bball Image( path: "Costume_Selection/basketball.png")
3 option tux Image( path: "Costume_Selection/tux.png")
4 end
5 end
```

I update the graph **costumeSelection** to place the costume in front of the base character design and create two states to explicitly define the two costume types. Explicitly defining the states allows me to control the names of the alternatives in runtime that tools like the Shiro Playground uses when displaying them.

```
1 include 'geom'
2
3 node Costumes[bball] begin
4 option bball Image(originX: 0.0, originY: 0.0, file: "file://costume_selection/basketball.png")
```

95
Figure 4.8: Draw the base character

```plaintext
option tux Image( originX: 0.0, originY: 0.0, file: 'file://
costume_selection/tux.png')
end

graph costumeSelection begin
  character Image( originX: 0.0, originY: 0.0, file: 'file://
costume_selection/character.png')
  costumes Costumes
  g Group
  g.children(~costumes.active, ~character)
end

state tux begin
  graph costumeSelection
  costumes[tux]
end

state bball begin
```
4.4.3 Completed Code

Put together, my solution is:

```python
22 graph costumeSelection
23 costumes [ bball ]
24 end
```

Figure 4.9: Draw the character with the two costume options

4.5 Analysing Poverty Data

4.5.1 Problem Statement

Imagine you are an operations staff member at an aid organization and a CSV containing poverty data for a number of nations your organization works in has made it into your manager’s hands. The rows of the CSV consist of the country name and the percentage of the population in poverty in 1960, 1990, and 2010 (See Table 4.1 for a sample). This means
that the smaller the percentage, the less people in the country that are impoverished. Your manager would like you to see if there is anything interesting you can learn from the file.

<table>
<thead>
<tr>
<th>Country</th>
<th>Percentage</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aruba</td>
<td>49.224</td>
<td>1960</td>
</tr>
<tr>
<td>Andorra</td>
<td>41.550</td>
<td>1960</td>
</tr>
<tr>
<td>Afghanistan</td>
<td>92.007</td>
<td>1960</td>
</tr>
<tr>
<td>Angola</td>
<td>89.565</td>
<td>1960</td>
</tr>
<tr>
<td>Albania</td>
<td>69.295</td>
<td>1960</td>
</tr>
<tr>
<td>Arab World</td>
<td>68.639</td>
<td>1960</td>
</tr>
</tbody>
</table>

Table 4.1: Sample of the CSV data

### 4.5.2 Solution

Even with only three columns there is a wide variety of different questions that can be explored with the dataset. For the sake of brevity, I will answer only one question, “What was the average percentage of people in poverty in 2010 and which countries were below it?”

I start by defining a graph called `poverty` to read and filter the CSV. We are only interested in the data from 2010, so I filter it out. Once I have filtered the data I display it in a table view.

```plaintext
1 include 'ui'
2 include 'data'
3 graph poverty begin
4   read ReadCSV
5   read . path ('PovertyAnalysis/Poverty.csv')
6 filter2010 FilterTable
7   filter2010 . table (read . table)
8   filter2010 . column ('Date')
9   filter2010 . operator ('==')
10  filter2010 . toMatch ('2010')
11 viewer TableView
12   viewer . table (filter2010 . matches)
13   viewer . tableName ('2010 Poverty Data')
14   viewer . originX (100.0)
15   viewer . originY (100.0)
16 end
```

Listing 4.17: Filter CSV to only include 2010 data

Now that I have the data read and filtered I can calculate the average poverty for 2010. To do this I create another node, `GetBelowAveragePoverty`. 
Figure 4.10: Render the list of countries in 2010

```r
node GetBelowAveragePoverty begin
  input table Table
  a Average
  a.table(table)
  a.column("Percentage")
  aboveAverage FilterTable
  aboveAverage .table(table)
  aboveAverage .column("Percentage")
  aboveAverage .operator(<)
toString Double2String(a.average)
aboveAverage .toMatch(toString)
sc SelectColumn
sc.table(aboveAverage.matches)
sc.column("Country")
output countriesAboveAverage Table(sc.table)
```
Listing 4.18: Calculate the countries below the average poverty.

GetBelowAveragePoverty has a single input that takes table. The table data flows through the Average instance a to calculate the average of the column. The average value is used to filter the table. To filter the table by percentage, the percentage column is selected and evaluated with the less than operator. Because I am only interested in the Country column, I reduce the filtered table to only one column using the SelectColumn node.

Finally, I modify the graph to display the countries below the worldwide average by creating an instance of GetBelowAveragePoverty and connect filter2010’s output to its table port. I update viewer.table to receive the output from belowAveragein2010.countriesAboveAverage.

My completed code:

```bash
#include 'ui'
#include 'data'

node GetBelowAveragePoverty begin
  input table Table
  a Average
  a.table(table)
  a.column('Percentage')
end
```
aboveAverage FilterTable
aboveAverage.table(table)
aboveAverage.column("Percentage")
aboveAverage.operator("<")
toString Double2String(a.average)
aboveAverage.toMatch(toString)

sc SelectColumn
sc.table(aboveAverage.matches)
sc.column("Country")

output countriesAboveAverage Table(sc.table)
end
graph poverty begin
read ReadCSV
read.path("PovertyAnalysis/Poverty.csv")
filter2010 FilterTable
filter2010.table(read.table)
filter2010.column("Date")
filter2010.operator("==")
filter2010.toMatch("2010")

belowAveragein2010 GetBelowAveragePoverty
belowAveragein2010.table(filter2010.matches)

viewer TableView
viewer.table(belowAveragein2010.countriesAboveAverage)
viewer.tableName("Countries Below Average Poverty in 2010")
viewer.originX(0.0)
viewer.originY(0.0)
end

state povertyAnalysis begin
graph poverty
end

Listing 4.19: Completed Poverty Analysis

4.6 Evaluation

I evaluate Shiro by demonstrating how it can be used to solve problems in a variety of domains. The first way we measure if my research is any good is if the system works as intended and demonstrates that it is possible to extend a parametric system to support the representation of multiple alternatives. Given the problems solved earlier in this chapter,
the work is successful. I have designed and implemented a parametric system with support for alternatives and solved several problems with it. “Look Mom! It works!”

When doing a controlled experiment, a researcher is expected to discuss the internal validity, external validity, and ecological validity of the experiment and their findings. Ecological validity refers to how well a study approximates the real world. External validity is a claim to how well the results of the study can be generalized to other people and situations. Internal validity is an indication how well confounds have been avoided. A confound occurs where more than one independent variable maybe manipulated, which reduces the researcher’s ability to make claims that manipulating an independent variable caused the observed results. The measures of validity I have described are often used to evaluate controlled experiments, where there are clear independent and dependent variables. The intent of my work is to build and demonstrate a new technology and as a result is not structured as a controlled experiment, so the traditional definitions validity do not directly apply. Instead, what is possible is to adapt the definitions of the concepts as necessary so they apply to design. Ecological validity refers to how well research approximates the real world, so
we can use this definition without modification. External validity requires modification so the concept refers to how well the technology can be used by other people in a variety of situations. Internal validity does not apply. In my work, I do not have a research question that can be examined by the manipulation of a variable and observing the result of another. This work is design and is exploratory in nature.

4.6.1 Ecological Validity

Shiro provides a syntax for describing alternatives in parametric systems. It can be used to solve problems in a variety of different domains as demonstrated previously in this chapter. In terms of evaluating a technology by demonstration, ecological validity is one of the most important factors to consider.

Shiro is a working implementation of a parametric system as described by Elements of Parametric Design [56]. It provides a syntax for expressing parametric systems. The syntax provides way for programmers to create nodes and connected ports. Solutions are represented by graph declarations and state configurations. Shiro’s functional paradigm supports recursion and operations on collections. One weakness of this work is that it only demonstrates the solution of problems using written code. It is common for designers and other people to use a graphical programming environment rather than writing code. Given the challenges of designing and implementing a new language, I left this task as future work because tools such as Processing\textsuperscript{2} and Generative Components\textsuperscript{3} allow designs to specify their programs with code and the goal of my work was to develop a parametric system with support for alternatives rather than develop all the parts of a new parametric design application with support for alternatives. For the sake of this work remaining a manageable scope, I only implemented Shiro as a textual language. As I will show in Chapter 5, I believe this work lays the foundation for a system with more robust support for alternatives.

Use of Shiro to solve real world problems provides additional strength to the argument that this work is ecologically valid. In evaluating Shiro, I tried to solve problems people would face in reality. The logo design example is a common task asked of many graphic designers. The design process might be longer if the design explores more alternatives, but the type of work done here is typical of logo design tasks I have participated in running my design consultancy.

One criticism of my choice of example problems might be that the problems are too simple. All of the problems can be solved over the course of a single day. Some can be solved in a few hours. None of the examples represent problems requiring multiple days of work. While the problems can be solved in a single day, they are representative of the type of work that would be done on problem requiring multiple days. For example, the data analysis problem (Section 4.5) completely answers a single question. Further analysis

\textsuperscript{2}http://processing.org

\textsuperscript{3}http://www.bentley.com/en-US/Promo/Generative%20Components/default.htm
of world poverty for those three years would generate more questions and each of those
questions would be answered in a similar manner. I argue a longer problem would only
represent more of the same type of work. The work required would differ in volume not type.
I make this argument because it’s possible to solve problems of many sizes with existing
programming languages and Shiro has many of the same features as other languages. It
has a means of sharing code—the include statement. It has a means of encapsulating
computation—the node. It has a means of defining multiple graphs and defining states for
each graph definition. I cannot say conclusively that Shiro will make it easier to solve large
problems, but by argument by similarity, Shiro should be able to handle larger problems at
least as well as other programming languages. How the addition of states and subjunctive
nodes impact this use, is a question for future research.

4.6.2 External Validity

Closely related to ecological validity, is external validity. Having solved problems in a
variety of domains provides support that this work is externally valid. Shiro is not domain
specific and can be adapted for use in any domain that has problems able to be expressed
as parametric systems. To adapt Shiro to a new domain, a set of functions need to be
implemented to represent any domain specific computations. For example, if Shiro were to
be used for audio processing, functions to manipulate tracks and frequencies would need
to be implemented. This is the approach I took when implementing the functions for data
analysis. I implemented the functions I needed to complete the manipulation of a table of
data values. The other way to adapt Shiro to a domain is to create nodes of already existing
functions for domain specific tasks.

Shiro is also consistent with existing literature in the field. It can be used to imple-
ment features described in previous user interface designs. Shiro extends the current notion
of a parametric system with subjuncts. Chen’s formalism for expressing alternatives could
provide a basis for specifying generators, or programs for creating new alternative from
existing ones. Chen’s formalism[11] does not take into account alternative computations,
but applying his operators to Shiro would allow parameter values and computation to be
varied.

Kolarić et al.’s [24] operators of pass value and pass object can be implemented using the
Shiro runtime. To implement pass value, a host application needs only to modify a single
property in a graph and all the states that reference the graph will be updated. Pass object
can be implemented by creating a new graph with a new instance of the object to be passed
and updating the state to refer to this graph. Kolarić et al.’s [24] and Zaman et al.’s [58]
notion of isolating changes to an artifact can be implemented by wrapping a property in
a node and making each of the unique values an options. Then the state referring to the
graph, being isolated, can be updated to select the correct option.
4.6.3 Internal Validity

Internal validity does not apply to this work as the concept addresses the need to guard against confounds in an experiment. The purpose of this work is to develop a technology that makes it easier to manage alternatives. The impact of subjunctive nodes and states on design or visual analytics practice might be experimentally evaluated in the future and in those experiments internal validity should be preserved. For the purposes of the evaluating this work, however, the evaluation should be done with respect to goals of extending a parametric representation. Shiro is a successful design because it works and provides a new technology to problem solvers. Problem solvers can now use subjunctive nodes and states to simultaneously represent alternatives in a parametric system. Before this work, problem solvers were subject to work-arounds to manage alternatives.

4.6.4 Replication and Alternatives

Parametric design tools like Grasshopper provide replication mechanisms (See 2.3) to allow programs to implicitly represent iteration and in Section 2.6.5 I described how Grasshopper provides a basic means of representing alternatives by allowing designers to store sets of input values. Given the ability to represent iteration and store sets of input parameters, one might ask if Shiro is necessary. Don’t replication and the ability to save input parameter sets allow the representation of alternatives in a single model file? In the simplest sense, yes. These two features allow the representation of multiple things in a single model file. In fact, any computational representation with lists and the ability to choose which list is active can represent data alternatives. What can’t be done with Grasshopper is vary computation. You can create a subgraph as an alternative. In Shiro, options can represent both values and computation, whereas in Grasshopper replication and set of input parameters prevents alternative computation from being represented.

Shiro introduces the constructs of a subjunctive node, graph, and state to create an explicit representation of the alternative artifacts in program. Using replication and a stored input parameter set, the alternatives are mixed into the definition of the graph. The drawback of this approach is it becomes difficult for tool builders to reference an alternative. There is no symbol that represents them. Replication is best used to combine lists and represent multiple items within a single artifact. For example, replication is useful for creating grids of points used to create geometry. Replication concerns making multiple items in an artifact, which is a separate concern from representing an alternative. Shiro makes alternatives first-class objects in parametric systems and allows the alternative itself to become the object manipulated by a higher level abstraction like a genetic algorithm. Replication requires that alternatives be represented at a lower level abstraction.
Chapter 5

Future Work

In the process of building Shiro, I learned much about how to represent alternatives in a parametric system. In addition to Shiro itself, the second contribution this thesis makes is to capture how I would have built Shiro if I knew when I started this work what I know now. These insights are shared here and, should someone endeavour to extend my work, I believe my observations provide the person with a clear way forward. The first step to improve Shiro is to implement a more robust internal, abstract representation of a program. Further work is also need to better understand the computation and user interaction needs of systems that provide first class support for alternatives.

5.1 Integrated Text, Graphical Programming, and Direct Manipulation Interfaces

To support the development of tools that depend on Shiro’s parametric system, we need to change the way a program is represented in the runtime. In the current design (See Section 3.3), the runtime stores a copy of the parse trees generated during parsing and rewalks them during evaluation. The parse trees are create by the parser ANTLR [39] generates from the Shiro grammar and are defined in terms of objects from the ANTLR runtime library. The problem with this approach is that it provides no way for host applications, or applications that integrate the Shiro runtime to modify the current program. Host applications would be forced to generate Shiro code and reload the modified code any time a change is made.

This limitation makes it difficult to provide ways to modify a Shiro program other than text. For example, a tool builder might want to create a drawing editor that allows the designer to draw lines like he does in Adobe Illustrator, but to have the drawing represented as a parametric system. To do this, the Shiro runtime must allow the tool builder to write code in the host application to do operations like create new nodes, create connections between ports, add options to subjunctive nodes, and define brand new subjunctive nodes.
from existing instances of nodes. For this to be possible, the runtime must provide an API. To support such an API, the Shiro runtime needs to maintain an editable internal representation of the program. In the current implementation, the parse trees are generated by an ANTLR parser and are very difficult to modify. Instead, the runtime should create its own abstract representation of a Shiro program during the parsing phase, so host applications can modify the running program.

The internal representation of the Shiro program is necessary to generate a textual version of the current state of the runtime. If a host application is able to modify the current state of the runtime without creating Shiro code and re-interpreting it, it follows that a host application should also be able to output the current state of the runtime as code. With an API to modify the current state of the runtime and output it as Shiro code, tool builders have what is needed to create the user experiences they need to support working with alternatives.

An internal, abstract representation of Shiro code would provide the foundation to support direct manipulation, a graphical programming environment, a code editor and changes made via an API. With direct manipulation, tool builders allow users to interact with objects in their domain. For example, in a graphics application, users draw shapes and modify their properties. In order to build such a user interface, the developer translates mouse and keyboard events into calls to the API that modifies the Shiro runtime. Similarly, in the graphical programming environment, users interact with less abstract graphical representations of Shiro nodes and ports. Users create instances of nodes and create connections between them. The implementation of the GUI widgets translates changes to the widgets into changes in the current program in the runtime. A text editor that is kept in sync with the runtime can also be provided. Changes to the text are identified and converted into the appropriate changes to the runtime. Successful designs will incorporate the concepts of brushing and linking [8] from visual analytics and liveness, localization, and lookahead [34] from parametric CAD. I predict the best tool for manipulating alternatives will be one that provides direct manipulation, a graphical programming environment, and a textual editor as user interfaces, so that a user is able to choose the representation that works best for his task and is easily able to move between them.

5.2 Event Driven Execution

During the course of this work, I became aware of RxJava\(^1\) and the field of functional reactive programming (FRP) [38, 32]. Functional reactive programming provides us with ideas for a different approach to evaluate parametric systems. FRP extends the Observer Pattern [54] to create representations of functions that notify their observers called Observables when

\(^1\)https://github.com/ReactiveX/RxJava
they finish executing. Observables can be organized into streams and data is transformed as it flows through the stream.

Shiro is executed using the approach described in *Elements of Parametric Design* [56]: the runtime sorts the ports in a Shiro program into a topologically sorted list and iterates through the list to evaluate each port. The weakness of this approach is that it requires execution of the program to be explicitly called. If a parameter is changed, the graph is not recalculated until execute is called. This approach works well for executing programs that are expected to exit at the end of their execution; however, it makes it more challenging to develop interactive applications because a developer has to write code to explicitly evaluate the parametric system. In contrast, using a more event-driven approach, like FRP, would cause a change to the value of a port to invoke the execution of the program and only the ports observing the changed port would execute ports downstream for it. By automatically evaluating the parametric system each time new values are assigned to the parametric system’s inputs, the runtime would be able to handle streams of input data.

I implemented a prototype of event-driven execution, but did not, because of time, integrate it into the Shiro runtime or explore it in detail. At a glance, the technique works and will remove the need to create a graph to represent dependencies so the ports can be topologically sorted.

5.3 Improving Performance

Little thought was given to runtime performance of the Shiro runtime as it was implemented. I wanted to avoid the trap of early optimization and focus my efforts on finding a way to represent alternatives. I did, however, keep my eyes open for opportunities to improve the performance of the system because if Shiro is to be a successful tool for representing alternatives it must be able to efficiently execute many hundreds and likely thousands of alternatives. In this section, I discuss some ways the runtime can be made more efficient.

5.3.1 Parallel Execution and Caching

The approach of evaluating Shiro by topologically sorting a graph and evaluating the produced list is embarrassingly serial. The Shiro runtime iterates through each of the declared states and evaluates the graph with the specified options. To evaluate large sets of states, this process will be slow. A logical next step is to make use of the multiple cores present in most modern CPUs and take advantage of Shiro being a functional language. One possible approach to parallelization might be to do the computation of each port on separate thread.

A cache may also make the computation of alternatives in Shiro programs more efficient. Because Shiro is a functional language, there is one and only one result tuple for each unique set of arguments passed to function. This means that once a function has been evaluated it should never have to be evaluated with the same parameters again. For complex
computations, I hypothesize the value of the cache will be quickly noticed. For simple operations, like addition and subtraction, I am not sure if the overhead of the cache look up will be more expensive than actually recalculating the result. Given that non-trivial programs will likely include many complex functions, implementation of a cache will likely provide a performance benefit.

Two possible approaches for implementing the cache are a global cache and a local cache. With a global cache, the runtime will maintain a single cache for all functions. A global cache would provide the simplicity of a central location to store all of the results. It comes with the downside that every function will need to maintain a reference to the cache. Because writes to the cache would be done from within the function implementation, it would be still possible to set function specific caching rules. For example, it might be better to prevent simple math ports for being cached if that turns out to be more expensive. The function could contain a cache policy that would determine if the result was to be cached. Using a local cache would remove the need for each port to maintain a reference to the cache and for functions where caching doesn’t make sense the cache can be left unimplemented. The drawback is the approach might increase implementation complexity as multiple instances of a function are created in Shiro program so the actual computational part of the function would need to be abstracted out of the function implementation, so it can be used in multiple port instances. A cache comes at the cost of increased memory usage, but should make Shiro programs execute faster.

5.4 Type System

Shiro type system is simple and does not use type inference. Types either match or not. The only place where a small amount of intelligence is implemented is in the basic math operators that take both Integer and Double and coerce the values so there is no loss of information. When an Integer and Double are passed as arguments the Integer is coerced into a Double. Future work should examine if a more robust type system is needed and if so what that type system should allow. What follows is a sketch of how the type system might be approached.

In Shiro types can be defined in terms of the inputs, and outputs of the functions. Nodes allow the programmer to define their own types in addition to those provided by the runtime in the standard library. One of the core concepts of Shiro is to allow programmers to represent and relate blocks of computation. Functions are related by connecting their inputs and outputs. As a result, types can be defined in terms of a functions inputs and outputs.

This notion of types is similar to duck typing in Ruby, where if an object talks like a duck, it is a duck. In Ruby objects are strongly and dynamically typed. Calling a method on the object is equivalent to sending a message to the object. As long as an object responds
to the message it can be used in place of any other object that also responds to the message; the type of the object does not change. If an object does not respond to the message, it throws a NoMethod exception.

For example, given:

```plaintext
node A begin
  output x Double(1.0)
end

node B begin
  output x Double(1.0)
end

node C begin
  output x String('1.0')
end
```

Listing 5.1: Definitions of nodes A, B, and C

Functions A and B are input compatible. Two functions are input compatible if and only if the names, positions, and types of their inputs are the same. That is they share the same input interface.

Functions A and B are output compatible. Two functions are output compatible if and only if the names, positions, and types of their outputs are the same. That is they share the same output interface. Functions A and C are not output compatible because their outputs match in name and position, the type of C.x is String and the type of A.x is Double.

Functions A and B are compatible. Two functions are compatible if and only if they are output compatible and input compatible. Functions that are compatible can be used in place of each other.

In practice, this means that functions that are compatible can be assigned to one another. For example:

```plaintext
node Box begin
  input length Double
  input width Double
end

node Square begin
  input length Double
  input width Double
end

node F begin
  input box Box
end

a Double(100.0) // instantiate a Double with value 100.0
```
Box and Square are defined as compatible types and can be used in place of one another. Support generic for type definitions that only specify the inputs or outputs of the function might also be possible. For example, it might makes sense to write:

```
node A begin
  output x Double (1.0)
end

node B begin
  output x Double (4.0)
end

node X begin
  input a <[]: [x: Double]> // abstract type definition
  output o Double (a.x)
end

x X(a: ~B()) // sets a to an instance of B
x.o // outputs 4.0
```

Listing 5.3: Generic Types

where an anonymous reference is passed to set the instance of a. `input a <[]: [x: Double]>` says that a must be passed a function instance that has no inputs and whose first output is named x and is of type Double. Node X is defined so that either A or B can be passed as the node. The same might be possible using a reference path:

```
node Box begin
  input length Double
  input width Double
end

node F begin
  input box <[length: Double, width: Double]:[]>
end

a Double (100.0) // instantiate a Double with value 100.0
b Box // instantiate a Box

t F(~b)
```

Listing 5.4: Instance passed via a reference path
Being able to distinguish input compatible types may also be useful for the type of graph refactoring operations needed to combine graphs when merging solutions—something that is necessary in direct manipulation or graphical programming environments.

Thus to acknowledge the role of inputs and outputs in Shiro’s type system, I cheekily continue in the pattern of naming typing styles after animals and propose that Shiro be elephant typed; if a function eats like a elephant, and poops like an elephant, it is an elephant.

5.5 Treat Solution Space as Database

Shiro allows a programmer to represent a program with many possible results. These results represent the solution space for the program. Each alternative is a point in the space. Shiro state definitions describe the configuration of an alternative and represent a point in the solution space. In a parametric model with a number of options defined, the number of possible alternatives is combinatorially large. To provide a more flexible mechanism for retrieving the parts of the solutions space the user is interested in, I think the solution space should be treated as a database and the user allowed to query it. Given the following simple Shiro program:

```plaintext
node Layout begin
  input width Double
  output length Double
end

node Pool [basin] begin
  option singleLane Layout (width: 10.0, length: 30.0)
  option basin Layout (width: 20.0, length: 20.0)
  option olympic Layout (width: 25.0, length: 30.0)
end

node Box begin
  input length Double
  input width Double
  input height Double
  output volume Double (length * width * height)
end

node Cost begin
  input volume Double
  output cost Double (volume * 12.0)
end

graph poolCosts begin
  pool Pool
```
Recall this program defines several alternative pool design. Rather than always require the user to define states or have the runtime to generate all the possible states, I think it would be useful to allow the user to write a SQL-like query of the evaluated graph in each alternative. The query would allow the user to specify queries like, “Show me all the pools that cost less than $20,000.” The syntax could be modeled after the WHERE-clause of a SQL query and might look something like:

```
WHERE poolCosts.c.cost < 20000
```

The query could be used to search evaluated solutions and to constrain the what alternatives are evaluated if the query condition references an input condition. A query condition that references an input condition can be used to constrain the number of states that need to be evaluated as inputs do not depend on the rest of the graph and can be used to create state definitions. Also, it might be useful to be able to query the properties of the artifact the program represents. This would require an artifact specific implementation of search and the runtime could provide a way to extend the runtime’s implementation of search, but it is conceivable a user might want to search the alternatives with a particular property not easily determined by the properties of the graph, say find all images fitting a certain color palette, or find all images of a particular shape. Combined with intelligent state generation, search could provide an important mechanism for tools to support the creation and exploration of large collections of alternatives.

## 5.6 Generating Alternatives

In Section 3.2.3, I described how option and state definitions can be used to represent alternative values. Shiro does not provide a way to generate a specific set of alternatives. The runtime either generates all of the possible alternative states or it computes the states defined by the state definitions in the program. An alternative value must be declared as a option and realized with a state definition. A Shiro programmer must manually define each alternative he wishes to realize. I considered the generation of alternatives outside the scope of this work as my goal was to design and implement a compact representation for alternatives. Now that we have a way to represent alternatives, we can begin to explore how to generate the alternatives.

It is possible to generate Shiro states with an external program. The program would need to define nodes and options in the nodes. These generated definitions could be generated
as Shiro source code or as a program that uses the Shiro runtime API. To use a genetic algorithm or some other means of generating states would require the algorithm to define the appropriate options, states, and graphs. Evaluation of the output could be done by inspecting the graph or the resulting artifact. The possibilities for implementing generators are infinite as a generator is simply a program that generates alternatives. Anything that can be used as input in a program could be used influence the creation of alternatives. Shiro will compute defined alternatives and return the results. What the algorithm does after that, is up to the creativity of its designer.

Chen’s [11] notion of operators on variation heads provides one possible starting place. Recall Chen described variation heads, in Shiro terms, as the set of ports not dependent on any other port. By operating on the sets of possible values for a variation head, a number of different results could be specified. Work to apply his work will take into account Shiro’s concept of an option and consider it in the evaluation of operators on variation heads. With Shiro, the definition of an alternative is expanded beyond the values specified in the variation head and includes the options selected for each subjunctive node in the graph. The output of Chen’s operators will be a Shiro graph and state definitions.

5.7 Reconcile History and Alternatives

In Section 3.6, I briefly discussed one perspective of alternatives and history. Further work is needed reconcile the concepts of history, exploration of alternatives, and more generally the relationship between time and how people create artifacts. These words have different meanings depending on whether a user is using them or a developer. Artifacts are created by users using tools over time. Yet, history, or provenance rather, refers to the transformations the artifact undergoes. Where in the past the creation of alternatives has largely occurred as the result of the serial work of user, Shiro makes it for a user to simultaneously operate on a single artifact with many possible alternatives states. So, what is the lineage of artifact? What role does its state play? How do people comprehend having multiple simultaneous states of an artifact? Unpacking these concepts will be an important part of continuing to advance the quality of creativity support tools.

5.8 Build a Tool

In the prior sections, I have described a series of smaller problems created and prompted by the creation of Shiro. Each of those sections represents a problem that needs to be solved to continue this work; however, to put this work to the best test and to learn how to build tools with integrated support for parametric alternatives, I think a more complete creativity support tool needs to created. I believe it should be one that integrates a direct manipulation, graphical programming, and scripting.
Prior research has resulted in a body of knowledge that solves or describes solutions to many of the problems one would encounter when building a more complete tool. Future work should consider the work of Kirsh [23, 21, 22] on thinking with representations, the work of Kolaric et al. [24], Zaman et al. [58], Shireen et al. [42], and Maleki et al. [33, 35, 34], and Woodbury et al. [4, 57, 56] on interacting with alternatives in a parametric system. From this foundation, some of the possible research questions that are:

- How do designers use the three different representations of their design?
- What kinds of tasks do each representation support?
- How do designers use the ability to create nodes and states to complete their work?
- What are the strengths and weakness of the Shiro syntax and constructs?

Given parametric systems are commonly used in the field of design. I believe an excellent possibility for the tool that integrates the research I have cited is a two dimensional vector editor. Tools like Quartz Composer\(^2\) and Origami\(^3\), which provide a graphical programming environment for drawing, have started to become popular as user interface prototyping tools. Direct manipulation tools like Adobe Illustrator and Sketch are also popular. I believe that a tool that integrates the graphical programming and direct manipulations will provide an excellent area for further experimentation as the design field has become familiar with the interaction techniques and will reduce the difficulty of studying a new phenomenon like alternatives in the field of design.

Shiro provides a foundation on which to continue exploring how to support alternatives in parametric tools. While the Shiro runtime bears many of the weakness of a research a prototype and needs to be improved in the areas of performance and internal language representation, the Shiro runtime provides a foundation on which to build tools that support alternatives in parametric tools.

\(^2\)https://en.wikipedia.org/wiki/Quartz_Composer
\(^3\)https://facebook.github.io/origami/
Chapter 6

Conclusions

Human problem solving is a non-linear and exploratory process that involves making multiple alternatives and expressing them in external media. Current tools lack support for alternatives and require users to use workarounds to work with them. People are forced to improvise when working with alternatives because most tools have a single-state document model. This means tools are designed to only operate on a single artifact at a time. For example, Microsoft Word only allows a user to work on a single document at a time. In Grasshopper, a designer may try many parameter values for his parametric model, but only a single set them can be active at a time.

To address this problem, Shiro provides a multi-state document model for representing alternatives. By defining subjunctive nodes, nodes, with options, user can define parametric systems with multiple states. States can be explicitly defined or generated by the runtime. The introduction of subjunctive nodes, options, and states as constructs in the language allows Shiro programmers to simultaneously express alternative versions of an artifact. The Shiro runtime takes care of evaluating each of the states and provides an API for host applications to access the results. Because of Shiro’s declarative syntax, Shiro can be extended to support history by implementing transactions in the runtime. Host applications would only need to generate the transactions and the runtime would allow the host application to step forward and backward through history. Jumping to a particular point in time is made more efficient by Shiro’s compact representation of the document’s state. The document state can be reloaded and re-executed instead of applying a sequence of transactions, which, in cases where the history record is long, will be quite a bit slower.

I demonstrated Shiro’s value by solving problems in the domains of data analysis and design. The Shiro Playground provides a reference implementation of an application that utilizes the Shiro runtime to author and display alternatives. The Shiro runtime provides a platform on which future research into alternatives can be completed. Hopefully, having the runtime available will enable interface researchers and computing researchers to collaborate and more quickly refine new approaches for displaying and manipulating alternatives.
Bibliography


Microsoft. Multiple-document interface (mdi) applications.


Appendix A

Shiro ANTLR Grammar

The following is the ANTLR\(^1\) grammar for Shiro. Its syntax is similar to EBNF.

```
grammar Shiro;

shiro : includeStmt*  
    shiroStmt*  
    EOF  
    ;

includeStmt: INCLUDE STRING_LITERAL NEWLINE;

shiroStmt:  
    anonymousGraphStmt  
    | nodeDecl  
    | graphDecl  
    | stateDecl  
    | NEWLINE  
    ;

stateDecl:  
    STATE stateName BEGIN NEWLINE  
    stateGraphSelection NEWLINE  
    stateStmt*  
    END  
    ;

stateName:  
    IDENT  
    ;
```

\(^1\)http://www.antlr.org/
stateStmt
   : stateActivation NEWLINE
   | NEWLINE
   ;

stateGraphSelection
   : GRAPH (IDENT | DEFAULT)
   ;

stateActivation
   : optionSelection
   | nestedOptionSelection
   ;

nestedOptionSelection
   : nodeName=IDENT LSQUARE activeObject=IDENT RSQUARE BEGIN NEWLINE
      (stateActivation NEWLINE | NEWLINE)*
      END
   ;

optionSelection
   : nodeName=IDENT LSQUARE activeObject=IDENT RSQUARE
   ;

graphDecl
   : GRAPH IDENT BEGIN NEWLINE
   graphStmt+
   END
   ;

graphStmt
   : portAssignment | funcDeclInit | funcDecl | NEWLINE
   ;

nodeDecl
   : NODE MFNAME ('[ optionSelector ']')? BEGIN NEWLINE
   nodeStmt
   END
   ;
anonymousRef
   : reference
   ;

reference
   : REF fullyQualifiedType ( LSQUARE activeObject=IDENT RSQUARE )?
     ( '(' arguments? ')' )? outputSelector?

123
outputSelector
   :  (LSQUARE selectedOutput=(IDENT| NUMBER) RSQUARE)
   ;

funcDeclInit
   :  name=IDENT fullyQualifiedType ( LSQUARE activeObject=IDENT RSQUARE )?
      (('(' arguments ')'))
   ;

funcCall : fullyQualifiedType ( LSQUARE activeObject=IDENT RSQUARE )?
      (('(' arguments ')'))?;

funcDecl
   :  name=IDENT fullyQualifiedType ( LSQUARE activeObject=IDENT RSQUARE )?
   ;

arguments
   :  argMap | argList
   ;

argMap
   :  (keys+=IDENT '::' values+=arg)(',', keys+=IDENT '::' values+=arg)*
   ;

argList
   :  arg(',, arg)*
   ;

arg: expr;

optionSelector
   :  IDENT
   ;

nodeStmt
   :  (portstmt  
      | portAssignment 
      | nodeDecl 
      | NEWLINE)*
   ;

portDecl
   :  OPTION? accessModifier? funcDecl
   ;
portDeclInit
  : OPTION? accessModifier? funcDeclInit
  ;

portstmt
  : ( portDeclInit | portDecl ) NEWLINE
  ;

portName
  : IDENT
  ;

accessModifier
  : INPUT | OUTPUT
  ;

fullyQualifiedType
  : types+=MFNAME (’.’ types+=MFNAME)*
  ;

path
  : (REF| SELECT)? segments+=pathSegment (’.’ segments+=pathSegment)*
  ;

pathSegment
  : IDENT
  | (INPUTS| OUTPUTS) LSQUARE pathIndex RSQUARE
  ;

pathIndex
  : index=(NUMBER | IDENT)
  ;

portAssignment
  : path ‘(‘ arguments ’)’ NEWLINE
  ;

anonExpr
  : expr NEWLINE
  ;

anonymousGraphStmt
  : portAssignment
  | funcDeclInit NEWLINE
  | funcDecl NEWLINE
  | anonExpr
  ;
expr : '(' expr ')' #parensExpr
| NOT_OP expr #notExpr
| MINUS_OP expr #negExpr
| expr AND_OP expr #andExpr
| expr OR_OP expr  #orExpr
| expr (DIV_OP | MULT_OP | MOD_OP) expr #multExpr
| expr (PLUS_OP | MINUS_OP ) expr  #addExpr
| expr (GT | GTE | LT | LTE) expr  #comparisonExpr
| expr ( EQ | NEQ ) expr  #equalityExpr
| fullyQualifiedType #typeExpr
| anonymousRef    #anonRefExpr
| funcCall        #inlineFuncCall
| path            #pathExpr
| listLiteral     #listExpr
| NUMBER          #numExpr
| BOOLEAN_LITERAL #boolExpr
| STRING_LITERAL  #stringExpr
;

SELECT : '@';
REF : '~';
DEFAULT : '^';
INPUT : 'input';
OUTPUT : 'output';
EVAL : 'eval';
THIS : 'this';
NOT_OP : '!' ;
AND_OP : '&&';
OR_OP : '||';
PLUS_OP : '+' ;
MINUS_OP : '-';
MULT_OP : '*';
DIV_OP : '/';
MOD_OP : '%';
LSQUARE : '[';
RSQUARE : ']';
GT : '>'; 
GTE : '>=';
LT : '<';
LTE : '<=';
EQ : '==';
NEQ : '!=';