

# **Creating with Code**

An Introduction to Functional Programming,  
User Interaction, and Design Thinking

Version 1

**Christopher Kumar Anand**

**Gurleen Dulai**

**Luna Yao**

**Mariyam Arief**

**Olisha D'Mello**

**Sarath Sajiv Menon**

**Christopher William Schankula**

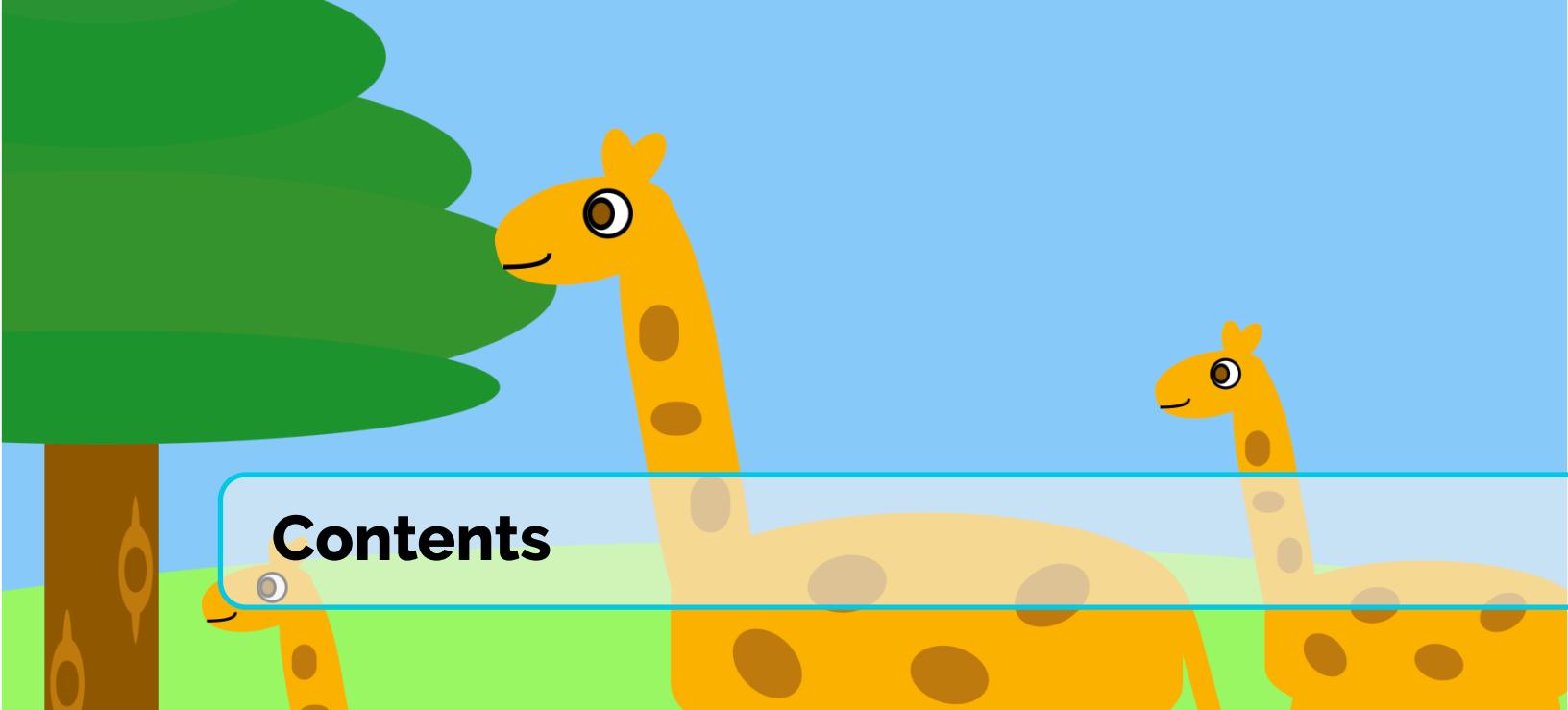
Copyright © 2023 Christopher Kumar Anand, Gurleen Dulai, Luna Yao, Mariyam Arief, Olisha D'Mello, Sarath Sajiv Menon, and Christopher William Schankula

Generated using XeLaTex, using template [Legrand Orange Book](#)

Published by [Fondation STaBL Foundation](#), with a grant from the [Paul R. MacPherson Institute for Leadership, Innovation and Excellence in Teaching at McMaster University](#), and support from [McMaster Start Coding](#).

ISBN 13: 978-1-7388695-0-3

Licensed under the Creative Commons Attribution-NonCommercial 4.0 License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <https://creativecommons.org/licenses/by-nc-sa/4.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “as is” basis, without warranties or conditions of any kind, either express or implied. See the License for the specific language governing permissions and limitations under the License.



# Contents

I

## Elm

<b>1</b>	<b>Getting Started</b>	<b>11</b>
<b>1.1</b>	<b>Drawing with Stencils</b>	<b>11</b>
<b>1.2</b>	<b>Transformations</b>	<b>16</b>
<b>1.3</b>	<b>Text</b>	<b>19</b>
<b>1.4</b>	<b>Don't lose that perfect colour!</b>	<b>20</b>
<b>1.5</b>	<b>Make Your Own Functions</b>	<b>21</b>
<b>1.5.1</b>	Grouping Shapes	22
<b>1.6</b>	<b>Comments and Naming</b>	<b>26</b>
<b>2</b>	<b>Let's Get Moving</b>	<b>27</b>
<b>3</b>	<b>Functions and Scope</b>	<b>32</b>
<b>3.1</b>	<b>Foundations</b>	<b>34</b>
<b>3.2</b>	<b>Types</b>	<b>35</b>
<b>3.3</b>	<b>Partial Function Application</b>	<b>37</b>
<b>3.4</b>	<b>Operators versus Functions</b>	<b>39</b>
<b>3.5</b>	<b>Anonymous Functions</b>	<b>40</b>
<b>3.6</b>	<b>let ... in ...</b>	<b>40</b>
<b>3.7</b>	<b>Modules</b>	<b>41</b>
<b>3.8</b>	<b>Scope</b>	<b>43</b>

<b>4</b>	<b>To be or not to be in the Basement .....</b>	<b>48</b>
<b>4.1</b>	<b>State Diagrams .....</b>	<b>49</b>
<b>4.2</b>	<b>Implementing in Elm .....</b>	<b>51</b>
4.2.1	Example: an Elevator .....	51
<b>4.3</b>	<b>The StateDiagrams Module .....</b>	<b>55</b>
<b>4.4</b>	<b>SD Draw .....</b>	<b>56</b>
<b>4.5</b>	<b>Model-View-Update with TEA .....</b>	<b>67</b>
<b>4.6</b>	<b>Simulation .....</b>	<b>69</b>
<b>4.7</b>	<b>Real-Time Interactive Games .....</b>	<b>75</b>
<b>4.8</b>	<b>Clickable Ruler .....</b>	<b>77</b>
<b>4.9</b>	<b>Mouse Over .....</b>	<b>78</b>
<b>4.10</b>	<b>Slider .....</b>	<b>80</b>
<b>4.11</b>	<b>Drag and Drop .....</b>	<b>82</b>
<b>4.12</b>	<b>Composing Model-View-Update Modules .....</b>	<b>85</b>
<b>5</b>	<b>More Useful Math .....</b>	<b>91</b>
<b>5.1</b>	<b>Clipping and Shape Math .....</b>	<b>93</b>
<b>5.2</b>	<b>Following A Path .....</b>	<b>97</b>
<b>5.3</b>	<b>Animation via Interpolation .....</b>	<b>100</b>
<b>5.4</b>	<b>Animation in Vector Spaces .....</b>	<b>102</b>
<b>6</b>	<b>Core Packages .....</b>	<b>106</b>
<b>6.1</b>	<b>Core Data Types, Math, etc. ....</b>	<b>106</b>
6.1.1	Basics .....	106
6.1.2	Tuple .....	111
<b>6.2</b>	<b>Strings .....</b>	<b>112</b>
6.2.1	String .....	112
6.2.2	Char .....	116
6.2.3	Higher-Order String functions .....	116
6.2.4	Left versus Right .....	119
<b>6.3</b>	<b>Polymorphism and Standard Interfaces .....</b>	<b>119</b>
<b>6.4</b>	<b>Containers .....</b>	<b>121</b>
6.4.1	List .....	122
6.4.2	List Queries .....	123
6.4.3	Result, Maybe .....	136
6.4.4	Array, Dict, Set .....	140
6.4.5	Bitwise .....	147

<b>6.5</b>	<b>Development</b>	<b>148</b>
6.5.1	Debug	148
<b>6.6</b>	<b>Elm Platform: Communicating with the outside world</b>	<b>150</b>
6.6.1	Platform.Cmd	150
6.6.2	Random Numbers: Let's roll the die	150
6.6.3	Task: Making the world go around	153
6.6.4	Platform.Sub	156
<b>7</b>	<b>Tower of Hanoi</b>	<b>158</b>
<b>7.1</b>	<b>Dividing and Conquering Map</b>	<b>163</b>
<b>8</b>	<b>Composing Music in Elm</b>	<b>164</b>
<b>8.1</b>	<b>Music Creator &amp; Basics</b>	<b>164</b>
<b>8.2</b>	<b>The Elm Music Slot</b>	<b>165</b>
8.2.1	Validation & Tempo	165
8.2.2	myMusic	166
8.2.3	GraphicSVG Definitions & Lyrics	168
<b>8.3</b>	<b>Next Steps</b>	<b>168</b>
<b>9</b>	<b>Algebraic Expressions</b>	<b>169</b>
<b>9.1</b>	<b>Simplification</b>	<b>171</b>
<b>9.2</b>	<b>Derivatives</b>	<b>173</b>
<b>10</b>	<b>Switches to CPUs</b>	<b>175</b>
<b>10.1</b>	<b>CPU</b>	<b>176</b>
<b>10.2</b>	<b>Traces</b>	<b>179</b>

## II

## Norman's Principles

<b>11</b>	<b>Knowledge in the World</b>	<b>182</b>
<b>11.1</b>	<b>Memory</b>	<b>183</b>
<b>12</b>	<b>The Principles</b>	<b>186</b>
<b>12.1</b>	<b>Visibility</b>	<b>186</b>
<b>12.2</b>	<b>Discoverability</b>	<b>186</b>
<b>12.3</b>	<b>Mapping</b>	<b>187</b>
<b>12.4</b>	<b>Signifiers</b>	<b>188</b>
<b>12.5</b>	<b>Consistency</b>	<b>189</b>

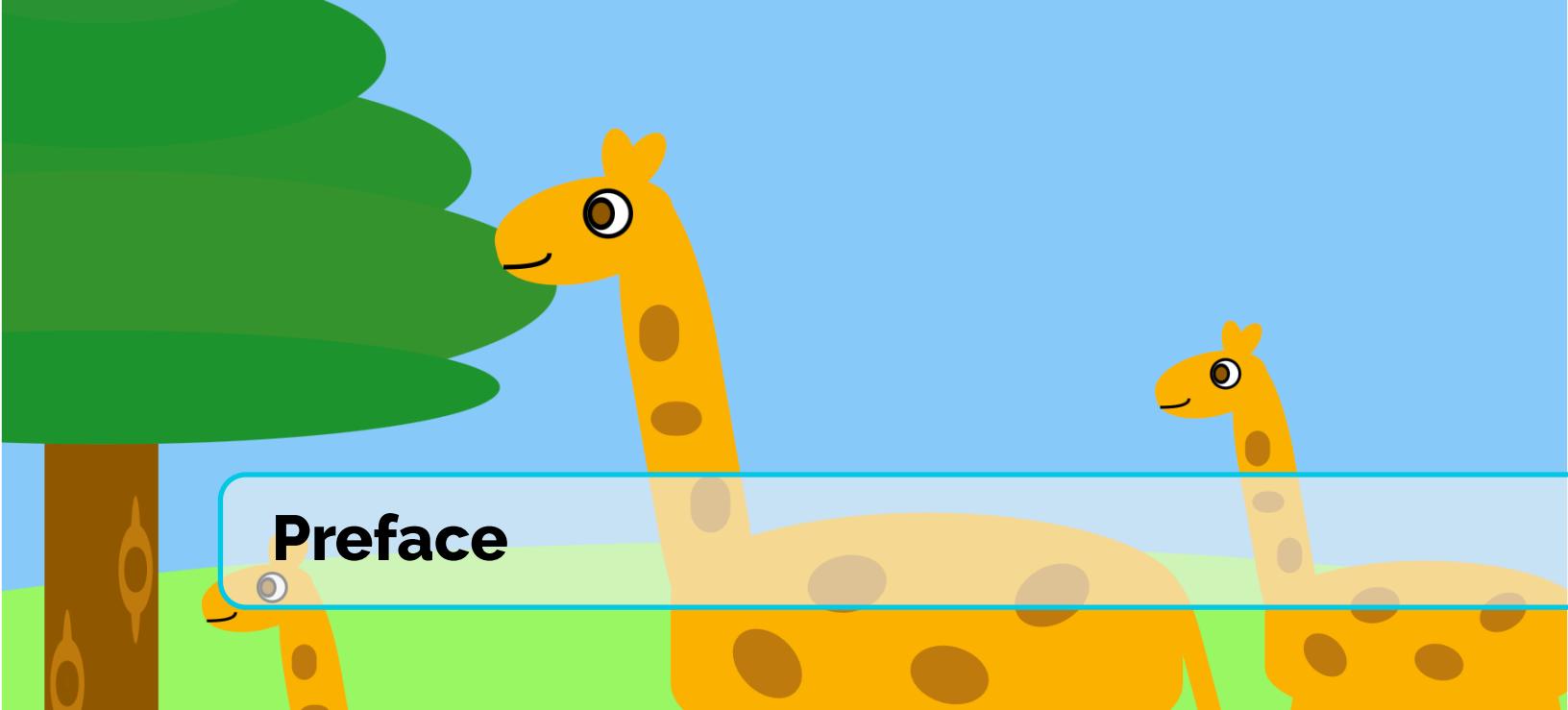
<b>12.6</b>	<b>Constraints</b>	<b>189</b>
<b>12.7</b>	<b>Feedback</b>	<b>190</b>

### III

## Design Thinking

<b>13</b>	<b>History</b>	<b>194</b>
<b>13.1</b>	<b>Herbert Simon and Design Science</b>	<b>194</b>
<b>13.2</b>	<b>George Dantzig and Operations Research</b>	<b>196</b>
<b>13.3</b>	<b>Theory + Practice</b>	<b>199</b>
<b>13.4</b>	<b>The Double Diamond</b>	<b>201</b>
<b>14</b>	<b>Example: This IS your Grandfather's Gaming App</b>	<b>202</b>
<b>15</b>	<b>Design Thinking Templates</b>	<b>290</b>
<b>15.1</b>	<b>Starting the First Phase of Divergence</b>	<b>292</b>
<b>15.2</b>	<b>Researching Possible Project Areas</b>	<b>293</b>
<b>15.3</b>	<b>Choosing a Focus</b>	<b>293</b>
<b>15.4</b>	<b>Interview Preparation</b>	<b>295</b>
<b>15.5</b>	<b>Practice Interview</b>	<b>296</b>
<b>15.6</b>	<b>Evaluation of the Practice Interview</b>	<b>298</b>
<b>15.7</b>	<b>Revising Interview Questions</b>	<b>299</b>
<b>15.8</b>	<b>Interviews and Interview Evaluations</b>	<b>300</b>
<b>15.9</b>	<b>Pivot or Not?</b>	<b>301</b>
<b>15.10</b>	<b>Problem Definition</b>	<b>302</b>
<b>15.11</b>	<b>Symptom or Disease?</b>	<b>303</b>
<b>15.12</b>	<b>Solution Ideation</b>	<b>305</b>
<b>15.13</b>	<b>Prototyping</b>	<b>307</b>
<b>15.14</b>	<b>Feedback</b>	<b>309</b>
<b>15.15</b>	<b>Creating an Action Plan</b>	<b>310</b>
<b>15.16</b>	<b>Comparing Prototypes</b>	<b>311</b>
<b>15.17</b>	<b>Assessing Technical Challenges</b>	<b>312</b>
<b>15.18</b>	<b>Assessing Risks</b>	<b>313</b>
<b>15.19</b>	<b>Peer Feedback</b>	<b>314</b>
<b>15.20</b>	<b>Pitching Your Solution</b>	<b>315</b>
<b>16</b>	<b>Example: Math Visualizer</b>	<b>316</b>

<b>Bibliography .....</b>	<b>367</b>
<b>Articles .....</b>	<b>367</b>
<b>Books .....</b>	<b>367</b>
<b>Index .....</b>	<b>368</b>



## Preface

If you want to learn about software helping people, you have come to the right place! You may already know some programming or be a complete beginner. *Either way, this book is for you.*

It brings together three ingredients: programming, principles of user interface design, and a step-by-step approach to turning an inkling of an idea into software which solves someone's problem.

Beyond the efforts of the authors, this work builds on over a decade of teaching computer science to people from kindergarten to graduate school, both in official courses and unofficial outreach programs. We thank the hundreds of teachers who have welcomed us into their classrooms, the hundreds of McMaster students who have told us when educational theory translates into effective practice (and when it doesn't), and the tens of thousands of children who have poured their energy into a new creative endeavour. Of the many volunteers, we are especially grateful to the campers who volunteered to be mentors and developers: Shireen, Alexandra, Cindy, Eshaan, Mariam, Arhona, Aditya, and the graduates of "Design Like a Girl". We also acknowledge the student leaders, Yumna, Rumsha, and Alyssia, whose work led to McMaster Start Coding, and the earlier students who pioneered evidence-based outreach activities which led us to Elm: Kevin, Curtis, Helen and Tiffany.

Throughout this early development, volunteers were often motivated by one of two things: either they excelled at math which opened doors that they wanted to hold open, or they struggled with it initially, and they wanted to build a better education pathway to let more children through. We call our approach to integrating math and computer science Algebraic Thinking. Math is the best tool for organizing ideas we have, and you will find it threaded through Part I.

Part I is about Elm programming, written in a conversational style with very few assumptions about what you already know, and lots of sample code. At the same time, we plant the seeds for many ideas you will learn about if you pursue a degree in CS, and give you pointers to further reading, to make it easy to dive deeper. If some sections are review to you, take this opportunity to think about how the material is presented, and how you

could have presented it differently. Computer Science is a young field, but there is already some interesting history. If something interests you, but you don't have time during the busy semester to follow up, make a plan to do so in the summer.

It starts with very concrete instructions for getting started, but in the middle shifts gears to talk more about the design of the language and gives tips for organizing your own software. For example, Elm containers (lists, sets, arrays, and dictionaries) are a great example of orthogonal design: following a mix-and-match pattern (pick any function, pick any container) as much as possible. That allows us to give more attention to the non-orthogonal parts which are what makes each container special.

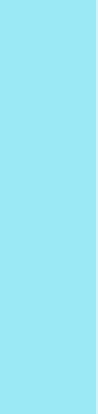
When we discuss the Tower of Hanoi, we give you more than the three-line answers to the two obvious problems, we give you the Divide and Conquer strategy, which you can apply to many problems. And if you follow a known strategy—and there are few better known—and use the established vocabulary in discussing your solution, then other programmers will be able to easily follow your design.

If Part I is held together by mathematical threads, Part II is built on a foundation of memory. Part II is about communicating your app's functionality to a memory-limited human user, following Norman's Principles for interaction design. Some of them will seem almost too simple, others may be confusing. Either way, if you reflect on these principles when evaluating your own work, and other people's apps, you will be able to crystallize knowledge of the principles, allowing you to clearly communicate what you do and do not like about a proposed design. Further practice will then embed that knowledge in "muscle memory" making their application automatic.

Now that we've evaporated math phobias, and explained how to waterproof your app against leaky memory, in Part III, we are ready to tackle poor communication! Many software projects either fail to produce solutions, or solve the wrong problem. Either way, it boils down to poor communication. As a developer, you have to understand your user, but face a huge communication barrier. In some ways, the more you learn about programming, the higher the barrier gets, as it gets harder to imagine how users conceptualize software as black boxes within unknowable insides.

Part III is about understanding user needs, following as simple a template as we were able to make for a process which is necessarily open-ended. Communicating with your users is nothing like writing essays. In fact, it is more about admitting what you don't know, and making your user comfortable enough to tell you. You will learn some of the pre-history of Design Thinking, and the field which came to be known as Design Science, but mostly you will see how that theory is turned into practical steps, each one boiled down to a slide, with tables, graphs, and interview prompts. This is because it is something you can learn best by doing.

Please send comments and bug-fixes to [anandc@mcmaster.ca](mailto:anandc@mcmaster.ca).



# Elm

<b>1</b>	<b>Getting Started</b>	<b>11</b>
<b>2</b>	<b>Let's Get Moving</b>	<b>27</b>
<b>3</b>	<b>Functions and Scope</b>	<b>32</b>
<b>4</b>	<b>To be or not to be in the Basement</b>	
		<b>48</b>
<b>5</b>	<b>More Useful Math</b>	<b>91</b>
<b>6</b>	<b>Core Packages</b>	<b>106</b>
<b>7</b>	<b>Tower of Hanoi</b>	<b>158</b>
<b>8</b>	<b>Composing Music in Elm</b>	<b>164</b>
<b>9</b>	<b>Algebraic Expressions</b>	<b>169</b>
<b>10</b>	<b>Switches to CPUs</b>	<b>175</b>



# 1. Getting Started

## 1.1 Drawing with Stencils

[to contents](#)

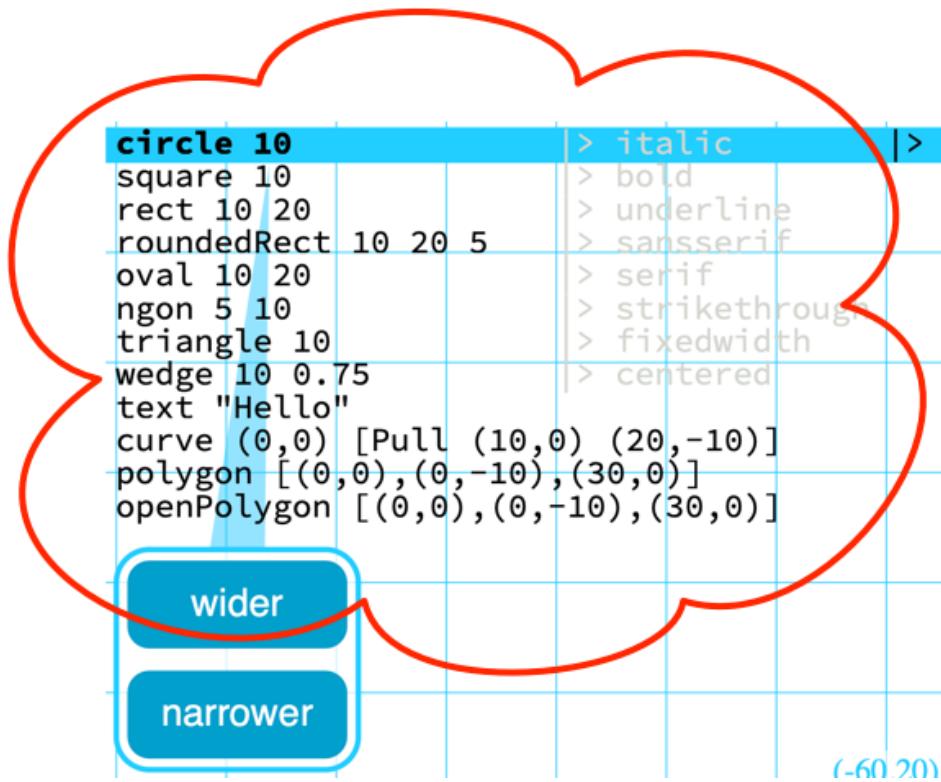
In this book, we will build our knowledge of Computer Science the way we build houses. Buildings need a foundation, usually made of concrete. When learning new ideas, we need to start with what we know: things we can touch, feel, and hear. *Concrete* things! For most of us, sight is our most information-dense sense, and why vision is the sense served by most of our information technology. In this chapter, we'll learn how the GraphicSVG library was built to take advantage of our early experiences with shapes and colour to teach Computer Science. We realize that this is exclusionary to people with visual impairments.<sup>1</sup>

Think back to the first ruler you took to school. It was probably colourful plastic with holes in the middle to help you draw basic shapes: circles, squares, triangles, etc. Those holes are called stencils. We start the same way, with functions to create those “holes” you can later fill in, or outline. But unlike the plastic stencil, our software stencils can be any size you choose. To help you learn them, we've collected them together in the ShapeCreator<sup>2</sup>:

---

<sup>1</sup>We do have an alternate plan—to build a second path out of music supported by keyboard-only tools—but this is a long-term project. In the meantime, see <https://www.youtube.com/watch?v=p2at4S8bLGU> what our first users accomplished with our prototype library for teaching Computer Science through music.

<sup>2</sup><https://macoutreach.rocks/SC3.html>



*Reading Strategy:* If you have access to the internet, you can open it now with the link in the last footnote. Throughout the book, there will be links to sample code and related reading materials. You can either play with the sample code right away, or keep tabs open to experiment with when you get to the end of a section or paragraph.

There are stencils for circles, squares, rectangles, rectangles with rounded edges, ovals, regular (equal-sided) polygons, triangles, wedges, text, curves, and polygons. Triangles are a special type of polygon, so I mostly use the function

```
ngon 3 radius
```

instead of

```
triangle radius
```

but `triangle` is easier to remember for beginners.

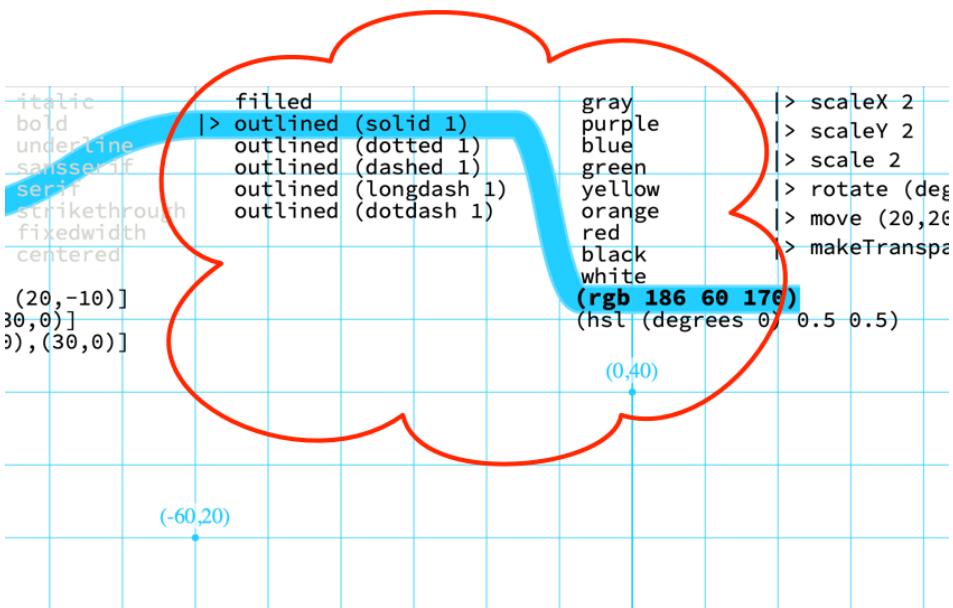
Each of the functions has one or more inputs to control the size or sizes. For example, `circle` has one input, which, you can probably guess must be the radius. When you click on `circle` or use the arrow keys to move the focus there, a pair of buttons pop up labelled “wider” and “narrower”. You can experiment with any of the functions in the ShapeCreator using these tweakers, and their names give away their purpose. For example, if you click on `rect` or `oval` you see a second set of buttons for “taller” and “shorter”, because these stencils are not necessarily as tall as they are wide, unlike `circles` and `squares`. If you remember ( $x, y$ ) coordinates, width before height, horizontal before vertical position, you already know what most of the inputs do. The exceptions are the third input for `roundedRect` (the roundness) and the second input for `wedge` which you tweak with “mouthier” and “mouthiless”. If you are at your computer, go ahead and try tweaking this parameter to see why its

technical name is “mouthiness.” While you’re at it, try all of the stencils so you are familiar with their inputs.

Once you know what these functions do, you don’t need the ShapeCreator, since you could type them straight into the code editor, but if you have worked to get a shape just right, you can copy the code from the “Copiable Code” bubble which is always up-to-date with your latest tweaks:



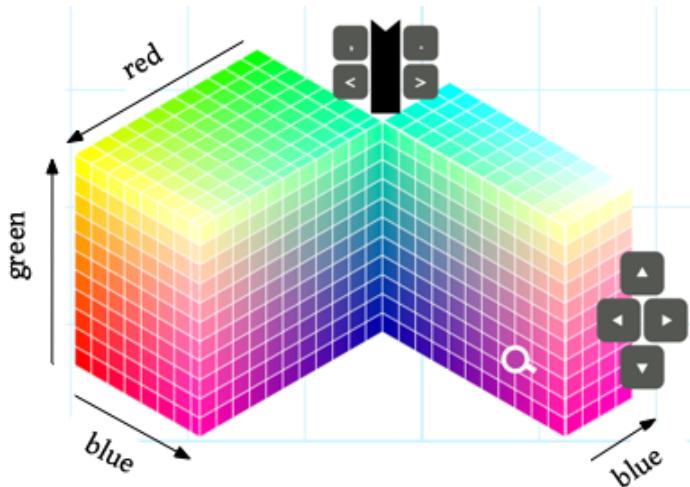
If you have picked a basic stencil like `circle`, you can follow the blue ribbon along to the right where it swerves toward the function you pick to turn that stencil into a shape, either `filled` or `outlined`.



Both of these functions take the stencil as one of their inputs, but first they require the colour to use as an input, and in the case of `outlined` they also require a line type to trace the outline of the stencil. To simplify the display, the line type is grouped with the

`outlined` function, so instead of two choices in the next column, there are six. The colours are all lined up in the next column. You can click on as many colours as you want, but only your latest choice will be accepted. Two of the colours may be hard for you to say, since they lack vowels: `rgb`, `hsl`. That is because they are really acronyms, not colours, and every colour your computer can make<sup>3</sup> can be represented by these functions.

Let's look at these representations! The `rgb` function has three inputs representing red, green, and blue components.



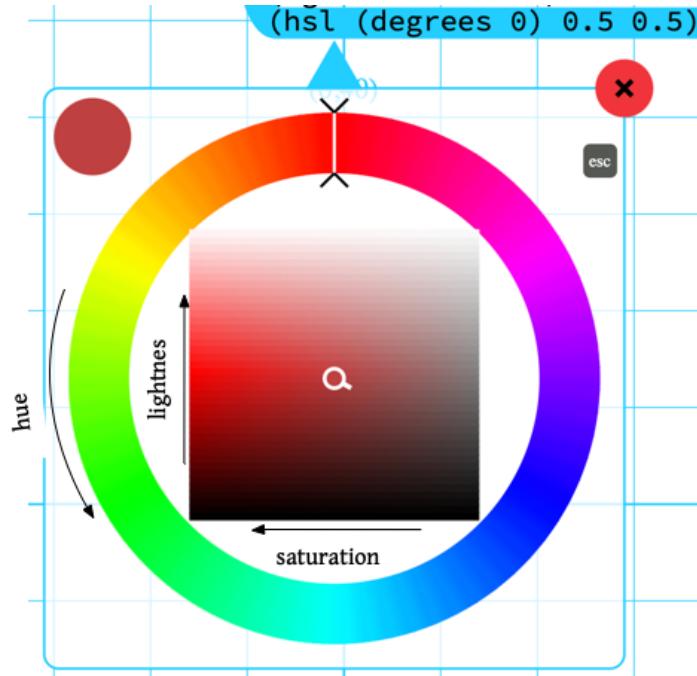
This matches the way most electronic displays work, from antiquated cathode-ray tubes to liquid crystal displays to organic light emitting diodes. This strategy closely matches the sensitivity of the rods embedded in human retinas at the backs of our eyes.<sup>4</sup>

The `rgb` popup is displayed as a book, so you can think of it as flipping through the pages of a book, where each of the pages corresponds to a different amount of blue. The page you open to shows the variation of colour as the amount of green increases as you move up, and the amount of red increases as you go from the spine of the book to the edge of the pages. (The colours are the same on both faces of the page, and you can click on either page to pick a colour.) Notice that white is in one corner of the cube, and black in the opposite corner. White is always visible, but black is only visible if you turn to the first page of the book. Green is on the top of the cube, opposite white, which makes sense since it is what you get

<sup>3</sup>Actually, if you have a more expensive computer, your display may produce 1024 colour values rather than the 256 which was considered good enough a few years ago. You probably won't notice a difference when coding, but you definitely can with high-quality photographs and movies. Here is test video from Netflix, in which you can see bars on the bottom of the gray bar, but not the top if you have a 10-bit ( $1024 = 2^{10}$ ) colour display: <https://www.netflix.com/watch/80018593>.

<sup>4</sup>Interestingly, our retinas have two types of receptors, cones for colour and rods for seeing at night, and the distribution of them varies from the centre of our vision to the periphery, and from person to person. Most people have three types of cones sensitive to red, green, and blue, but people with colour blindness lack one type of receptor, and interestingly, some women have an extra type of cone, giving them a whole new dimension to their vision, and at twilight, both rods and cones are active, so most people have 4-dimensional colour vision to some degree. Wikipedia has a lot more about it: [https://en.wikipedia.org/wiki/Photoreceptor\\_cell](https://en.wikipedia.org/wiki/Photoreceptor_cell)

when you take away all the red and blue from white. The other corners, yellow and cyan, are both secondary colour (mixtures of two primary colours). Around the bottom are two primary colours, red and blue, at opposite corners, with red always visible, and blue only visible when black is. The final corner, magenta, is what you get when you mix blue and red.



Hue-Saturation-Lightness, *hsl*, is *not* how colours are displayed, but is probably closer to how we think of them. In software, we have a few names for the idea of creating an alternative way of thinking about and interacting with data, including *abstraction*, *wrapper*, and *facade*. An abstraction is the most abstract :) and generally means that we hide a lot of complex details behind a simple idea. Rather than figuring out how to mix colours in the right proportions (even though that is what happens), we call up colours by their position in the rainbow. This both matches a universal experience people share, and the physical reality of different wavelengths of light.<sup>5</sup> When we call it a *wrapper*, we are focussing on the actual function that we write to give programmers a more convenient way of accessing core functions. This idea is so useful, experts in object-oriented programming reinvented it as *facade*.<sup>6</sup>

<sup>5</sup>If you don't know how a rainbow is created, have a look at [https://en.wikipedia.org/wiki/Visible\\_spectrum](https://en.wikipedia.org/wiki/Visible_spectrum).

<sup>6</sup>Facade ([https://en.wikipedia.org/wiki/Facade\\_pattern](https://en.wikipedia.org/wiki/Facade_pattern)) is an example of a “design pattern” ([https://en.wikipedia.org/wiki/Software\\_design\\_pattern](https://en.wikipedia.org/wiki/Software_design_pattern)). Functional programmers often say that design patterns translate as functions in languages where functions have more power, like Elm. Their thinking is that rather than explaining patterns with sometimes complex rules, it is better to create functions which can *only* be used the “right” way, eliminating the need for design patterns and their many rules. This is not magic. Instead of learning the rules for the design patterns and best practices for using them, programmers have to remember the functions and sometimes complex types. On the one hand, you

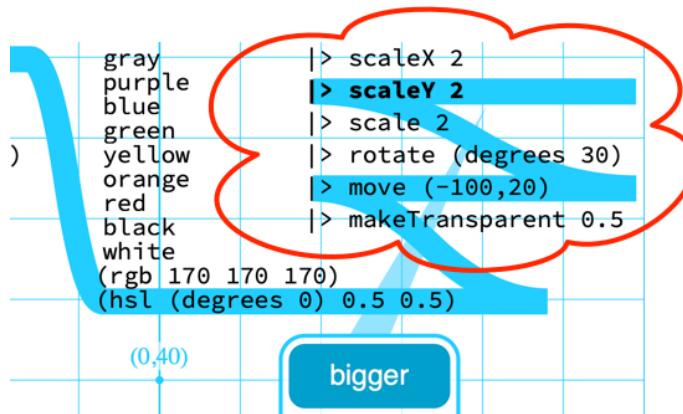
Try it yourself! Find a picture or graphic with a few colours, or pick a few colours to focus on. Try to represent those colours using the two different colour functions. The `hsl` picker only works with mouse or touch, whereas the `rgb` picker also supports keyboard input, which most people find to be more efficient. Set a timer, and see how long it takes with each tool. Do you have to consciously think about how the colours are represented to create your colours? Do your friends have the same experience?

## 1.2 Transformations

[to contents](#)

After filling or outlining a `Stencil`, you have a `Shape`. Note the capital letters! In Elm, types start with capital letters. Types help describe what functions do, and when the Elm compiler processes your program, it can use these types to detect errors, like trying to fill a `Shape` rather than a `Stencil`. We will talk more about types in Chapter 4.

If our library stopped here, all your pictures would be stacks of shapes in the middle of the screen. Not too useful. Fortunately, we also included *transformations* which you may remember from grade 2: translation, rotation, and scaling. We can stretch shapes in one direction using `scaleX` or `scaleY` or in both directions using `scale`. We can rotate shapes about the centre of the window or translate with `move` using Cartesian ( $x, y$ ) coordinates. Finally, we can even make shapes (more) transparent using `makeTransparent`.



You can activate these transformations by tapping/clicking on them or by navigating to them and pressing return or enter (depending on your keyboard). The order of the transformations matters if you have a `move` mixed in, and the ribbon shows you the order in which they are applied, as does the Copiable Code box. If you turn a transformation off, and then turn it back on, it will be moved to the end of the list of transformations. In the ShapeCreator, you can only use one of each transformation, but once you copy your code to the editor, or type it from memory, you can choose to stack up as many as you want in a row. One type of transformation actually requires you to stack multiple `moves`: if you want

can kind of understand a design pattern and still write code, even though it will confuse other programmers and will often fail in some cases. On the other hand, it may be a struggle to write code which will compile, but if it does compile, it will very likely work, and since all uses of the pattern use the same function, they all work the same way.

to rotate a shape around a point which is not  $(0,0)$ , the centre of the screen, you will need to first move the desired pivot point to the centre, then rotate, and finally move back. For example

```
|> move (-20,0)
|> rotate (degrees 45)
|> move (20,0)
```

will rotate by  $45^\circ$  about the point  $(20,0)$ .

Now that you have learned how to make shapes, try creating an emoji in the WebIDE by copying one shape at a time. Log in at <https://cs1xd3.online> with your username and password. Click the **+New** button to open the list of activity types you have available, and choose “Animation v.0” **+Create**. This will open a new module with the starter code

```
// Your shapes go here!
myShapes model =
[  
 ]
```

Clicking the big blue play button—called the compile button—will send the code to the server where it is compiled into JavaScript and sent back to your web browser to be displayed in the pane to the right of your code. Since there is nothing in your `myShapes` function, you will get a blank output. Now copy the shape you created in the ShapeCreator in between the square brackets, `[ ]`. The square brackets define a list, and once you have a shape in it, it is a list with one shape. If you make a mistake, you will get your first compiler error. For example, if you had accidentally selected the closing bracket before pasting, it will be missing in your code, and you will get an error like

```
-- UNFINISHED LIST ----- ID/↔
Test.elm
```

I cannot find the end `of` this list:

```
3| [  
4| circle 10  
5| |> filled (hsl (degrees 0) 0.5 0.5)  
6| |> move (-100,20)  
7| |> scaleY 2  
    ^
```

You can just add a closing `]` right here, and I will be all `↔` set!

If you have not written programs in a textual language, this may come as a shock! Fortunately, the Elm compiler has the most helpful error messages we have seen.

One more thing you need to know is that when you copy successive shapes, you need to add commas to separate them, and, if not, the resulting error would not mention the missing comma. If you had copied the above circle twice, without a comma, you would see:

```
-- TYPE MISMATCH ----- ID/↔
Test.elm
```

The 2nd argument to `scaleY` is not what I expect:

```
7| |> scaleY 2
8| circle 10
     ^^^^^^
```

This `circle` value is a:

`Float -> Stencil`

But `scaleY` needs the 2nd argument to be:

`Shape userMsg`

To understand why you get this error, you need to know what the `|>` does. I'm surprised you haven't asked! We call it a "forward pipe", because the output of the code on the left (lines above) is piped into the code on the right (lines below). The pipeline concept and the use of the vertical bar are considered by programmers who care about our history to be one of the biggest "little" things which make coding easier<sup>7</sup>. We will talk more about pipes in Chapter 3.

Now you are probably happy to learn what the `|>` is doing in your code, but how does that explain the error? Well, the compiler tries to make sense of the code just before and just after the `|>` separately, before seeing if the right-hand side is a function which can process the output of the left-hand side. In our case, it gets stuck on the right-hand side, which, because we forgot the comma, includes the start of the next shape. The right-hand side should be `scaleY 2`, which is a function turning one `Shape` into another one twice as big. If we'd remembered the comma, the compiler would have then checked the left-hand side and found that it was, in fact, a `Shape`, so all would be well. But because there was no comma after the `2` it kept going, found the second `circle`, and tried it out as the input of the function, but `circle` is not a `Shape`, it is a function, so it fails, and puts the `^^^^^` under the `circle` to point out what it doesn't like, and tells you that it has type `Float -> Stencil` instead of `Shape`. If this explanation sounds a bit shaky at this point, don't worry, we will explain types in depth later, and for now, all you have to do is NOT FORGET THE COMMAS IN YOUR LISTS. :)

One way to help remember the commas in your lists is to line up the brackets and commas like this:

```
myShapes model =
  [ circle 10
    |> filled (hsl (degrees 0) 0.5 0.5)
```

<sup>7</sup>See [https://en.wikipedia.org/wiki/Pipeline\\_\(Unix\)](https://en.wikipedia.org/wiki/Pipeline_(Unix)) for pointers to the history of its invention in 1973.

```

|> move (-100,20)
, circle 10
|> filled (hsl (degrees 0) 0.5 0.5)
|> move (100,20)
]

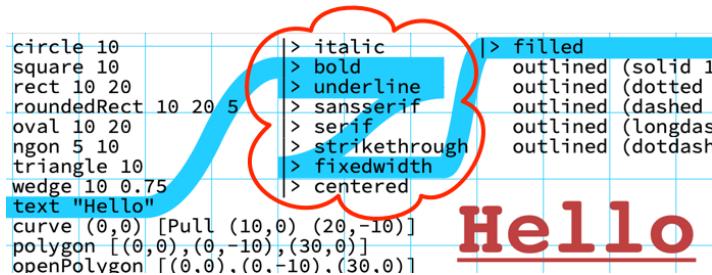
```

Elm is a language where indenting matters, so it is a good habit to indent your code so you can see its structure like this. Most functional programmers do this.

## 1.3 Text

[to contents](#)

There are a few advanced `Stencils` we will skip over in this edition, due to lack of time (including `curve`, `polygon`, and `openPolygon`). You can find more about them in the documentation<sup>8</sup>, but `text` is hard to live without, so let's go over it now.



As you can see, when you select `text` from the list of `Stencils`, another list of transformers appear. You may know transformers as alien robots caught in a struggle of good versus evil, but transformer is also a nickname for *automorphism* (possibly related to autobots). Automorphisms are functions whose inputs and outputs are of the same type. The fact that the input and output sets are the same means that we can apply transformers one after another. In Chapter 3, we will talk more about this property, but if you happen to be a Transformers fan-fiction writer you can get started now on developing storylines for Unital Magma and other concepts not yet mined by the scriptwriters.<sup>9</sup>

You may think we are just playing around with shapes, but we have already discovered two sets of automorphisms! One to transform `Shapes` and one to transform `Stencils`. If you are paying attention at home, you may have noticed a plot hole at this point. The function `bold` makes sense as a transformer of text stencils, but if it is a transformer of `Stencils`, that means it applies to `circles` and `squares` too! What does it mean to make a `circle` italic?

The simple answer is that a function which takes the input and gives it back as the output is a transformer<sup>10</sup>. In general, if we have a function defined on a subset of its output set, we can extend it to a function from the whole set to itself by defining it to return the input for any values not in the original subset.

<sup>8</sup>docs: <https://package.elm-lang.org/packages/MacCASOutreach/graphicsvg/latest/GraphicSVG#notifyTapAt>

<sup>9</sup>For example, the set of automorphisms forms a concrete category (see [https://en.wikipedia.org/wiki/Concrete\\_category](https://en.wikipedia.org/wiki/Concrete_category)) of Monoids (see <https://en.wikipedia.org/wiki/Monoid>).

<sup>10</sup>We call this the `identity` function, which exists in the `Basics` package <https://package.elm-lang.org/packages/elm/core/latest/Basics#identity>.

The more complicated answer is that when we defined the library, we made a choice to make text transformations work this way. Another valid choice would have been to define a special `TextStencil` type with the text transformers acting on this type and not on other stencils. We could then have created separate `filledText` and `outlinedText` functions to turn these special stencils into `Shapes`. We did not do this, because we did not want to create another two functions for beginners to learn, and errors they would inevitably make in applying the wrong function to the wrong type of stencil. In other languages, we could have used two types with a single function name `filled` accepting either stencil type as an input. Note that we said a single function *name* and not a single function. This may seem like a sleight of hand, a meaningless distinction, or fastidious use of the English language.

All of these are true!

And they represent a sweeping under the rug of implementation details. To be an object-oriented programmer is to be used to tripping over the resulting bumps in the rug from similar design choices. To be a functional programmer, using a more fully featured language, is to accept that what seems like straightforward functions like `filled` exist in a multiverse of type classes or generic types, and at any moment the compiler may spit out a type error transporting you to a strange alternate universe.

If you think we must be able to do better, then you are on your way to being a Computer Scientist!

## 1.4 Don't lose that perfect colour!

[to contents](#)

I hope you are already having fun drawing with Elm and especially liking the flexibility of the colour pickers. You could draw a field with a hundred flowers, and each one can have a subtly different shade of purple. For those of you fortunate enough to have visited the Lilac Garden at the Royal Botanical Garden across the water from our home at McMaster University, this is perfect! But if you are instead trying to draw the hundred thousand tulips in Ottawa (originally donated by the Dutch royal family for protecting them during the Second World War), this will take a loooong time, and isn't really necessary, because mass tulip plantings are so impressive precisely because each tulip can have the same colour. In this case, we can get the colours just perfect, and then create our own colour definitions:

```
tulipYellow = hsl (degrees 59) 0.975 0.551
tulipRed = hsl (degrees 351) 0.996 0.557

myShapes model =
  [ oval 4 7
    |> filled tulipRed
    |> move (-80,20)
  , oval 4 7
    |> filled tulipRed
    |> move (-77,20)
  , oval 4 7
    |> filled tulipRed
    |> move (-74,20)
  , oval 4 7
```

```

    |> filled tulipYellow
    |> move (-80,30)
, oval 4 7
    |> filled tulipYellow
    |> move (-77,30)
, oval 4 7
    |> filled tulipYellow
    |> move (-74,30)
]

```

In programming, we call values like `degrees` 59 above *magic numbers*. (That is bad, evil-wizard magic, not good-witch magic.) Any time these numbers appear in your code, you are just asking for a future contributor (including you!) to change some but not all of these values. In the case of colours, this will lead to visual inconsistency in your interface, or weirdness in your picture. More serious is when you compare these values anywhere in your code, in which case changing some values will inevitably break things in ways which are hard to find and hard to fix. So for the sake of your future self, create definitions like `tulipYellow` in your code, and use those values. If your botanist cousin starts bugging you that you got the colour slightly wrong, you only have to change the value in one place, and you will know that the right value will be used everywhere. And when your nosy younger cousin trying to understand your code looks at it, they won't need to ask you what `|> filled tulipYellow` does, because you've made it obvious.

## 1.5 Make Your Own Functions

[to contents](#)

With a little help from your friend the ShapeCreator, you have started using several functions, from `circle` to `makeTransparent`. They take inputs and turn them into outputs. In the case of `circle`, a single numerical input is turned into a `Stencil`. The great thing about using a functional language is that it is easy to create your own functions. You can literally create a function by adding a single letter to a definition. Let's look at the tulip example, because a single oval is not a great tulip! Let's say we improved the tulip and made a definition:

```

tulip =
[ rect 1 10
  |> filled green
, oval 4 7
  |> filled tulipRed
  |> move (0,3)
  |> rotate (degrees 15)
, oval 4 7
  |> filled tulipRed
  |> move (0,3)
  |> rotate (degrees -15)
]

```

Rather than making a new definition for each colour of tulip, we can turn it into a function:

```
tulip c =
  [ rect 1 10
    |> filled green
  , oval 4 7
    |> filled c
    |> move (0,3)
    |> rotate (degrees 15)
  , oval 4 7
    |> filled c
    |> move (0,3)
    |> rotate (degrees -15)
  ]
```

where just adding the `c` between the name of the function and the equal sign turns it into a function with one input, and it lets us use that input in building the output using the variable `c`. Of course, I would never use a single letter for a variable name in real life, and am only doing it to make a point that a single additional letter can turn a simple value definition into a function definition. Notice that everywhere we previously used `tulipRed` we now use `c`.

We can use this function to define our shapes

```
myShapes model =
  tulip tulipRed
```

but what we really want to do is use it to create multiple tulips, like this

```
myShapes model =
  [ tulip tulipRed
  , tulip tulipYellow
  , tulip tulipRed
  ]
```

but that won't work, because `tulip` doesn't create a `Shape` but a *list of them*, which would give us a type error. Fortunately, we have a fix for this...

## 1.5.1 Grouping Shapes

Grouping solves our problem with multiple tulips:

```
tulip colour =
  [ rect 1 10
    |> filled green
  , oval 4 7
    |> filled colour
    |> move (0,3)
    |> rotate (degrees 15)
  , oval 4 7
    |> filled colour
    |> move (0,3)
    |> rotate (degrees -15)
```

```
]
|> group
```

The `group` function takes a list of `Shapes` and creates a single `Shape` which is the sum of the parts. And as a `Shape`, we can apply any of our transformations—just once, because the transformation is applied to all of the component `Shapes` as a group. So the code

```
myShapes model =
  [ tulip tulipYellow
  , tulip tulipRed
    |> move (10,0)
  , tulip tulipYellow
    |> move (20,0)
  , tulip tulipRed
    |> move (30,0)
  ]
```

gives us four non-overlapping tulips.

And since groups are `Shapes`, we can group them together. To make a flower garden, we can group the flowers into rows, then the rows into beds, and arrange the beds into a flower garden:

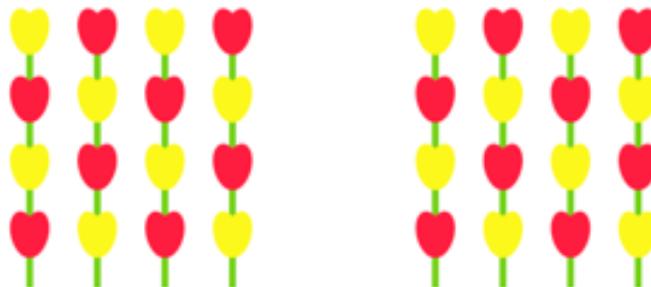
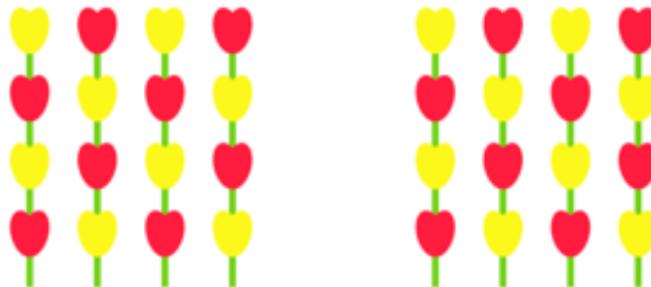
```
tulipYellow = hsl (degrees 59) 0.975 0.551
tulipRed = hsl (degrees 351) 0.996 0.557

tulip colour =
  [ rect 1 10
    |> filled green
  , oval 4 7
    |> filled colour
    |> move (0,3)
    |> rotate (degrees 15)
  , oval 4 7
    |> filled colour
    |> move (0,3)
    |> rotate (degrees -15)
  ]
|> group

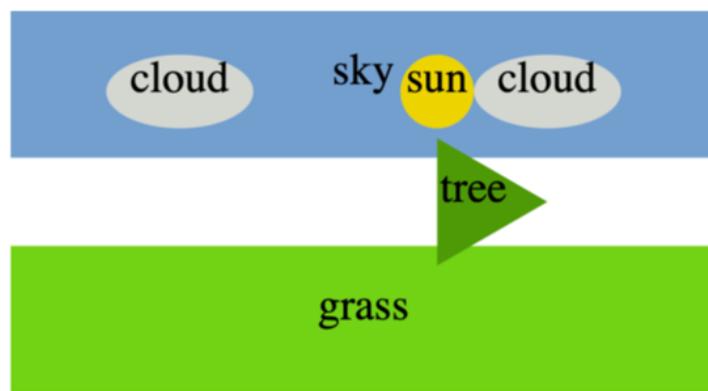
row c1 c2 =
  [ tulip c1
  , tulip c2
    |> move (10,0)
  , tulip c1
    |> move (20,0)
  , tulip c2
    |> move (30,0)
  ]
|> group
```

```
flowerBed =  
  [ row tulipYellow tulipRed  
    , row tulipRed tulipYellow  
      |> move (0,-10)  
    , row tulipYellow tulipRed  
      |> move (0,-20)  
    , row tulipRed tulipYellow  
      |> move (0,-30)  
  ]  
|> group  
  
myShapes model =  
  [ [ flowerBed  
      |> move (-30,-30)  
    , flowerBed  
      |> move (30,-30)  
    , flowerBed  
      |> move (-30,30)  
    , flowerBed  
      |> move (30,30)  
  ]  
|> group  
|> move (-10,10)  
]
```

Note that the `row` function has two inputs, for the two interleaved colours, whereas `flowerBed` is a simple definition (with no inputs), because every flower bed is arranged the same way, but when it calls the `row` function, it alternates the order of the colour inputs in each row. Finally, although it is natural for functions to return grouped shapes, we can create groups on their own without creating a function. Look at the `myShapes` function. It returns a list of `Shapes`, but that list is actually a single, grouped shape. We do this because the way the flower beds are organized, the beds are not symmetrical about the centre of the screen, but by putting them in a `group`, we can use a single `move` to move all of them.



On top of saving you from writing a lot of code—imagine **move**ing every part of every tulip, one piece at a time—using **group** in this way is also an example of an important problem-solving strategy. *Problem Decomposition* is when you break a bigger problem down into parts you can solve, or at least break into smaller pieces of their own. If you want to draw something complicated, break it into parts. A sunset needs a sun, sky, clouds, grass, and a tree. You could start by making definitions for each of those and putting the text for that shape there:



Now you know everything you need, and you can start working on the pieces one at a time. You may think, “No big deal! I could have drawn that anyway.” Unfortunately, it is when we are facing a really tough problem that we are most likely to forget our strategies. By practicing them on simpler problems, we can try to make them a habit which comes to the rescue when we most need it.

## 1.6 Comments and Naming

[to contents](#)

Since we are talking about good habits, we should also talk about comments.

```
{- this is a long comment
   just to say how much we
   like tulips, and that
   we made a function to draw them
   -}

tulip : Color -> Shape msg
tulip petalColour =
  [ -- this is short comment
    -- draw stem first so it is "under" the petals
    rect 1 10
    |> filled green
    -- we want to rotate petals around their bottom...
    , oval 4 7
    |> filled petalColour
    |> move (0,3) -- ...so move them up first
    |> rotate (degrees 15)
    , oval 4 7
    |> filled petalColour
    |> move (0,3)
    |> rotate (degrees -15)
  ]
|> group
```

The first comment doesn't really tell us anything that we need to know and couldn't figure out immediately, but sometimes we do need to write long comments. Sometimes there are many ways of decomposing a problem, or there is no obvious way at all. That is the most important time to write a long comment to save future contributors a lot of time trying to figure out a global pattern by looking at your code line by line.

On the other hand, giving the input variable the name `petalColour` is more useful than any comment we could write about it, because it is not just there at the point of declaration, but is there everywhere you use it. In the early days of computer science when most programmers were one-thumb typists, it was common to use single-letter names. Now, even if you are a one-thumb typist, most code editors auto-complete variable names for you, so you hardly have to type at all.

Some things, however, don't stand out from the code and variable names, like the reason we put the stem earlier in the list of `Shapes` than the petals. This is where short comments can be very useful. Many programmers write short comments for each of the parts of a function before writing the function. The comments are like a to-do list, which you “check off” by writing some code under them. When they finish, they already have useful short comments in the code.



## 2. Let's Get Moving

Creating pictures with GraphicSVG was easy. Even more fun is making animations. The simplest way to think about animations is as a function which has the current time as its input and a `Shape` as its output.

When you create a new `Animation` module in `cs1xd3.online`, you see

```
myShapes model =  
[  
]
```

You learned in the last chapter that the variable `model` is an input to the function `myShapes`. The `model` variable has type `{ time : Float }` which means that it is a *record*, a kind of container, with one thing in it, called `time` which is a number (including fractions). We will learn more about records in Chapter 4, but for now, all you need to know is that we can access it like this<sup>1</sup>:

```
myShapes model =  
[  
  circle model.time  
    |> filled red  
]
```

When you compile it, at first you won't see anything. Then you notice a tiny red circle which grows. After 64 seconds, it touches to top and bottom of the output pane, and after 96 seconds, it touches the sides too. This is because the output pane is always  $128 \times 192$  units, which means that half the height is 64 and half the width is 96. For this, you can work out that the number in `model.time` is the time since the animation started, measured in seconds.

Kind of cool, but what can we do with this? Anything! Well, anything which depends on a number. Try doing the following things:

1. Make a shape move across the screen.

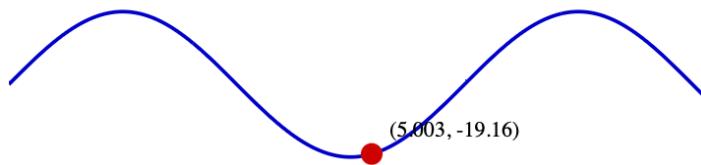
---

<sup>1</sup>grow: <https://cs1xd3.online>ShowModulePublish?modulePublishId=5d44d8e7-a37c-454f-b505-e9e688fd800a>.

2. Make a shape move back and forth.
3. Make a hand wave.
4. Make an eye blink.
5. Make a leaf turn from green to red.
6. Make stars twinkle (not all at the same time).

How many could you do? Could you do the first one? Does it look like this<sup>2</sup>? The problem is that it goes off the screen in a few seconds. You could recompile it every time it goes off the screen??? Ok, not a good answer, but keep reading.

For the second challenge, we can't just use `model.time` but somehow need something to go back and forth. Do you know functions which go back and forth? Well, probably not, because to understand the behaviour of mathematical functions we usually plot them going up and down. Since a lot of what we do in animating is figuring out how to get functions to go up and down just when we need them to, we have a function, `plotGraph`<sup>3</sup> to plot them so you can see them synchronized with your animation, like this:



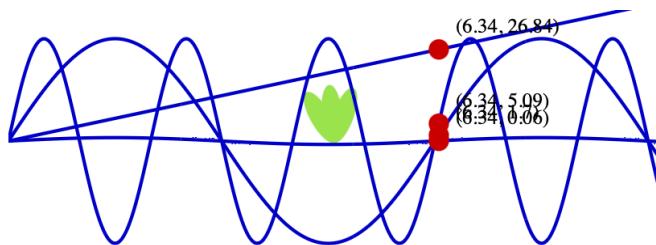
With that hint, you should be able to move back and forth<sup>4</sup>.

If you had trouble with the third challenge, look back at the tulip. It has two petals which are rotated the way a hand waves back and forth. Some cartoon alien hands have two fingers, so you could even borrow the code from there! Do you get something like this<sup>5</sup>?

Before we go on to the next challenge, we should probably talk about the ()'s. Let's plot

```
fun1 t = 30 * sin t
fun2 t = 30 * sin 3 * t
fun3 t = 30 * sin (3 * t)
fun4 t =      sin t
```

which gives us:



<sup>2</sup>(1) <https://cs1xd3.online>ShowModulePublish?modulePublishId=3b339b67-062b-4e1d-8a91-3d420808a32d>

<sup>3</sup>plotGraph: <https://cs1xd3.online>ShowModulePublish?modulePublishId=09875ef2-1dda-4bca-bab2-0ebd6f3af394>

<sup>4</sup>(2) <https://cs1xd3.online>ShowModulePublish?modulePublishId=7df42b71-ebc8-4308-ad29-9aa091ae301c>

<sup>5</sup>(3) <https://cs1xd3.online>ShowModulePublish?modulePublishId=e4a6bc40-7340-4b8e-80e7-d6f949585a28>

To figure out which one is which, you need to know the mathematical meaning of ()s, which is “do insides first”, and you need to know that in Elm, we do functions next, before multiplication and division, which come before addition and subtraction. What trips most people up is that functions in Elm are separated from their inputs by spaces, without any other types of symbols. This means that

```
sin 3 * t = (sin 3) * t
```

and not

```
sin (3 * t)
```

Ok, try to match them up now.

One of the curves in the plot is a straight line. How does that happen when they all have a `sin` which we know wiggles up and down? Did you figure out that `sin 3` is a constant, so

```
fun2 t = 30 * constant * t
```

and any constant multiplied by `t` is going to produce a line.

Now one of the curves is much flatter than the others. Among the remaining functions, two are multiplied by 30, and one isn't. That would be `fun4` which is a squished down version of `fun1`.

That leaves `fun1` and `fun3`. The difference here is that we multiply the input `t` by 3 in one case. Multiplying the input has the effect of scaling in the horizontal direction. Multiplying by 3 squishes it left, so `fun3` has three times as many wiggles. Getting out my wiggle meter, `fun1` has 1.5 wiggles, and `fun3` has 4.5. Checking that with my calculator, it agrees with my trusty, rusty wiggle meter.

That brings us to the blinking eye. Does your eye look like this<sup>6</sup>? If so, I expect you have a headache by the end of the day, because you are squeezing your iris. You might want to try this approach<sup>7</sup>. The problem with this approach is that you need a flat background around the eye, otherwise the `rect` would stick out. There is a better way to do this, in chapter... Hold on a minute, if we tell you where all the good bits are, are you really going to read this whole book? I didn't think so!

How about the leaf? Would this be easier with `rgb` or `hsl`? This would be a good place to use the ShapeCreator to try out the different colours. It's pretty easy to see that `hsl` gives us a pretty good transition as it goes from  $120^\circ$  to  $0^\circ$ , which after darkening just a bit gives this<sup>8</sup>. But if you wait long enough you will see the leaf go blue! How do we fix that? Well, if you try shaking your sleeve, you might find out that `if ... then... else` falls out of it. At least that's what happened to me. This expression

```
...
if condition then
    trueValue
else
    falseValue
```

<sup>6</sup>(4) <https://cs1xd3.online>ShowModulePublish?modulePublishId=0dccdc28a-958b-4122-acbb-ea65944e0f04>

<sup>7</sup>(4') <https://cs1xd3.online>ShowModulePublish?modulePublishId=bbc72775-3b85-45ef-a294-c70117d9a0f6>

<sup>8</sup>(5) <https://cs1xd3.online>ShowModulePublish?modulePublishId=d17ca177-b819-4c37-b66a-1f0550ebfed>

can be used with any combination of three expressions:

- an expression which is `True` or `False`, called the *condition*
- an expression to calculate in the case of `True`
- an expression to calculate in the case of `False`

The last two expressions have to have the type we need in this part of the code. Since we are creating `Shapes`, we could do that:

```
if model.time < 12 then
    oval 40 20
        |> filled (hsl (degrees (120 - 10*model.time))
                      1
                      0.46
                    )
else
    oval 40 20
        |> filled (hsl 0 1 0.46)
```

but you have to look carefully to see that only the hue changes, so it would be clearer to write<sup>9</sup>

```
oval 40 20
    |> filled (hsl ( if model.time < 12 then
                           degrees (120 - 10*model.time)
                         else
                           0
                         )
                  1
                  0.46
                )
```

We can do a lot with `if` expressions, especially when we put them inside each other, which we call nesting. For example, we often need to test a sequence of conditions, such as for an animation which has multiple phases, each starting after another 2 seconds:

```
if      model.time < 2 then
    animation0
else if model.time < 4 then
    animation1
else if model.time < 6 then
    animation2
else if model.time < 8 then
    animation3
else
    animation4
```

Notice how we can make the code easier to read by lining the conditions up? Unfortunately, for the last challenge, the twinkling stars, we cannot use this approach, because stars twin-

---

<sup>9</sup>(5') <https://cs1xd3.online>ShowModulePublish?modulePublishId=952eac1a-46fd-45cf-b517-df510a004ad5>

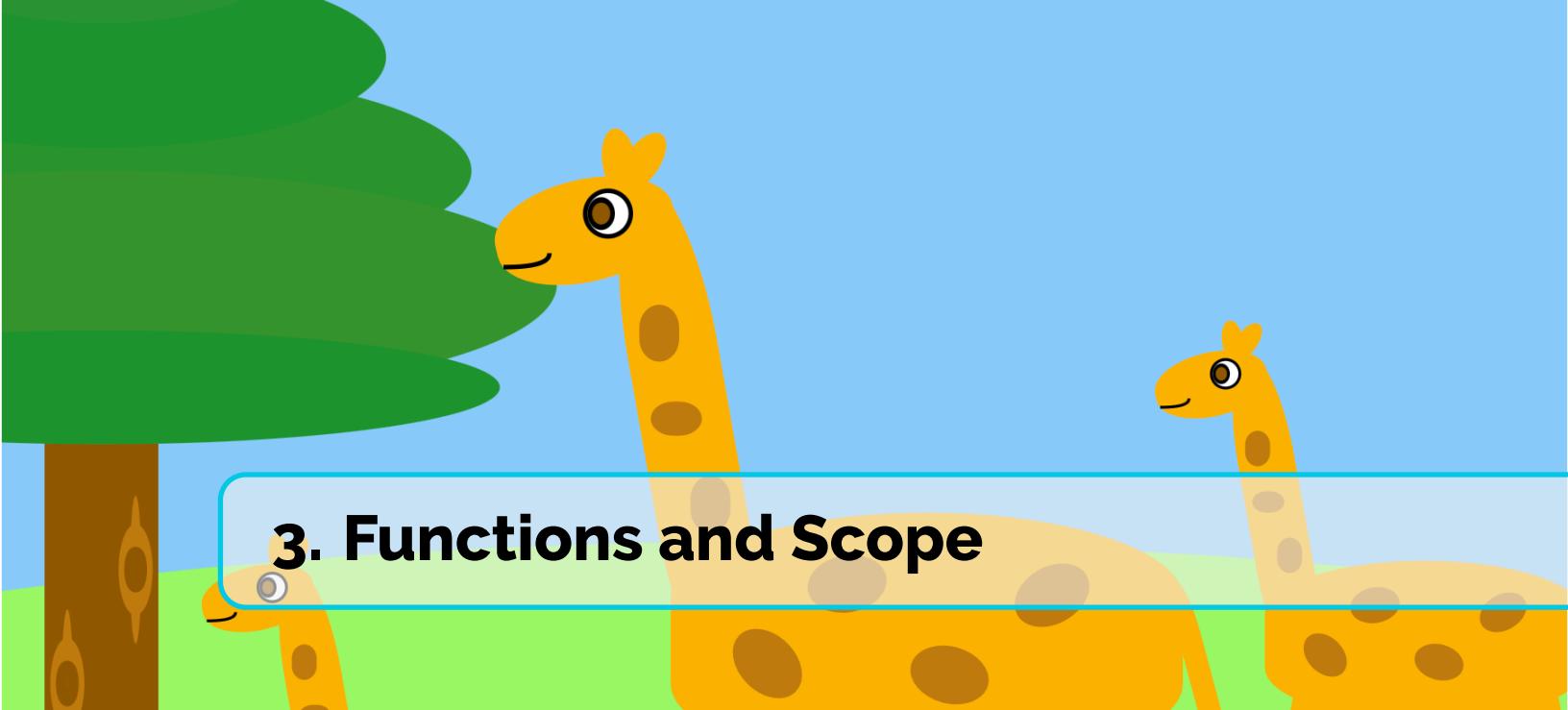
kle forever. But if you think about a function which goes up and down forever, maybe you can figure it out now if you couldn't already do so.

Figured it out yet? Hint: combine `sin` and `ifthenelse` or use `sin` by itself. Here are three ways<sup>10</sup> of having a star twinkle. Did you think of all three? Did you think of other ways? Now pick one of them and figure out how to have them twinkle independently. You may have noticed that the positions you pick for stars don't look right. This could be because you aren't a good random number generator. You can get actual random numbers from [random.org](https://random.org). Later we will learn how to process a list, but for now, I've typed<sup>11</sup> in the numbers from [random.org](https://random.org).

---

<sup>10</sup>(6) <https://cs1xd3.online>ShowModulePublish?modulePublishId=413706ef-3e97-4161-9281-13a9dcb5a07b>

<sup>11</sup>(6') <https://cs1xd3.online>ShowModulePublish?modulePublishId=a0b7f0c0-6cea-4885-a86a-0815d4a40982>



### 3. Functions and Scope

Elm is a functional language. But every practical programming language depends on functions. What makes it a *functional* language and not just a language with functions?

1. Elm functions *are* functions, meaning they return the same outputs every time you call them with the same inputs. This is called *purity*—outputs depend *purely* on the inputs. It is also called referential transparency.
2. Functions are much like other values. The number 2 is a value, but so is the function which doubles its input. For now, you see this in the ease of turning a value definition into a function definition. Later you will learn the flexibility of higher-order functions, which take other functions as inputs.
3. Defined values are defined values, they do not spontaneously change. It is tough enough to remember all the definitions in a big program, but in most languages, you have to learn multiple definitions for each variable, and try to keep track of which one is in effect at the line of code you are trying to understand. This is called having immutable data. If you think of a value definition as a function with zero inputs, then immutability follows from purity, because a function with zero inputs can never have different inputs, so must remain the same.

If you've made it to middle school, you should be surprised that we have to state the first point, because that is what you learned a function does: takes inputs and returns outputs. On Monday, on Tuesday—any day of the week. You have a multiplication table to memorize, because multiplication always returns the same product when given the same two inputs. Unfortunately, most non-functional languages don't actually deliver real functions, and this is a design choice that language developers are trying to undo because it has caused sooo many bugs.

How did this happen? It's important to know a few things. First, “computers” existed before electronic computers, it's just that they were people who shuffled numbers around. Unlike our computers today, the people actually understood what they were doing and could sometimes flag errors, and make suggestions for improving the way computations were completed. Desks, paper, and chalk boards were pretty important to the first comput-

---

ers, so methods of calculating needed to take into account that desks were only so big, and people could only keep track of so many papers at once.

Next, the evolution of computers happened for two slightly different reasons. One, people needed to compute things faster than ever; and two, mathematicians wanted to know if what they were doing made any sense (this breaks down into three parts:

1. did their definitions cover everything (completeness),
2. did they make sense (consistency), and
3. could you actually solve the problems being posed (computability)).

To solve the last part, they developed ways of describing computations, including Lambda calculus and Turing machines. This fed the machine of human curiosity, which being insatiable, led to a whole branch of what we today call computer science. Turing machines evolved into practical blueprints for designing electronic computers. Lambda calculus evolved into functional programming but, unfortunately, only after most people in the field thought they knew how programming languages should work.

The big push to compute faster was driven by the urgency of defeating fascism during World War II. Different machines were developed to solve specific computations, but one person travelled around the United States trying to speed things up by connecting new problems to existing solutions. Of the many problems, the largest was the simulation of nuclear explosions, and this problem was complicated enough that it demanded the development of better methods. The history of this period is both incredibly sad, and also inspiring. It isn't well known because so much about the war was kept secret at the time. There is a podcast about the first programmer<sup>1</sup>.

The difference between answering the theoretical questions and building working computers is that the theoretical questions can be solved by individual mathematicians, and in their papers and letters, we have a record of some of their thoughts; whereas the development of computers required more people than we can recognize, and once we started solving real problems, the number of people grew very fast.

The development of software is in a whole different category, because unlike math, physics, and engineering, there were no professors of computer science—there weren't even words to describe it!

The person who travelled around trying to speed up calculations needed for the war effort was already a famous mathematician, John von Neumann. It is his name that is now attached to the basic design from which today's computers evolved, the “von Neumann Architecture.” His design was based on the Turing machine, and not surprisingly, given the urgency of the war, it focussed on the low-level details of computation. It is not that people did not think about abstract and incredibly important algorithms, but only a tiny number of people were able to translate the high-level ideas into actual programs. As computer applications in science and business grew, few people were interested in rethinking the way people programmed. They only do it whenever there is a crisis<sup>2</sup>: when it becomes just

---

<sup>1</sup>If you are an adult and feeling emotionally invincible today, you can get a real insight into how Klára Dán von Neumann, the first person to write working programs for machines, had to invent things as she went along. If you are not yet an adult, ask one to decide for you. <https://www.lostwomentofscience.org/season-2>

<sup>2</sup>A few select crises: The GOTO problem <https://en.wikipedia.org/wiki/Goto>. Note that this problem is im-

too hard to organize people to write working programs. Crisis by crisis, we seem to keep moving in the direction of functional programming.

What is the moral of the story? Software is not like math or physics. Since we are still learning the best way to do things, and we should probably wait another 50 years before trying to teach people. :) If you don't have 50 years to wait, learn to love learning!

At some point, you will definitely be puzzling over a new software concept, and wonder why you couldn't just keep doing things the way you were doing them. To paraphrase an idea I learned from Cynthia Solomon who attributed it to Seymour Papert (both pretty amazing people in computer science education): Imagine if European scientists had said, "You know what? Don't fix what's not broken. Let's keep using Roman numerals." (I, II, III, IV, V, ...) Actually, I don't really think you can imagine how hard your job as a student would be today if we were still using that clunky system. You should be supremely grateful that they decided to give Hindu numbers a chance. Of course, we can be somewhat thankful to the Persian and Arab textbook writers, and the Jewish translators too. :)

You just don't know what new concept is going to have that power until you learn it.

## 3.1 Foundations

[to contents](#)

Some questions are valuable because of the thinking that goes into the answer, like "What came first, the chicken or the egg?" which may lead to us discovering evolution. Mathematicians had a similar question about a century ago: "What came first, the set or the function?" Well, that is not how they state it, but mathematicians started seriously asking, "Can mathematics be built out of a small number of building blocks?" This was a philosophical question until digital computers were invented which, for practical reasons, represent everything in terms of a small number of building blocks (true, false, and logical operations). It turns out that we can build mathematics up from building blocks, but it has to be done in stages, with every stage building more complex objects from the objects of previous stages. Bertrand Russell and his teacher Alfred Whitehead became famous a century ago for building up much of known mathematics from sets and logic, avoiding contradictions in previous attempts to build mathematics up from sets alone.

But what did come first? Sets or functions? It turns out that there are alternative ways of building mathematics, starting with sets, starting with logic, or starting with functions. It is helpful to know that there are multiple starting points, meaning that we can define one in terms of the other and vice versa. Since this can be done logically without creating contradictions, it is ok to think about our programs in terms of functions, in terms of objects, or in terms of bits and bytes (groupings of the true and false values we know all our digital<sup>3</sup> computers are built upon). Complex problems are a *lot* easier to understand using

---

possible in functional programs. The "Mythical Man-Month" problem [https://en.wikipedia.org/wiki/The\\_Mythical\\_Man-Month](https://en.wikipedia.org/wiki/The_Mythical_Man-Month). Note that this problem only arose after men took over programming from women. :) The painfulness of writing or using with user interfaces led to the development of Smalltalk. <https://en.wikipedia.org/wiki/Smalltalk> This first successful Object-Oriented language meant everyone wanted to use OO design everywhere, which precipitated a new crisis leading to Design Patterns [https://en.wikipedia.org/wiki/Design\\_Patterns](https://en.wikipedia.org/wiki/Design_Patterns) which essentially tell you not to do things in OO languages, except for their short list of sensible patterns.

<sup>3</sup>We will omit analogue and quantum computers from this book, although they are both very interesting.

the language of functions. Constructing the fastest possible solution to a problem, on the other hand, is difficult to do without looking into the bits and bytes. One definition of a good computer scientist is someone who can identify the right view to take to solve the current problem efficiently, and even look at the same problem through different lenses when required. Of course, it is human nature not to do this! Because of the difficulty we have in accessing our long-term memories, we naturally prefer to keep reusing the same concepts we used for our last problems.

Ok, the chickens are tired of us hanging out in their coop, and want us to get back to Elm programming.

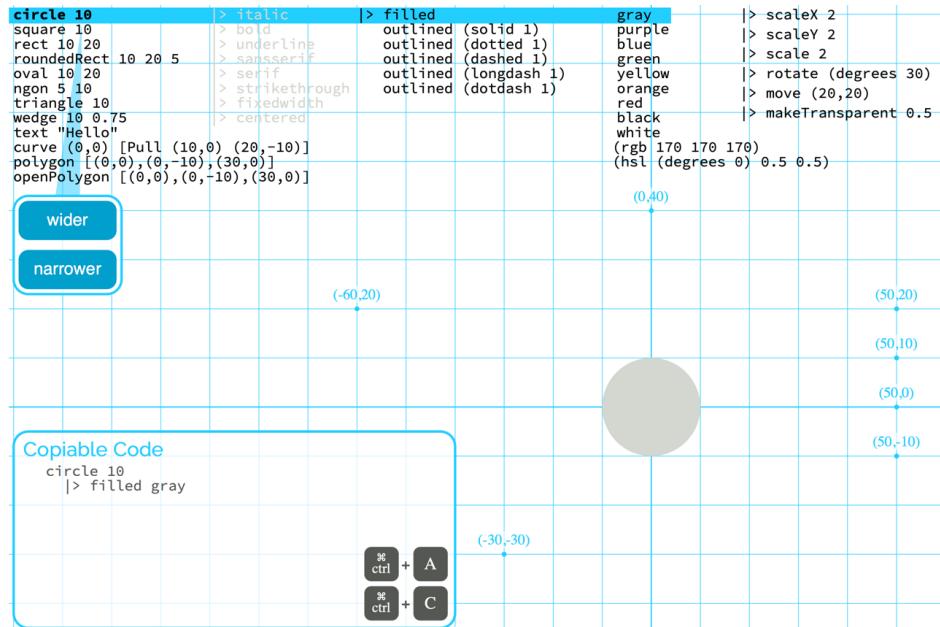
## 3.2 Types

[to contents](#)

If you are thinking that Elm is a functional programming language, so surely functions come first, you would be right! From our very first examples, we started using functions without really talking about them. In Elm, we use notation to give functions a type, like

`circle : Float -> Stencil`

which means that `circle` is a function which takes a `Float` (number) input and gives back a `Stencil`. The types are vitally important to understanding many problems, and knowing what we do about Russell's work, it is reassuring to know that types are there protecting us from certain mistakes<sup>4</sup>. But if you think back to using the ShapeCreator, the focus was on the functions and how they can be composed.



Embedded in the code created by the ShapeCreator was forward pipe `|>`, which we explained to mean that we take the value on the left and feed it to the function on the right. We call it a pipe in analogy to a processing plant, like a water filtration plant, which takes in

<sup>4</sup>Russell's Paradox: [https://en.wikipedia.org/wiki/Russell's\\_paradox](https://en.wikipedia.org/wiki/Russell's_paradox)

dirty water, pumps it through pipes to filters and sedimentation tanks, and finally through the city and into our homes. The pipe is itself a function,

```
(|>) : a -> ( a -> b ) -> b
```

which means that it takes a first input of some type `a` and a second input of type `a -> b` (i.e., a function) and returns a value of type `b`. We can use it in the “normal” function order by writing

```
(|>) (circle 5) (filled red)
```

This matches the type definition, but it is harder to read. We could eliminate it by writing

```
filled red (circle 5)
```

but this deemphasizes the flow, and gets harder to follow as parentheses are added:

```
move (-10,-10) ( rotate (degrees 45) ( move (10,10) ( filled red ←
  (circle 5) ) ) )
```

which is why the ShapeCreator generates it as

```
circle 5
|> filled red
|> move (10,10)
|> rotate (degrees 45)
|> move (-10,-10)
```

Elm also provides us with backward pipe

```
filled red <| circle 5
```

which does save the parentheses, does match the most common order for parentheses in mathematics but, when written on multiple lines, puts the `Stencil` farther and farther away from where you start reading. You are free to use it, but then you are also free to walk around on your hands instead of your feet. We like to do things the easy way.

You are probably pretty impressed by this leap forward in notation, almost as impressive as switching from Roman to today’s Hindu-Arabic numbers. But functional languages include even more powerful abstractions which allow us to combine functions, without bothering with values. This is usually referred to as “point-free” notation because the idea arose in the new geometry developed in the late 19th century, which we now call (algebraic) topology<sup>5</sup>. Topology asks what we can know about spaces other than the 3D space we live in. We will talk more about the subset of spaces called vector spaces when they ride in to the rescue in Section 5.4. For now, it is enough to know that, to answer the question “How many spaces are there?” they needed a way of equating spaces, and functions which take an input from one space and return an output in a second space are the natural way of comparing spaces. If a second function exists which takes every output of the first function as its input and returns the original input of the first function as its output, we say it is the (left) inverse function. If we were naming it today, we would probably call it an “undo” function,

---

<sup>5</sup>See [https://en.wikipedia.org/wiki/Algebraic\\_topology](https://en.wikipedia.org/wiki/Algebraic_topology).

but they didn't have undo, or even control keys, back then. Before the invention of point-free notation, functions were defined by algebraic expressions in terms of their coordinate variables. Some computations need to be done with coordinate variables, but it took a long time to figure out that some answers were independent of the choice of coordinate system. Point-free notation made it much easier to see these things.

Point-free notation plays the same role in program synthesis. We often need to use variables bound to input values to express our functions, but sometimes they just create too many trees to see the forest. In mathematics, when we have two functions  $f : X \rightarrow Y$  and  $g : Y \rightarrow Z$ , where  $X, Y, Z$  are sets,  $g \circ f$  is defined to be the function

$$(g \circ f)(x) = g(f(x)). \quad (3.1)$$

In Elm, we can do the same thing. If `f : X -> Y` and `g : Y -> Z` then we can define `g << f` to be the function

```
gof x = g ( f ( x ))
```

which is equivalent to the definitions

```
gof1 x = x |> f |> g
gof2 x = f x |> g
gof3 x = g <| f x
gof4 x = g <| f <| x
gof5 x = (g << f) x
gof6 x = (f >> g) x
```

Did you notice we snuck in `(>>)` which composes functions from left to right instead of from right to left? There are a lot of ways to connect two functions together. You can see that `(<<)` and `(<|)` match the order function names take when composed using nested parentheses, but we prefer to use `(>>)` and `(|>)` which match the reading direction in English.

## 3.3 Partial Function Application

[to contents](#)

Functional languages like Elm give us a few more ways of making functions out of old functions. Partial Function Application allows us to take a function with multiple inputs and make a new function with fewer inputs. Most of our `Shape` transformations use this approach. Let's look again at

```
circle 5
|> filled red
|> move (10,10)
|> rotate (degrees 45)
|> move (-10,-10)
```

Here are the types of the transformers:

```
filled : Color      -> Shape -> Shape
move   : (Float,Float) -> Shape -> Shape
rotate : Float       -> Shape -> Shape
```

In a case like this, where there are no parentheses in the type signatures, the implicit parentheses go right to left, i.e.,

```
filled : Color      -> ( Shape -> Shape )
move   : (Float,Float) -> ( Shape -> Shape )
rotate : Float       -> ( Shape -> Shape )
```

This is called associativity, and is the same reason that  $4 - 5 + 7 = (4 - 5) + 7$  and not  $4 - (5 + 7)$ , which in this case don't give the same answer! For arithmetic, we associate left to right, meaning that we add the parentheses starting on the left and working outwards. Function signatures, on the other hand, are right-associative, meaning that we add parentheses on the right.

What this means is that `filled` can be thought of as a function which takes one `Color` input and returns a function `Shape -> Shape`, e.g.,

```
(filled red) : Shape -> Shape
```

All of the transformations are designed this way, because if we have functions

```
fun1 : Shape -> Shape
fun2 : Shape -> Shape
fun3 : Shape -> Shape
```

then

```
(fun1 >> fun2) : Shape -> Shape
(fun2 >> fun3 >> fun1) : Shape -> Shape
(fun3 >> fun3 >> fun3 >> fun2 >> fun2 >> fun1)
                           : Shape -> Shape
```

Every composition of functions of this type makes sense! We call functions `a -> a`, with one input of the same type as the output, endomorphisms<sup>6</sup>. Endomorphism combines Greek roots *within* + *shape*. Early topologists discovered that a lot could be said about mathematical objects just by knowing about the existence of functions between them, which spawned a new branch of mathematics called Category Theory. Category Theory was picked up by computer scientists<sup>7</sup> as an alternative to set theory or logic as a tool for understanding (and proving properties like correctness) about them.

How is this useful? Every year, a couple of students will ask something like “Why do we have to bother with the insides, we can tell what functions do just from their type signatures?” In fact, this is true in some cases, and this insight parallels the insights of the first topologists. So *Bravo!* to those students.

To really appreciate this, you need to learn some of the core Elm packages, especially `List`, but let's look at a little example using what we know. Imagine you are creating a paint-by-numbers app. You want to draw some shapes both as outlines and as filled-in shapes, depending on the choices the user makes. We could draw our shape, and then copy and paste the code so we can change the `filleds` into `outlineds`, but then if we ever change the original drawing, the outlined version will not be in sync. This is a violation of a

---

<sup>6</sup>See <https://en.wikipedia.org/wiki/Endomorphism>

<sup>7</sup>See <https://mitpress.mit.edu/books/basic-category-theory-computer-scientists>.

principle of information science called the Single Source of Truth<sup>8</sup> (SSoT), which says that you shouldn't have the same information stored in multiple places, and is pretty important for large systems. So, instead, we can define the parts which will not change:

```
myStencil = square 10
myColour = rgb 25 150 255
myTransform = move (20,30) >> rotate (degrees 20)
```

and then we can use them in two ways

```
theOutline = myStencil
            |> outlined (solid 0.5) myColour
            |> myTransform
theColouring = myStencil
              |> filled myColour
              |> myTransform
```

But this still involves duplication of our definitions, so we could define

```
draw isFilled stncl clr trans =
  stncl
    |> (if isFilled then
        filled
      else
        outlined (solid 0.5)
    ) clr
    |> trans
```

which reduces some of that additional typing, and is useful if we are storing a `Bool` value for the state of this element.

## 3.4 Operators versus Functions

[to contents](#)

Another way that we know that functions are overwhelmingly important to mathematics is the number of different names for them. In programming, you will run across *functions*, *operators*, *combinators*, and *lambda expressions*. Operators are just another way of writing functions, called *infix* notation. Since one of the first functions we learn is `+`, as in  $1 + 1$ , we are used to writing functions this way, but did you ever stop to think that it is only possible to represent functions with two inputs this way?! In another universe where the most important functions have more than two inputs, we would probably write algebra in a totally different way. When we don't want to write these functions using infix order, Elm gives us a way to refer to them as “normal” functions,

```
addTwo x y = (+) x y
```

Often we want to do this, because we want to partially apply a function to create a new one:

```
multByTwo = (*) 2
```

which of course is the same as

---

<sup>8</sup>See [https://en.wikipedia.org/wiki/Single\\_source\\_of\\_truth](https://en.wikipedia.org/wiki/Single_source_of_truth).

```
multByTwo x = (*) 2 x
```

but with less typing. Now, I personally do not recommend using this style everywhere, because it is not visually obvious that `multByTwo` is a function taking one argument, but we will see many useful applications in later chapters.

## 3.5 Anonymous Functions

[to contents](#)

To complete the list of ways in which we can define functions, we need to introduce the method which cannot be named, because it is anonymous! We call these functions *anonymous* because they get defined without giving them a name.

You are used to defining

```
flower petal =
  [ oval 20 10 |> filled petal
  , oval 10 20 |> filled petal
  , circle 10 |> filled darkBrown
  ]
  |> group
```

Although it seems silly, we could separate the naming of the function and the definition

```
flower =
\ petal ->
  [ oval 20 10 |> filled petal
  , oval 10 20 |> filled petal
  , circle 10 |> filled darkBrown
  ]
  |> group
```

The notation `\...->...` means “a function maps input ... to output ...” Why do we use “\” for this? We do this because most people don’t know how to type the Greek letter  $\lambda$  (lambda) on their keyboards, and it looks a bit like that. We want to use  $\lambda$  because Alonzo Church used it to represent the binding of a variable to an expression to define a function, which is the basis for Lambda calculus. Although Lambda calculus and Turing machines are equivalent ways of defining computation, nobody told Hollywood, because Turing has a movie<sup>9</sup> but Church doesn’t<sup>10</sup>.

## 3.6 let ... in ...

[to contents](#)

Anonymous functions can be a handy way for defining functions which are only ever used once, but even if you only use a function once, it is often helpful to name it, so that other people know what it does. Generally, a function you can define in a fraction of a line can be understood by others without breaking up the flow of reading the surrounding code,

<sup>9</sup>...with many inaccuracies [https://en.wikipedia.org/wiki/The\\_Imitation\\_Game](https://en.wikipedia.org/wiki/The_Imitation_Game)

<sup>10</sup>It’s not a movie, but Lambda calculus is more popular among programming language enthusiasts who meet at a website called [lambda-the-ultimate.org](http://lambda-the-ultimate.org), which is a good place to find out what is cool in programming languages.

but anything longer *will* interrupt the flow of reading. So if other programmers could read your code better knowing what the function does from its short but descriptive name, then you should name it. How do we do that, without creating a function we don't want other parts of the code to use? This is one of the things `let ... in ...` is good for. For example

```
sortSpecial lst =  
  let  
    compareFun = ... -- really long and complicated  
                  -- function to sort data structure  
  in  
  List.sortBy compareFun lst
```

lets us define a function only used in this one sort. The advantage here is that the line `List.sortBy compareFun lst` is readable by itself, and you don't have to give yourself a headache parsing the complicated function to do the compare every time. Other great properties of `let ... in ...` are that

- You can define multiple values and functions which depend on each other.
- You can use (main) function arguments here without having to add them as arguments to every locally defined function, as you would if they were on the top level.
- You can hide these definitions from other parts of your code, even in the same function. This concept of restricting access is so important, we will now devote a whole section to it...

## 3.7 Modules

[to contents](#)

Lambda calculus and Turing machines were great for defining what computation is, and if programming languages were just about telling computers how to do computations, we could skip the rest of the book! But as we have said, programming languages are just as much or even more about communicating with other programmers, and organizing our ideas so we can remember them. This is why we keep developing new programming languages and design principles. Let's look at two principles first and how we implement them.

*Information Hiding*<sup>11</sup> (IH) and *Separation of Concerns*<sup>12</sup> (SoC) are closely related. Imagine you are working with a group on a large application, bigger than what you can manage alone. You are working on one part of the application and making good progress, when suddenly your code stops working, because one of the functions you are using disappears. You investigate and find out that your friend “made it better” without telling anyone. Actually, it was better for *her* use, but you didn't expect it to work that way. Now you've lost half a day figuring it out.

SoC is the principle that you should not try to solve whole systems at once, but instead break them down into pieces—wait a minute, isn't that just a problem-solving strategy? Yes, it is a great problem-solving strategy which we like to call Divide and Conquer, but it is not enough for programmers. That is because, unlike a machine, which is finished when it is all screwed together and delivered to the customer, we can just keep adding features to a

<sup>11</sup>[https://en.wikipedia.org/wiki/Information\\_hiding](https://en.wikipedia.org/wiki/Information_hiding)

<sup>12</sup>[https://en.wikipedia.org/wiki/Separation\\_of\\_concerns](https://en.wikipedia.org/wiki/Separation_of_concerns)

program, so it is never finished in the same way. Because programmers can innovate at the speed of light, we need speed limits, and SoC adds a layer to D&C by requiring that each piece of the software have a stated purpose and that changes be made to that particular piece only if they maintain the purpose, or if everyone depending on that piece of software agrees to change the stated purpose.

Other scientists and engineers have awesome-sounding names for the things they work on, like “manifold”, “subassembly” and “warp coil”. So we call these pieces “modules”<sup>13</sup> (when designed to support SoC). You will find on your journey that it is a lot easier to say that “Of course, we are *totally modular*,” than it is to figure out the best way to divide up your software into pieces. Modularizing is like putting up fences. They may make some things slower—making you walk a bit farther, or adapting to the given library interface rather than just changing the library code—but done well they prevent a lot of headaches—like people walking over your vegetables, or other programmers changing the name of the function you were using.

One of the great advantages of more recent programming languages is built-in support for modules. In fact, software modules have existed from the early days of programming, but nothing was preventing you from violating their boundaries. Today, I think most programmers know that variable names created in one module should not be available in other modules, even if they don’t know why<sup>14</sup>. Many would say that you should have to jump through some hoops to make them available, so that you think twice about exposing the inner workings of your module to other people, because as soon as you do that, someone will find an unexpected way of using those variables in a way you do not expect, thereby making it harder for you to improve how your module functions internally without breaking their code. This process of hiding as many details of *how* your module works as opposed to *what* your module does, is called Information Hiding.

End of story? Not quite. Just like stomping out a biological pandemic, or an epidemic of malware, there is no single solution which will eliminate errors caused by adding new features. The good news for software designers is that unlike our DNA, which is vast and mostly a mystery, people do write all of our software, so we can define. We do not need to rely solely on the module system. We can develop new languages and practices, and we have. One of them is functional programming.

How does functional programming help protect us against unwanted side effects of adding a new feature? Well, by eliminating side effects! Not the side effects of changes to the software, but side effects of functions. As we have said many times, a function is something which takes an input and produces an output. This is true of all languages, but in languages without pure functions, the input is potentially all the data in the program and the output is potentially all the data in the program. What?! How can anyone think about all the data in the program and know if only the expected changes are being made? The answer, as anyone who has had to face this type of debugging challenge in a large C, C++, or Java program knows, is that you have to put on your detective hat and start experimenting.

---

<sup>13</sup>See [https://en.wikipedia.org/wiki/Modular\\_programming](https://en.wikipedia.org/wiki/Modular_programming).

<sup>14</sup>For example, see this thread <https://stackoverflow.com/questions/10525582/>

“Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth<sup>15</sup>. ” The unfortunate thing is that nobody writes novels about our detection, and we never get movie royalties, although many of us end up solving more difficult problems than Sherlock ever faced!

So it is much better to use what computer scientists call *pure functions*, and what math teachers call *functions*, with clearly defined inputs and outputs. As great as the principles of Separation of Concerns, Information Hiding, and Modularization are, as long as our functions are not pure, we keep building time bombs (technically called Single Points of Failure<sup>16</sup>) into our code, and rather than making our code safer, those other principles amount to a bunch of paperwork weighing down developers.

Whoa! That was a lot of Wikipedia links. Since this is a free textbook, hopefully, some of you will have money left over to donate to Wikipedia, which is a whale<sup>17</sup> of a resource. After that non-commercial break, we can move on to scope.

## 3.8 Scope

[to contents](#)

Scope actually predates a lot of the deep thinking above, because it was needed to solve a simpler problem: running out of variable names. Scope limits the parts of a program where a variable can be used. Really big programs need *many* definitions, and if all of those definitions are usable in all parts of your program, then you will end up with very long variable names, like `numCats`, `numCatsInFunctionF` `secondNumCatsInFunctionF`, and so on<sup>18</sup>

Scope is easy to see at a glance in Elm, because of indentation.

1. Definitions are *in scope* in all other definitions and expressions which start at the same level of indentation, and parts of those definitions with deeper indentation.
2. Variables defined by binding the inputs of a function are in scope in the function.
3. Variables defined in a pattern of a `case` expression are in scope within the result of that case.

What is a `case` expression? Let’s start with what we know. It is a more general condition than `if-then-else`. If you wrote:

<sup>15</sup> Arthur Conan Doyle’s Sherlock Holmes said this <https://www.goodreads.com/quotes/7471034-once-you-eliminate-the-impossible-whatever-remains-no-matter-how>.

<sup>16</sup> See [https://en.wikipedia.org/wiki/Single\\_point\\_of\\_failure](https://en.wikipedia.org/wiki/Single_point_of_failure).

<sup>17</sup> Read founder Jimmy Wales’ reasons for donating: [https://donate.wikimedia.org/w/index.php?title=Special:LandingPage&country=XX&uselang=en&utm\\_medium=sidebar&utm\\_source=donate&utm\\_campaign=C13\\_en.wikipedia.org](https://donate.wikimedia.org/w/index.php?title=Special:LandingPage&country=XX&uselang=en&utm_medium=sidebar&utm_source=donate&utm_campaign=C13_en.wikipedia.org)

<sup>18</sup> These are fun names, but not historically accurate, because the first computers were so short of memory that we could not afford such long variable names, or we wouldn’t be able to run the compiler. As computers gained memory, and we could afford decent compilers, the gender balance in computer science shifted, with the majority of women “computers” left over from the war years when they had started out computing artillery tables and logistics using operations research were displaced by men. Not only were the men paid more to do the same work—actually less work, because at that time typing was considered a lower-class skill suitable for women, but not for the male Engineers who replaced them—they also cemented a bad habit of using short, inscrutable variable names like `i` and `j`, as they struggled to write programs with their one-finger typing. But, to be fair, since the inability of men to type did create an imperative to add scoping to programming languages, it is perhaps a rare positive side effect of sexism.

```
if condition then
    trueExpression
else
    falseExpression
```

you could have written

```
case condition of
    True  -> trueExpression
    False -> falseExpression
```

That is a special case of matching a `Bool` condition.

If your character can only count to three, you can use this snippet to generate their dialogue from a `num:Int`:

```
case num of
    0 -> "zero"
    1 -> "one"
    2 -> "two"
    3 -> "three"
    _ -> "more"
```

Note that the `_` matches any value, and indicates that you won't use the matched value. Since you could use `num`, you didn't need the matched value here anyway, but you can also use it within patterns, as we will see.

If you define your own type:

```
type Residence
  = House      Int String      -- number and street
  | Apartment Int String Int -- ... and apartment number
```

Then you can build `case` expressions to match. Let's say you live in a village with one street, and you want to calculate delivery time, starting at your location, based on address and possibly climbing stairs, knowing that the only apartment buildings use the 10s digit for the floor.

```
case residence of
    House streetNum _
        -> 0.5 *toFloat ( abs (streetNum - restaurantLoc) )
    Appartement streetNum _ aptNum
        -> 0.5 *toFloat ( abs (streetNum - restaurantLoc) )
        + toFloat (aptNum // 10)
```

In this `case` expression, we see variable binding (`streetNum` and `aptNum`) and the use of “don't care” wildcard (`_`). The variables bound in the pattern to the left of a `->` and only be used in the expression on the right of it.

The use of `case` expressions is pretty important in Elm. For example, we couldn't really use result types without them, as you will see in Section 6.4.3.

One big difference between Elm and many other languages is that *shadowing* is not allowed. Let's look at an example:

```
fun x y =
  let
    x2 = x * x
    sum =
      let
        y2 = y * y
      in
        x2 + y2
    diff =
      (x - y) * (x - y)
  in
    sqrt sum - diff
```

This is a bit convoluted, but just to make a point. The definition `x2` is defined at the second indent level, so we can use it in other definitions at the same level, namely `sum`, `diff`, and `sqrt sum - diff`, but also in the definition of `y2` and the expression `x2 + y2`, because they are “inside” the definition of `sum`.

On the other hand, if we tried to change this to

```
fun x y =
  let
    x2 = x * x
    sum =
      let
        y2 = y * y
      in
        x2 + y2
    diff =
      (x - y) * (x - y)
  in
    sqrt (x2 + y2) - diff
```

we would get an error

```
-- NAMING ERROR ----- ↵
SavvyUser/Test.elm
```

I cannot find a `y2` variable:

```
65| sqrt (x2 + y2) - diff
   ^ ^
```

These names seem close though:

```
x2
y
e
pi
```

which is reasonable, since `y2` is not in scope, but not *that* reasonable, since it exists in the same function, but not in scope. Admittedly, once you understand scope, you are unlikely to make this error, since it is obviously at a higher indentation level.

This brings us to shadowing.

```
fun x y =
  let
    x2 = x * x
    sum =
      let
        x = y * y -- save typing, change y2 -> x
      in
        x2 + x
    diff =
      (x - y) * (x - y)
  in
    sqrt (x2 + y2) - diff
```

The above code does save us from typing two letters, but is awfully confusing and rightly results in an error:

```
-- NAMING ERROR ----- ↵
SavvyUser/Test.elm
```

The name `x` is first defined here:

```
54| fun x y =
  ^
```

But then it is defined AGAIN over here:

```
59| x = y * y -- save typing, change y2 -> x
  ^
```

Think of a more helpful name for one of them and you should be all set!

In a small piece of code like this, you are unlikely not to notice that there are two definitions of `x`—and no good reason to have them! You’ll just have to trust me that in more complex code, there are good reasons you might want to name two things `index` or `shape`, and it takes some effort to come up with different names to explain that. Maybe the first `index` is the one you are looking for, and the second `index` is the one in the data structure you are currently trying to match it to. If you do get a shadowing error, you may feel grumpy about having to think up different names for things which could be understood using the same name. Well, whatever time it takes to invent names when the code is fresh is a tiny fraction of what you would spend if you ever have to untangle the shadowed variables later, when it is not fresh in your mind.

Ok, but what about other modules? Are their definitions at the same level? The answer is...sometimes. It all depends on how you `import` them into the module you are writing. It

is recommended that you use

```
import List
```

or

```
import List exposing (List)
```

to import most modules. By default, it means that if you want to use a definition from the imported module you need to specify the module name, as in

```
List.sort [3,2,1]
```

(to sort a list of numbers). Adding the `exposing` keyword means that you do not need to use the module name as a prefix. In the example above, we expose the type constructor so that we can type

```
type alias ListOfInts = List Int
```

rather than

```
type alias ListOfInts = List.List Int
```

not a big deal, but a nice convention. Using definition names with a module prefix are called *qualified names*. On the other hand, for modules which are core to your work, and whose definitions do not overlap the names of other definitions you use, you can expose all definitions without need for the prefix like this

```
import GraphicSVG exposing(..)
```

Note that exposing all definitions does not prevent you from using the module name as a prefix anyway, and sometimes that is necessary. If, for example, we exposed multiple container modules, we would run into conflicts and need to use qualified names a lot, such as

```
import Array exposing(..)
import Dict exposing(..)
import Set exposing(..)
```

```
type MishMash = MM (Array Int) (Dict Int Int) (Set Int)
```

```
emptyMish = MM Array.empty Dict.empty Set.empty
```

So we strongly recommend always importing the container modules this way

```
import Array exposing (Array)
import Dict exposing (Dict)
import Set exposing (Set)
```

even if you only use one. Yes, you need extra qualifications which would not be necessary, but since we don't have any containers in Elm with names like

`ContainerWhereYouCanLookUpStuffBasedOnKeyThingy`

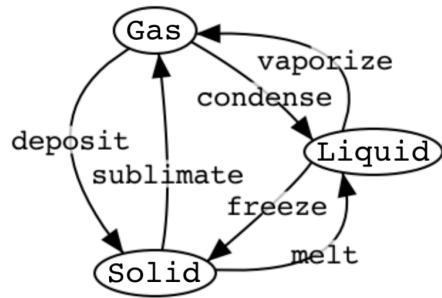
the small number of extra characters improves readability with minimal wear-and-tear on your keyboard.

## 4. To be or not to be in the Basement

Einstein used so-called thought-experiments, in which he imagined an experiment to evaluate the consequences of a theory. His elevator experiment led to the formulation of relativity, the most accurate theory we have of gravity.

We're going to do our own elevator experiment, and from it, develop a theory for change and a tool for modelling change, called a state diagram. With state diagrams, we can model changes to physical systems unlucky enough to be connected to our computers and changes to the graphical user interface through which we interact with our users.

You probably remember learning about states of matter in science class. And if you don't, just shake your head until the generated heat melts those frozen memories. The states of matter look like



where each of the bubbles is a state, and the arrows are called phase transitions.

In general, when we see a diagram like this in computer science, we call it a graph, the bubbles nodes, and the lines edges. If the edges have arrows for direction, it's a directed graph. Graphs are used to represent networks (from train and airplane networks, to the internet, predator-prey networks, and friendship networks); flowcharts showing how to do stuff (for both people and machines); trade flows and data flows; and so much more. One of the most famous algorithms in computer science is Dijkstra's shortest-path algorithm. It operates on a network graph and can be represented by a flowchart graph.

## 4.1 State Diagrams

[to contents](#)

A State Diagram is a directed graph, with nodes we call states, and edges we call transitions. Since keeping track of the state of mechanical devices, like elevators, subway trains, etc., is rather important, Computer Scientists have developed more complicated versions of state diagrams called State Charts, which can include nesting (diagrams within diagrams) and tools to create, visualize, and verify charts. Other tools can even create multiple programs to control the devices and react to outside events, so that the diagrams themselves play the role of the program, and the programmer gets to draw pictures instead of write code. Let's look at the example shown in Figure 4.1. Take some time to click on the "call" buttons outside the elevator and "go to" buttons inside the elevator. The first thing you will notice is that pressed buttons light up. Next, you will notice that the highlighted current state of the car moves through several states each time you click a button. Some states happen so quickly you may not even see them happen, namely the three "Closed" states. We need three "Closed" states because although the doors are closed and we cannot tell where the car is from the outside or the inside of the elevator, the control software does need to know which floor the elevator is on. Since this state diagram assumes the elevator is working and is not broken down, it does not allow for a state where the car is between floors and not moving.

### Exercise 1 (Elevator):

Take 5 minutes to write down all of the things which cannot happen with this elevator, because they are not represented in the state diagram. Now take 5 minutes to randomly pick two states on the ".motion" diagram and figure out which button click(s) take you from one state to the next, and write down the list of states you have to pass through.

A game graph is very similar to a State Diagram, because there is always a place (state) we are in, and for this to change, something has to happen. A single state diagram corresponding to a game would also include any items you are carrying, and in some games whether certain monsters are dead or not. This extra information would significantly complicate the state diagram, to the point where nobody could follow it. This is one case when it is better to use a mixture of a state diagram and parametrized state (like an [Int](#)), and to decompose the state into a product of states. The most flexible option is a state diagram in which states and transitions are labelled by a product of other state diagrams and parametrized states.

The only thing not normally allowed in a State Diagram is a bi-directional edge. If the same transition can go in both directions, two arrows are drawn. While we sometimes do not draw transitions which return to the same state in order to simplify our diagrams, it is unusual in state diagrams that each transition only affects one state, as in the game graph.

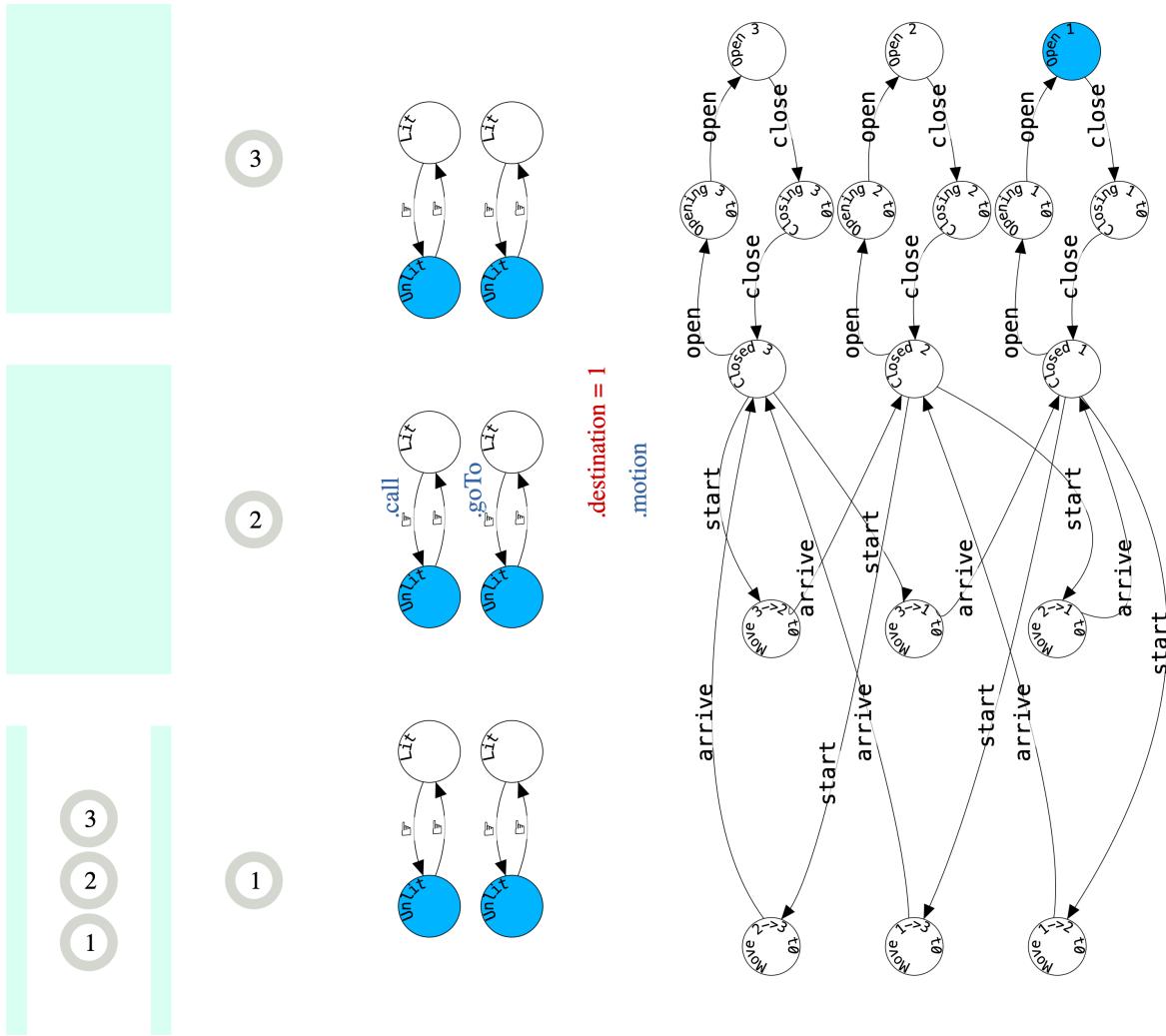


Figure 4.1: An elevator simulation with working buttons from <https://www.cas.mcmaster.ca/~anand/Elevator.html>. On the left, you can see the doors open and close and the elevator car go up and down in response to the buttons. On the right, you can see that each button has a state diagram to record whether it is lit or unlit, and a single diagram captures the state of the door (open, closed, opening, closing) as well as the movement of the elevator car.

State Diagrams are graphical representations of State Machines. Adding a “paper tape” (an infinite row of boxes in which the machine can write or read a symbol—a flexible memory) gives you a Turing Machine. These are the usual ways of defining computation so we can define and prove properties of programs, such as that some programs will go on forever without stopping, that one algorithm will take longer to solve a problem than another, that the solution produced by an algorithm is correct, and that the running algorithm always has certain properties (especially safety properties). So this is a pretty powerful tool, and any theoretical property can always be understood by translating it into a statement about a suitably complicated adventure game—so there is nothing to be afraid of in theoretical computer science, except maybe zombies and trap doors!

State is always present if software does *anything*. When a user understands that state or at least a simplified version thereof, they will say your software “makes sense” or “is logical.” Unless your software is a cryptic logic puzzle, you want your users to say these things, so it is a good habit to sketch out your state as part of your design process.

## 4.2 Implementing in Elm

[to contents](#)

How do we model state and transitions in Elm?

states = data

transitions = functions

Data is every piece of information we have, whether stored in a computer, carved into stone, written on papyrus, or as transmitted from teacher to pupil in the Gāyatrī metre.

In Elm, we can always recognize data types by the `type` keyword, but forming lists and tuples is another way of creating data. Be careful when using tuples that the meaning of the data is clear. Although tuples are really convenient, you probably need a comment explaining what the different components of the tuple represent, because you do not have descriptive names the way you do with constructors.

```
module StateDiagram where
```

### 4.2.1 Example: an Elevator

*State* captures everything we need to know about what is happening *now*, so we know what can happen next, and what actions a user could request.

- doors, open and closed
- be on a floor or in between floors
- doing nothing, or going to a floor
- buttons on floors or in the elevator being lit up

*Transitions* capture both user requests and actions and processes coming to a completion.

- elevator can arrive at a floor
- doors can open
- doors can close
- people can press buttons

- elevator arrives at a place where the button is lit, and the button goes dark
- door closes if no motion is detected (but we don't have this level of detail)

Data recording the state of the elevator has multiple components. Let's start small, with recording the state of one door:

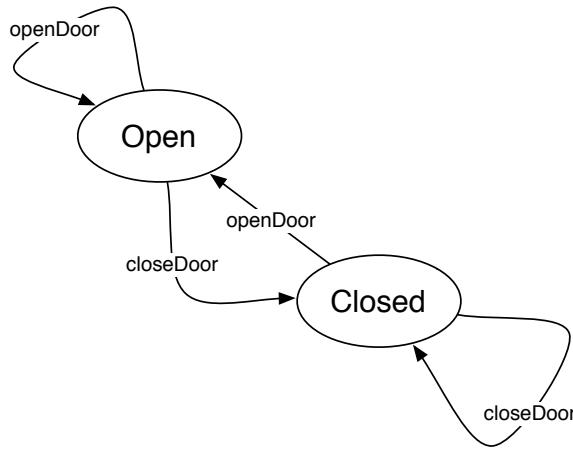
```
type Door
= Open
| Closed
```

Two transitions (each function is represented by arrows labelled by function name).

```
openDoor current =
  case current of
    Open  -> Open  -- self-loop, often omitted in diagram
    Closed -> Open
```

```
closeDoor current =
  case current of
    Open  -> Closed
    Closed -> Closed
```

We can capture these two states and two transitions with a diagram



Note again that when things get complicated, we often omit transition edges which come back to the same state, but in this example, we do draw these self-loops.

Going back to tracking state, we obviously need to know where the elevator is. There are many ways of doing this, and we came up with different solutions, but decided to go with recording the state as being at a floor or going to a floor. A more complicated elevator might record the exact position between floors, and if you are going to support smartphones apps which tell you when your elevator will arrive, you would need to keep track of the speed of the elevator and the number of people getting on and off. If we only needed to record where the elevator is, we could use the following states:

```
type Floor = Basement
| ToBasement
| Ground
| ToGround
| First
| ToFirst
```

Finally, we need a state for one button:

```
data Button = LitUp | Dark
```

The state of the elevator is then a product of these states. If you are not used to the idea of taking a product of sets, a product is what we get when we form pairs out of elements from two different sets. For example, on the line we have an  $x$  coordinate, but on the plane we have  $(x,y)$  coordinates—a pair of real numbers. We can even give our tuple a name:

```
type alias State = (Floor, (Button, Button, Button))
```

It is a `type alias` and not a `type` because we are not defining a constructor for it, but using the built-in tuple constructor. When we use the built-in tuple and record constructors, the Elm compiler can infer types without us giving them names, but that means that the resulting error messages and type signatures do not have meaningful names either. Aliases are there for people.

Buttons being pressed are the simplest user-initiated transitions. Each transition must have an implementing function of type `State -> State`:

```
callToBasement    : State -> State
callToGround     : State -> State
callToFirst      : State -> State
arriveAtGround   : State -> State
arriveAtBasement : State -> State
arriveAtFirst    : State -> State
```

Some state transitions are initiated by the user, others by external events. In this case, the user can call an elevator by pressing a button. Elevators do not have many external events, but there are probably some elevators out there which detect earthquakes and come to a stop. Let's look at one:

```
callToGround state =
  case state of
    (Ground, (bb,gb,fb)) -> (Ground, (bb, Dark, fb))
    (floor,  (bb,gb,fb)) -> (floor,  (bb, LitUp, fb))
```

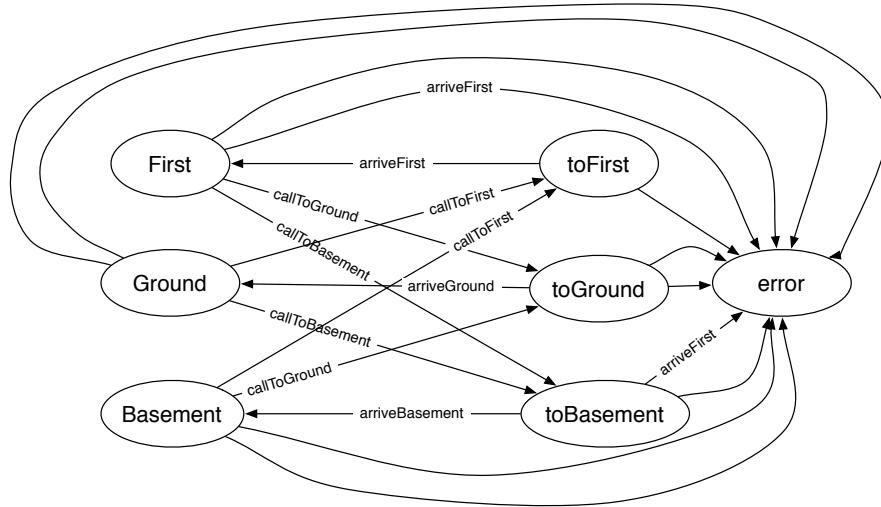
In this case, we will light up the ground-floor buttons when called to the ground floor, except when we are already there! We leave it as a homework problem to write the other functions.

There are also environment-initiated state transitions. When the elevator gets to the ground, then we need to change the `Floor` state and also the buttons for ground. Otherwise, we cannot get this transition, but we have to handle all the cases to avoid an error, so we will return the same state.

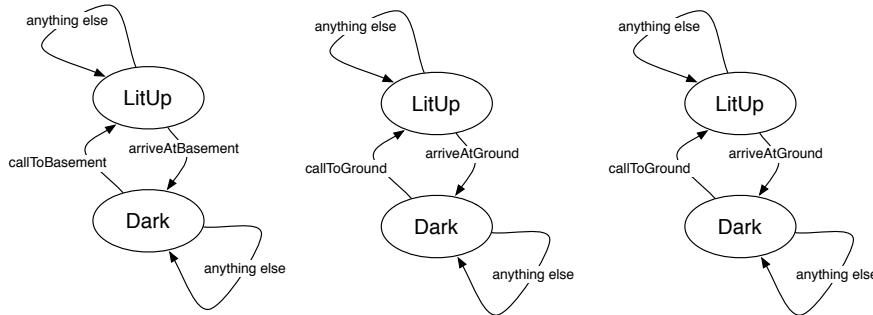
```
arriveAtGround state =
  case state of
    (ToGround, (bb,gb,fb)) -> (Ground, (bb, Dark, fb))
    otherwise -> otherwise
```

Notice that we can use the name of the variable being bound to tell other programmers that this is the `otherwise` case! We leave it as a homework problem to write the `arriveAtBasement` and `arriveAtFirst` functions. In a real system, we would want to check for getting illegal transitions and go into an error state.

Our best hope of getting all of these cases right is to start with a state diagram, or in this case, we can simplify things by drawing separate diagrams for the position of the elevator:



and the lighting of the buttons:



*Hey!* Why is the state of the door not integrated into the other state? People are going to get pretty upset when they find out the door doesn't open! To do this we would need to extend the state, by adding the door state to the tuple, or creating a new set of constructors. This is left as a homework problem.

In the linked example, we couldn't resist adding animations. This does make things more complicated, and we'll share the code now<sup>1</sup>, but we suggest you read on to Section 4.6 where we discuss the use of state for simulation.

<sup>1</sup>elevator: <https://cs1xd3.online>ShowModulePublish?modulePublishId=59a439a5-a229-4945-8916-b85068511e60>

## 4.3 The StateDiagrams Module

[to contents](#)

Let's return to basic science again, and the question that probably bugged you the most: who drew that masterpiece of a diagram? Elm, of course! And for the right price, you can draw your own neat diagrams, all you need is our `StateDiagrams` module which you import<sup>2</sup>,

```
import StateDiagrams exposing(..)
```

then define a type with your favourite states,

```
type States = Solid | Liquid | Gas
```

define the transitions via functions (which in this case we define to leave all states alone, other than the ones they change),

```
melt x = case x of
    Solid -> Liquid
    other -> other
freeze x = case x of
    Liquid -> Solid
    other -> other
sublimate x = case x of
    Solid -> Gas
    other -> other
deposit x = case x of
    Gas -> Solid
    other -> other
condense x = case x of
    Gas -> Liquid
    other -> other
vaporize x = case x of
    Liquid -> Gas
    other -> other
```

and use the `viewStateDiagram` function to draw it<sup>3</sup>:

```
myShapes model =
  [ ProfAnand.StateDiagrams.viewStateDiagram
    stateToString
    statePositions
    transitionPositions
    -- highlight 1 state with Just or none with Nothing
    (Just Solid)
    -- highlight 1 transition with Just or none with Nothing
    (Just (Liquid,"vaporize"))
  |> scale 0.5
]
```

---

<sup>2</sup>StateDiagrams: <https://stabl.rocks>ShowModulePublish?modulePublishId=fef35223-bbcf-4a24-9cb3-849675ff36d5>

<sup>3</sup>highlighted SD: <https://stabl.rocks>ShowModulePublish?modulePublishId=a48130b8-bbf0-4681-801f-69e88b3526d5>

This is the hard part, because in addition to a list of the states you want to be drawn, you need to figure out Cartesian coordinates to pin them to, and in addition to the transitions, you need to decide from which initial states to draw them and where to pin the labels. Note that while the states are data types, and Elm just knows how to show them as strings of characters, transitions are functions which Elm cannot show, so we need to provide a string to display.

## 4.4 SD Draw

[to contents](#)

Since the dawn of time, programmers have had to shuffle state diagrams around in their heads, occasionally etching them on to wide sticks or scratching them onto the beach at low tide. The point is that they had to keep track of the states in their code, and if they were using a language before the invention of enumerations, they had to maintain a consistent numbering of their states, because their compiler only understood primitive types.

But these are modern times, and we have powerful computers, so why not get them to manage the busy work of mapping the state diagram to the code? Actually, we can get them to do a lot more if we adopt Model-Driven Engineering<sup>4</sup> (MDE) and Model-Driven Development (MDD):

Model-Driven Engineering (MDE) is the practice of raising models to first-class artefacts of the software engineering process, using such models to analyse, simulate, and reason about properties of the system under development, and eventually, often auto-generate (a part of) its implementation.

There are different points of view on this, depending on whether you are a user, a developer, a manager, or other stakeholder. Also, not everyone is consistent with how they use these words. In this book, we will use these definitions:

MDD is for developers who don't type that fast, and want tools to take their ideas and automate a lot of the busy work involved in typing out the implementation.

MDE is for managers who want to know when a project will be “done” and have gone bald pulling out their hair, waiting for bugs to be found; they want assurances that if built to spec, the software will work.

One of the things most software developers have to think about is the perspectives of different stakeholders, something you will learn more about in Part III, Design Thinking. But while it is fine to examine other people's suspect motives, it is a bit unusual to look at our own. Take the chance!

Especially when applied to web development, these ideas are sometimes called “low code/no code” development. To some, this means having less code to write, ideally the repetitive bits. For others, it means saving a lot of money by not hiring any programmers.

Oops! I forgot to say what a “model” is in this context. Again, when you scratch the surface, you find a lot of different things called “the model”. This is because of different needs and perspectives. Most models include state diagrams. Some developers need differential equations in their models, because they are monitoring or controlling physical

---

<sup>4</sup>MDE: <https://codebots.com/app-development/what-is-model-driven-engineering>

devices or processes. Some developers need business processes or organizational charts. Many of these models are contained in Universal Modeling Language<sup>5</sup> (UML), but despite the name, they don't cover every useful model. (Although they are worth learning, nevertheless.) For example, UML does not include Petri Nets, which is weird, since Petri Nets<sup>6</sup> can model anything. Certainly, they are very effective at describing concurrency properties, i.e., communicating processes running at the same time (concurrently).

At their most ambitious, models capture the essential properties a system requires, so that it is possible to prove that implementing the system will deliver the required results. Although these requirements may barely scratch the surface of what a user-facing program needs to do—delight the customer—it is great if the program can never crash, or be hacked into sending our bank balance to an account in the British Virgin Islands. In fact, it is possible to prove safety properties of control systems for complex physical devices, and other systems—if the mathematical modeling is good enough.

If it is not possible to *prove* properties of our more complex programs, it may still be possible to simulate them in an abstract way, ideally, in a fraction of the time it would take to implement and test the program. The resulting transcripts can be checked for unwanted behaviour. In the case of expert users, who happen to be experts in something other than computer science, we can even format the transcripts as stories which are easy for them to read.

Maybe that sounds more idealistic to you than ambitious! You plan to get paid for writing programs, and if they don't work, that's the client's fault for not knowing what they want. You just want a tool that writes code for you after a bit of dragging and dropping, so you can get paid, and move on to the next project. Whether you are an idealist or someone who just wants to pay the bills, state diagrams should be part of the model.

Let's look at a really simple concrete example: writing an adventure game. Really simple will mean for us that we only need to track the whereabouts of a single player. No knapsack, no health points. We can model this as a state diagram. Let's do it! You can create your own game map, if you don't like ours. Just open <https://www.cas.mcmaster.ca/~anand/SD.html>. You will see an empty state diagram:

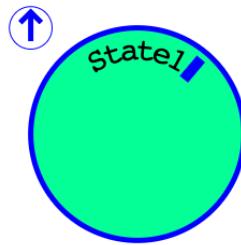
---

<sup>5</sup>uml: <https://en.wikipedia.org/wiki/UnifiedModelingLanguage>

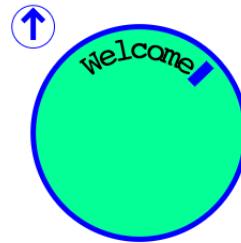
<sup>6</sup>PetriNet: [https://en.wikipedia.org/wiki/Petri\\_net](https://en.wikipedia.org/wiki/Petri_net)



You don't have any states, so click and drag the green **S+**, let go, and where your mouse was you will see



Use the delete key to get rid of **state1** and type whatever you want the starting state of the game to be:

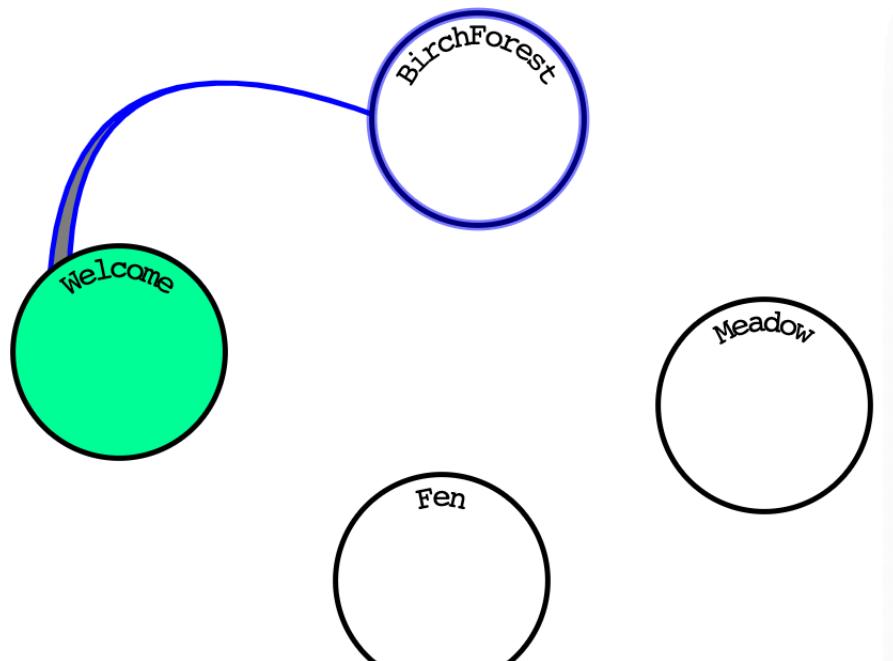


Now press return:



You can continue adding states by dragging the **S+** out, and if you make a mistake and drag too many, you can drag them into the trash. If you change your mind about where to put the states on the map, you can drag them with your mouse. You can also drag the background to pan the whole diagram, or if you have good eyes, you can zoom out with the magnifying glass buttons on the side.

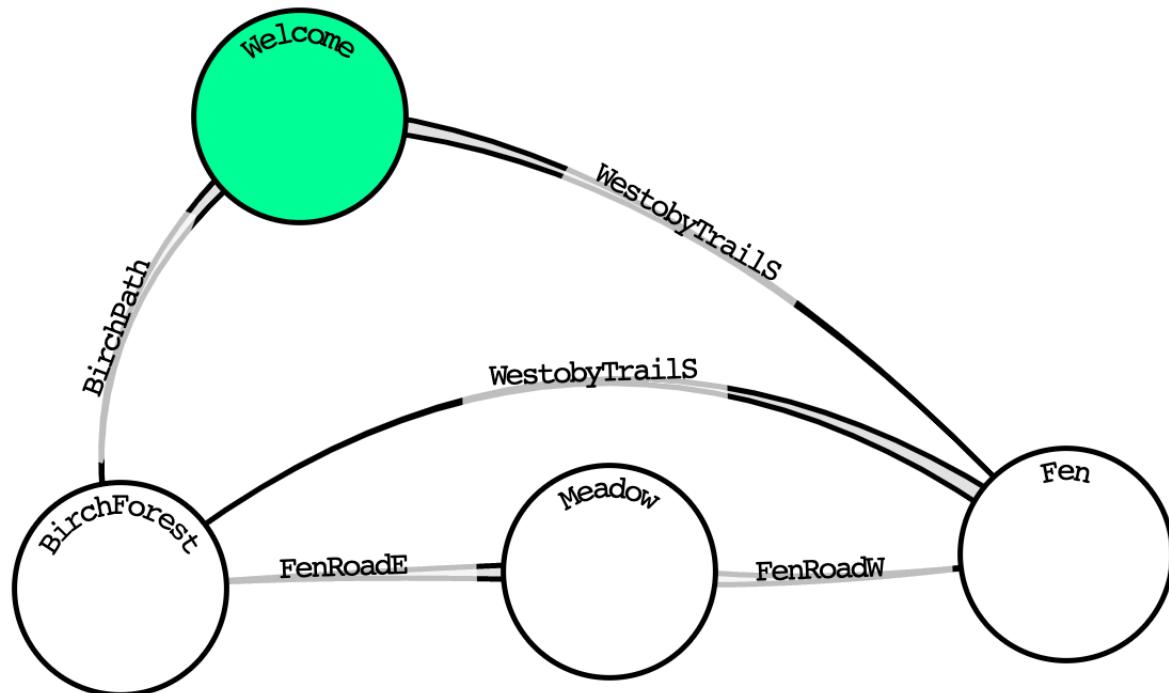
Once you have enough states, you can start connecting them with transitions. There are two ways to do this. You can hold the shift key and drag from one state to another. Alternatively, when a state is selected, you can click on the little arrow-containing circle and drag it to another state:



To edit the transition names, click, delete, and type. When you are finished, press return or enter or click on the background. If you try to finish editing when the name is empty, your screen will go red and tell you names cannot be empty (or include spaces, start with a number or underscore, etc.). Just say “Sorry computer!” and click on the background to get a second chance at creating a valid name. Since transitions are implemented as functions, it will allow multiple transition arrows to have the same name, as long as they start from different states. If your map has a transition name without arrows originating in every state, the transition function will be generated with patterns corresponding to self-loops which return the same state.

Naming transitions is usually harder than naming states. If you are making an adventure game set outside, with paths and roads, you could name them after the path and the direction, such as McMasterWayN or WestobyLaneE.

Now that you've created an example, can we use it to "analyse, simulate, and reason about properties of the system under development?" Let's look at my map first:



Some analysis is probably already finished! Your brain will immediately look at this diagram and notice that all the places are connected. But are they all reachable? For that, you need to look at the direction of the arrows, which start wide and end narrow. You can try to look at this state by state, and you will notice that Meadow only has arrows leading away, so it is not reachable. You can quickly fix this. But it is still possible to have unreachable states even if all states have entering transition arrows. Try to draw one now!

If you are stuck, what about if you added a self-transition to Meadow? It could be called **PickFlowers**. Well, that isn't a fix! In general, your map will break down into self-reachable components, and each of those components needs an inward transition.

This brings us naturally to simulation! You could print out your map and base a board game on it, with players moving around the map after rolling a dice or answering a trivia question. Technically, this is only a state diagram if there is one player, otherwise it is a product of state diagrams, one for each player. This is a physical simulation, and it has the advantage that you can see more than just the current state, you can see how the future could play out. Obviously, this could be automated, and we could even enumerate all possible paths through the state diagram. This is how we can check safety properties for software-controlled devices before we start implementing the code.

Coming back to reachability. Can you invent an efficient algorithm for determining which states are reachable? Hint, try using sets. In case of emergency, read the footnote<sup>7</sup>.

<sup>7</sup>Build the set of reachable states. Start with the start state (coloured green for us, but often drawn with a solid dot) and at each iteration of the algorithm, add states which are not in the reachable state, but are reachable from one of the reachable states. When you cannot add more states, you are finished.

I hope this gives you a feeling for how we can understand our designs better with the right models, and why the state diagram is my favourite model. But we still need to code all of this, right? Wrong!!! Try clicking on the cloud button, near the top-left corner:



You will see code to implement the diagram you've just drawn.

We'll go through the code generated from the above diagram, section by section. All the code produced by SD Draw is there, but we've changed the order to make explaining it easier. As usual, what we see is determined by the function

```
myShapes model =
```

First up is a big `case` expression. At their most basic, `case` expressions are equivalent to a sequence of `if-then-else` expressions, testing the result of the condition against a sequence of values.

```
    case model.state of
```

The `case` expression matches `model.state` to different patterns. You have already used `model.time` to create animations. We updated the time for you, so you could start animating right away. In order to keep track of where we are in the game, however, we needed to add another *field* to the model record. What's a record? It's like a bento box with multiple tasty values, called fields, each of which is labelled by a field name. Field names are easy to confuse with variables, but they are different.

Back to the patterns. For simple state diagrams, each pattern will match one state, and for a game, states are the same as places.

```
Welcome ->
```

The syntax `Welcome -> something` is equivalent to

```
if model.state == Welcome then
    something
else ...
```

In this case, we see that we display a "Welcome" sign:

```
[ text "Welcome"
    |> centered
    |> filled black
```

under which we display buttons for `BranchPath` and `WestobyTrails`.

```
, group
[
```

```

        roundedRect 40 20 5
        |> filled green
    ,
    text "BirchPath"
    |> centered
    |> size 8
    |> filled black
    |> move(0, -3)
]
|> move (-25, -25)

```

We know they are buttons, because we attach a `notifyTap` which means that the Elm system will send us a message if anyone taps (or clicks) on this button shape:

```

|> notifyTap BirchPath
,
group [
    roundedRect 40 20 5
    |> filled green
    ,
    text "WestobyTrails"
    |> centered
    |> size 8
    |> filled black
    |> move(0, -3)
]
|> move (25, -25)
|> notifyTap WestobyTrails
]

```

All of the code for drawing the buttons is here so that you can customize the button, to make it look like a signpost, or even a flashing red button in a state with an emergency eject transition.

Every state gets its own pattern...

```

Fen ->
[ text "Fen"
    |> centered
    |> filled black

```

but they don't all get the same number of buttons. The number of buttons must match the number of transition arrows leaving that state. Otherwise, there would be no way to leave.

```

,
group [
    roundedRect 40 20 5
    |> filled green
    ,
    text "WestobyTrails"
    |> centered
    |> size 8
    |> filled black
    |> move(0, -3)
]

```

```
        ]
            |> move (0, -25)
            |> notifyTap WestobyTrails
        ]
BirchForest ->
    [ text "BirchForest"
        |> centered
        |> filled black

    ]
Meadow ->
    [ text "Meadow"
        |> centered
        |> filled black
    , group
        [
            roundedRect 40 20 5
                |> filled green
            , text "FenRoadE"
                |> centered
                |> size 8
                |> filled black
                |> move(0, -3)
        ]
        |> move (-50, -25)
        |> notifyTap FenRoadE
    , group
        [
            roundedRect 40 20 5
                |> filled green
            , text "FenRoadW"
                |> centered
                |> size 8
                |> filled black
                |> move(0, -3)
        ]
        |> move (0, -25)
        |> notifyTap FenRoadW
    , group
        [
            roundedRect 40 20 5
                |> filled green
            , text "PickFlowers"
                |> centered
                |> size 8
                |> filled black
                |> move(0, -3)
```

```

        ]
      |> move (50, -25)
      |> notifyTap PickFlowers
    ]
  
```

In case you were wondering how the Elm compiler knows that `Welcome` is a state and `BirchPath` is a transition, no, it is not through mind reading! We define types for them. The first `type` in the generated code enumerates all the messages we can receive. By convention, we call it `Msg`, short for “message”, but we could call it anything, as long as we are consistent. The `Msg` type has a message for each of the transition names, but it also has a `Tick` message. This again could be called anything, but we use the name `Tick` to suggest that it happens regularly—like clockwork! This message has always been part of your animations, but was in hidden code. The vertical bar `|` can be pronounced as “or” and separates different possibilities for values of this type.

```

type Msg = Tick Float GetKeyState
| BirchPath
| FenRoadE
| FenRoadW
| GravelRoad
| WestobyTrails
| PickFlowers
  
```

Every Elm program needs a message type, unless it is a picture without interaction or animation. Every program with non-trivial interaction also should have a state type, but it is not technically a requirement. There are other ways to record the state of a program in the `Model`, such as encoding the state as a number, but this is frowned upon now, since it is confusing and makes mistakes too easy. Each line of the `State` type represents a different state of the program. Being a simple adventure game, each state is a different place.

```

type State = Welcome
| Fen
| BirchForest
| Meadow
  
```

So far, we have seen that defining types in this way produces very readable code for states and transitions. These are often called user-defined types, thinking of the programmer as the user of the programming language. We like to call them custom types, because we create them specially for a particular program or application.

Our game would be pretty boring if you always stayed in the same state, since this would mean staring at the same screen forever! For something to happen in the game, we need to be able to change states. All of this is handled by one `update` function which is called every time a message comes in. It takes that message (`msg`) and the current state of the application (`model`), and calculates a new state.

```

update msg model =
  case msg of
    
```

In this `case` expression, we see our first example of a non-simple pattern. The `Tick` message has a variable attached to it (`t`). It also has a piece of data we ignore, indicated by the

\_. We need that \_ there, even if we do not use the data, so that the pattern will match the message.

```
Tick t _ ->
  { model | time = t }
```

Hmm, that last line looks different, and it is. This notation mimics set notation, but it is not set notation. It is record update notation, and not surprisingly, we mostly use it in the update function. We could get by without it. We could have typed

```
{ time=t, state=model.state }
```

instead of `{ model | time=t }`. In the first version, we are defining a record, which is a collection of data organized by name (`time` and `state`). In the second case, we are saying copy all of the data from the record `model`, except for the bit named `time`, which we replace with the value `t`. There is not much of a difference here, but when we build large records, it does save a lot of typing. Even more importantly, we don't have to say exactly what is in our records. We only name the fields that we change. This is *really* convenient when we need to add a new field later on, and the update syntax keeps working, but the name-everything syntax breaks, because now it is missing the new field.

So now you know about records, but why was that code there when there is no `Tick` transition in our state diagram? Well, we automatically add the `Tick` so that `time` stays up-to-date in the model record, just in case you want to create an animation.

Now the rest of the patterns. In each case, after matching the message pattern, such as `BirchPath`, we need to check if that path leads away from the current state.

```
BirchPath ->
  case model.state of
```

Here we know that the user clicked on `BirchPath`. Since `BirchPath` only leads away from `Welcome`, we test for that, and update the state if we find it. Looking back at the state diagram, we see that it leads to `BirchForest`.

```
Welcome ->
  { model | state = BirchForest }
```

But what if we are not in `Welcome`? Well, first of all, this is impossible, the way `myShapes` was generated, but since someone may add buttons to the code for `myShapes`, we need to handle all cases. Since there are no other `BirchPaths` on the diagram, all other cases do not change the state, so we return the model as it currently is. To do this, we use a variable in the pattern `otherwise` which matches anything. We didn't need to call it `otherwise`, but this helps make the code more readable.

```
otherwise ->
  model
FenRoadE ->
  case model.state of
    Meadow ->
      { model | state = BirchForest }
```

```

        otherwise ->
            model
FenRoadW ->
    case model.state of
        Meadow ->
            { model | state = Fen }

        otherwise ->
            model
GravelRoad ->
    case model.state of
        otherwise ->
            model

```

Most of the cases in this case expression handle messages for which there is only one button. The `GravelRoad` and `WestobyTrails` messages are the exceptions. Since the `GravelRoad` transition was deleted during editing of the diagram, it doesn't have any buttons to generate it, so we only have an `otherwise` pattern. We should really eliminate this message from the type, and this pattern from the `case` expression. On the other hand, `WestobyTrails` appears twice in the diagram, so the inner case expression below matches for both `Welcome` and `Fen`. If you trace this path on the diagram, you see it goes right through the `Fen` with two clicks of the buttons labelled `WestobyTrails`. Can you follow how this works in your code?

```

WestobyTrails ->
    case model.state of
        Welcome ->
            { model | state = Fen }

        Fen ->
            { model | state = BirchForest }

        otherwise ->
            model
PickFlowers ->
    case model.state of
        Meadow ->
            { model | state = Meadow }

        otherwise ->
            model

```

At the bottom of the generated code, we define `Model`, the type for the `model` variable:

```

type alias Model =
    { time : Float
    , state : State
    }

```

It is a record with two fields. If you never make mistakes, you don't need it in your code, because Elm can figure it out from how you use it. This is called type inference. As an experiment, you could comment it out before compiling, and you will find an error on the line below, which we need to comment out as well, and then everything will compile. But if you make an error involving the `Model` type, the compiler will not be able to refer to it by name.

Note that `Model` is not a `type`, but rather a `type alias`. This is because the notation for a record defines the type, unlike the `Msg` and `State` types above, so all we are doing is giving the type a nickname—an alias!

```
init : Model
init = { time = 0
        , state = Welcome
      }
```

## 4.5 Model-View-Update with TEA

[to contents](#)

We now have the basic ingredients for understanding *The Elm Architecture*, which is the preferred way of building Elm applications. There is no Interpol department assigned to enforce this, but if you are going to read other people's code, or want to share your own apps and components, it is good to stick to conventions. In fact, the Elm Architecture is really only making explicit the only sensible way to write user interfaces in Elm.

Why now? Because we needed the idea of state. The starting point for designing a UI is to figure out what state the app must "remember" and to which actions the app must respond. For really simple apps, the responses are uniform. Imagine, the Beep App has one action, click its single button, and it responds by beeping. But most apps respond differently to external actions depending on what state they are in. For example, asking to close a document when there are unsaved changes will—I hope—prompt for saving before losing changes. Making separate tables of state and user actions is only part of the story, better to use a state diagram to capture all allowed sequences of user interaction. For even moderately interesting applications, this is not enough. Any app getting images from the internet, posting high scores to a leader board, or including animations will also need to respond to network or internally generated events for things like: completion of a download, passage of time, or loss of network connectivity.

Before we explain the Elm Architecture, why does software have *architecture*? What is the relationship to design? Both words are borrowed from art and building, with design originating in the marking of the outlines of, e.g., a building, and architecture coming from the Greek word for builder. In software, architecture means either a framework of rules to be used in designing software, or it means the highest-level part of the design. For a beginner, the existence of two words for this process should be evidence that software design is not easy, but is important<sup>8</sup>.

<sup>8</sup>More information, and a wealth of references is available at [https://en.wikipedia.org/wiki/Software\\_architecture](https://en.wikipedia.org/wiki/Software_architecture).

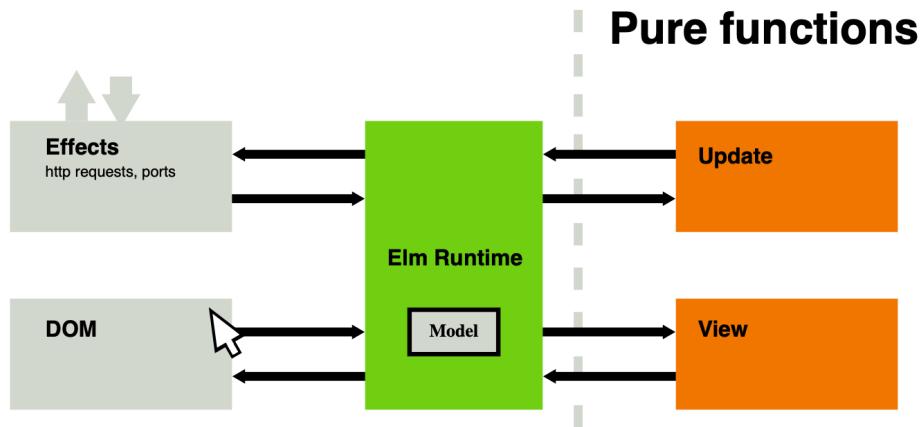
According to Wikipedia, architectures are high-level designs for large systems. The Elm Architecture or model-view-update is really an architectural pattern. It is related to an older pattern, model-view-controller<sup>9</sup> which was created to organize user interaction in Smalltalk-79<sup>10</sup>. This was way before people started using the word software architecture, back when object-oriented programming could legitimately be called modern.

But we don't need Wikipedia to tell us that "architecture" is meant to sound important, implying that we are solving some big problems when we talk about it. What is the big problem that model-view-update solves? In addition to the already formidable task of organizing UI code so it can be understood by mere mortals, it also solves a puzzle: in a pure language (without side effects, in which the same input always gives the same output) how can an app do different things? Now you either see this as a fundamental problem, or a non-issue. It all depends on what you think are the functions and what are the inputs. If you think the app should do the same thing only if the user types the same input, then no problem here, move along.

If you have used a UI library before, however, even as simple as

```
scanf("Type stuff!",&string_var);
```

in C, you see the same input ("Type stuff!") producing different outputs. So it is a *big* problem to do this in a pure language. This is where Elm takes a page out of the mathematicians' book! Since we cannot write interactive programs within the confines of a pure language, we create a bigger universe inside of which the pure language lives. That bigger universe is the Elm run-time system:



The functions which you have been writing, called by `myShapes`, are all part of the View box. The `update` function generated by SD Draw is the Update box. Those can both be pure functions, because they are part of a bigger universe, with the green Elm Runtime box mediating<sup>11</sup> between the outside world (including user input, network access, etc.) and

<sup>9</sup>mvc: <https://en.wikipedia.org/wiki/Model-view-controller>

<sup>10</sup>See <https://en.wikipedia.org/wiki/Smalltalk> for a brief history of this influential language, or even better, download <https://pharo.org> and test out a modern implementation

<sup>11</sup>See an animation as part of the slides: <http://www.cas.mcmaster.ca/~anand/TFPIE2019Slides.html>

your Elm program. The Elm Runtime cannot be pure, but that's ok, because we don't have to write it or debug it. Evan Czaplicki<sup>12</sup> handles that for us!

A couple of things to note about this architecture. First, it is not the only way to accommodate interaction with a pure functional language. Clean uses uniqueness types<sup>13</sup>, which tells the compiler to enforce that values can only be used by one thread of execution. Haskell borrows a concept from logic called monads<sup>14</sup>.

Second, the apparent inefficiency of this model can be overcome with cleverness. Calling the `view` function every time something changes in the runtime's `model`, and recalculating the display even though everything visible to the user could be the same seems like a lot of wasted effort. For a 1980s game programmer—programming in machine code and counting how many times a pixel gets overwritten—it is! But very few developers today would be capable of writing that code. The reality is that everything about web programming is pretty inefficient, which is accepted because computers are so much faster—until they aren't fast enough! If you are writing complex code, or you just hate inefficiency, the Elm rendering pipeline is actually able to achieve best-in-class efficiency<sup>15</sup> even without the programmer doing optimization on the source code. Part of this is a better implementation of something called a virtual dom, which means keeping track of what changed when you recalculated, and only redrawing those parts. In theory, any language could do these things, but when all functions are pure, we know that once we've seen these inputs before, we already know what the outputs will be. If there are side effects, you cannot know this without recalculating.

## 4.6 Simulation

[to contents](#)

We have now learned about state, its representation as state diagrams, and how interaction in Elm is organized using the model-view-update pattern. This is also the principle behind how we simulate physical states with a computer, whether to predict the weather, or create fun game physics. We can even simulate phenomena which we cannot explain in terms of basic physics, but whose observed behaviour follows mathematical models, like the traffic on a highway, or ghosting on a social network.



Let's start with something really simple—a rolling ball<sup>16</sup>:

```
myShapes model =  
  [  
    circle 5 |> filled red
```

<sup>12</sup>evancz: <https://github.com/evancz>

<sup>13</sup>uniqueness: [https://en.wikipedia.org/wiki/Uniqueness\\_type](https://en.wikipedia.org/wiki/Uniqueness_type)

<sup>14</sup>monads: <http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html>

<sup>15</sup>blazing: <https://elm-lang.org/news/blazing-fast-html-round-two>

<sup>16</sup>rolling: <https://cs1xd3.online>ShowModulePublish?modulePublishId=ab38d344-5fae-466f-98fe-db08314d0818>

```
|> move (model.x, model.y)
]
```

This is our `view` function. It takes the `model` as an input, and we see that we have added two state variables `x` and `y`, and use them as coordinates for our ball. If we define a type of our model, we must add them to it:

```
type alias Model = { time : Float
                    , x : Float
                    , y : Float
                    }
```

We didn't need to define a state type, because this system we are simulating doesn't have discrete states, it has continuous states—the `x` and `y` positions of the ball on our imaginary table—and `FLOATS` were meant for modelling them.

Before we can see how this state updates, we need to define our transitions. But there is only one, the `Tick` message we will get every time the browser is ready to redraw the screen—hopefully 60 frames per second (fps) to create a smooth animation.

```
type Msg = Tick Float GetKeyState
```

In a real-time simulation, we need to know how much time has elapsed, so that we can apply rules for state changes. In the case of a ball rolling on a table, we can assume the table is level and flat, so no applied forces will be acting on the ball. Therefore, the velocity will remain constant, and the position will change proportional to the elapsed time.

$$\Delta_x = v_x \Delta t \quad (4.1)$$

$$\Delta_y = v_y \Delta t \quad (4.2)$$

The change in `x` (respectively `y`) equals the change in time multiplied by the component of the velocity along the `x` axis (respectively `y` axis). We do this calculation, and apply the result every time we get a `Tick`:

```
update msg model =
  case msg of
    Tick t _ ->
      let
        deltaT = t - model.time
      in
        { model | time = t
                  , x = model.x + vx * deltaT
                  , y = model.y + vy * deltaT
                  }
```

That is the heart of the simulation, but for this first example, let's go over the whole module. We need to set the initial state, which includes the elapsed time, and the `x` and `y` positions:

```
init = { time = 0
        , x = 0
        , y = 0
        }
```

We also used variables for the velocity components, so we need to define them:

```
vx = 10
vy = 0
```

Go ahead and change them or the initial state, and see how the simulation changes!

We also define the main function, using `gameApp` in which you can change the title of your app, to “Really Exciting Simulation!” or another title with an appropriate level of enthusiasm.

```
main = gameApp Tick { model = init
                      , view = view
                      , update = update
                      , title = "Game Slot"
                    }
```

And we see that the `view` is not just `myShapes`, but also includes the size of the `collage` on which we will draw:

```
view model = collage 192 128 (myShapes model)
```

At this point, you can confidently predict that if you roll a ball on a table, it will roll off the end. What else can we do with a couple billion transistors? Let’s add forces acting on the ball.



Force causes acceleration, which is the change in velocity:

$$\begin{aligned}\Delta_x &= v_x \Delta t \\ \Delta_y &= v_y \Delta t \\ \Delta_{v_x} &= a_x \Delta t \\ \Delta_{v_y} &= a_y \Delta t\end{aligned}\tag{4.3}$$

Since now the velocity can change, we have to take the definitions for velocity away, and add velocity variables to our model.

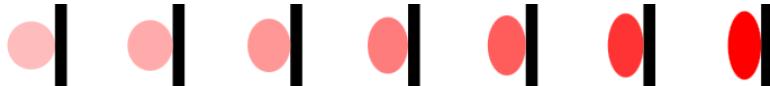
```
type alias Model = { time : Float
                     , x : Float
                     , y : Float
                     , vx : Float
                     , vy : Float
                   }
```

and add velocity updates to our `update` function

```
{ model | time = t
          , x = model.x + model.vx * deltaT
          , y = model.y + model.vy * deltaT
          , vx = model.vx + ax * deltaT
          , vy = model.vy + ay * deltaT
        }
```

Does it look<sup>17</sup> right? Now the problem isn't physics so much as plot. Once the ball exits stage right, there is no action—other than your audience rising up and demanding their money back. How about we add a wall for the ball to bounce into?

What is the physics of that? You probably never learned this, because it is actually pretty complicated. When the ball hits the wall, both the ball and the wall change their shape. The energy of motion is converted into stored energy in the bonds between the atoms within the ball and within the wall.



We call the collision perfectly elastic if all of the energy is converted in this way, and then it converts right back again. To make our lives easier, we can assume that the ball is rubber and the wall is steel, so the wall isn't going to do much deforming. The deformation of the ball is now another state variable we have to keep track of. Since our ball is moving along the  $x$  axis, we can start out modelling only the  $x$  squishiness. Fortunately, rubber balls, like many materials, exert a force proportional to the amount of squish. Meaning that the acceleration experienced because the ball pushes into the wall and the wall pushes back on the ball, will follow<sup>18</sup>

$$am_{\text{ball}} = F_{\text{ball}|\text{wall}} = k_{\text{springiness}} d_{\text{squish}} \quad (4.4)$$

a law which says that the acceleration,  $a$ , is equal to the product of a constant and the difference,  $d$ , between the normal shape of the ball and the squished shape. How you deal with the constant depends on your programming purpose. If you want to find out how buildings wobble (and hopefully not fall down) in an earthquake or strong wind, then you need to buy a bunch of scientific equipment and do experiments to figure out realistic values for the constants for the materials in your buildings. If you want to make a game, you try different constants and ask people which versions look “realistic,” or just fun. Let's see how translating this squishiness into code will work. Since we are in a 2D universe, it is easy to maintain the same ball area, as we squash along one axis, we squash along the other:

```
, circle 5 |> filled red
|> scaleX model.squishX
|> scaleY (1/model.squishX)
```

which we can accomplish by adding the current squish to the model. We can check the math on area conservation. Since area is height times width, then

$$A_{\text{original}} \times \sigma_x \times (1/\sigma_x) = A_{\text{original}} \quad (4.5)$$

where we use Greek “s” ( $\sigma$ ) for squish/scaling.

<sup>17</sup>acceleration: <https://cs1xd3.online>ShowModulePublish?modulePublishId=8f5dc790-2c9f-4850-9cf7-65dcd40ebbec>

<sup>18</sup>Hooke's law: [https://en.wikipedia.org/wiki/Hooke%27s\\_law](https://en.wikipedia.org/wiki/Hooke%27s_law)

Next, we'll put the ball in a box:

```
rect 190 126 |> outlined (solid 2) black
```

which has its right boundary at  $94 = (190 - 2)/2$ , since the boundary is 2 units wide, half inside the rectangle, but on both sides, so we subtract 2, then divide by 2 to find the distance from (0,0) to the boundary. We update the `update` function to add in this bounce when the ball is within its radius of the boundary:

```
if model.x + rBall > wallX then
```

... we calculate the amount by which the ball has deformed, and use the constant `kSquish` to convert that deformation into acceleration

```
let
  squish = wallX - rBall - model.x
  totalAx = ax + kSquish * squish
in
{ model | ... }
```

and we use the adjusted acceleration to adjust the velocity, as well as storing the value we need for `scaleX` and `scaleY` in the model, so that all of the squish calculations are in the same place.

```
, vx = model.vx + totalAx * deltaT
, ...
, squishX = 0.2*(5+squish)
}
else
{ model | ... }
```

If I were better at timing screenshots, I'd post one here, but in this case, it is better for you to try it yourself<sup>19</sup>. Now, if you run this demo for a while, you will notice that the bounce does not just roll the ball back to where it started—it gives it a bit of extra kick each time! Why does this happen?

Did we not faithfully follow Newton's laws? Well, not quite. Newton's laws work at instants in time. Our time steps may be small enough to fool our eyes, but they are not instantaneous. There are a few ways we could make them better:

1. We could make shorter steps and get closer to the instantaneous ideal.
2. We could follow in the footsteps of Madhava<sup>20</sup> and the Kerala School of mathematics, and apply calculus to create efficient and accurate polynomial series approximations.
3. We could follow in the footsteps of Lagrange<sup>21</sup> and refocus on energy, and ensure that our solutions have important properties like conservation of energy. And if we keep walking in this direction, we will arrive at Noether's Theorem<sup>22</sup>, which says that for every conservation law, there is a corresponding symmetry, and vice versa—in this case, energy.

<sup>19</sup>bounce: <https://cs1xd3.online>ShowModulePublish?modulePublishId=ce050cc3-8903-4076-aada-beee0d82dc8b>

<sup>20</sup>Madhava: [https://en.wikipedia.org/wiki/Madhava\\_of\\_Sangamagrama](https://en.wikipedia.org/wiki/Madhava_of_Sangamagrama)

<sup>21</sup>Lagrange: [https://en.wikipedia.org/wiki/Lagrangian\\_mechanics](https://en.wikipedia.org/wiki/Lagrangian_mechanics)

<sup>22</sup>Noether's Theorem: [https://en.wikipedia.org/wiki/Noether%27s\\_theorem](https://en.wikipedia.org/wiki/Noether%27s_theorem)

4. We could “lose” some of the energy by adding “friction” to reduce the velocity:

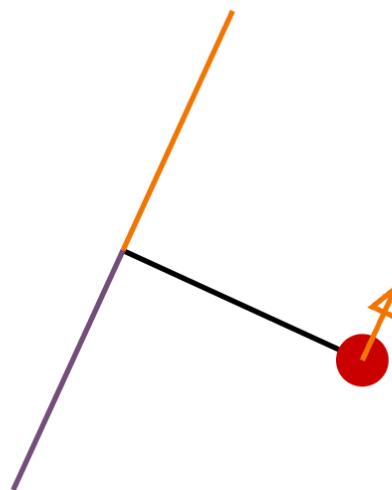
```
, vx = 0.95 * (model.vx + totalAx * ←
                 deltaT)
, vy = 0.95 * (model.vy + ay * deltaT)
```

We'll go with friction, which we put in quotes, because this is friction for the purpose of looking “realistic”, not being a realistic simulation.

Although we have a fix, a simple example illustrates where the problem comes from, and how it is inherent to the kind of stepwise approximations we use in simple game physics. Think about attaching a ball to a string this time and twirling it around your head. Your YouTube influencer friend's drone will capture the ball going in a circle, and you notice on the video later that (1) your makeup guy should have used a comb, (2) that the string is always tracing the radius of the circle, and (3) that the ball is always moving at right angles to the radius.

Knowing this helps you draw a diagram<sup>23</sup> for the video: You remember that sine and cosine are defined to be the coordinates on a circle at a particular angle:

```
let
  x = 50 * cos model.time
  y = 50 * sin model.time
in
  [ circle 5 |> filled red
    |> move (x, y)
  , ...
```



Having expressed the position as  $(x, y)$ , it is easy to draw the string using `openPolygon`, and it is useful to remember that  $(-y, x)$  is the point rotated by  $90^\circ$  counterclockwise ( $(y, -x)$  would be clockwise). Once you know this, you can draw line figures rotated to match the

---

<sup>23</sup>BallOnString: <https://cs1xd3.online>ShowModulePublish?modulePublishId=9bfff09eb-91d0-41c4-ad12-925bcbee3db9>

angle of  $(x, y)$ . Just take the coordinates of a point in the unrotated polygon  $[..., (u_i, v_i), ...]$ , and convert to  $[..., u_i \times (x, y) + v_i \times (-y, x), ...]$ . If you follow the layout in the code example, you will find it hard to make mistakes.

Ok, having absorbed that pro-tip, do you see the example where applying Newton's laws using steps really breaks down? Instead of calculating the position based on sine and cosine, use the perpendicular rule for the direction of motion, and add in steps:

```
Tick t _ -> { model | time = t
                  , x = model.x + stepSize * (-model.y)
                  , y = model.y + stepSize * model.x
                }
```

Since the direction of the step is always pointing outside the circle, you know the circle along which the ball moves will keep getting bigger. If you try it yourself<sup>24</sup>, you won't immediately see a problem, but you know what is going to happen, and pretty soon the ball starts floating off the screen. If you are less patient, increase the `stepSize` to see how the problem gets worse with increasing step size.

## 4.7 Real-Time Interactive Games

[to contents](#)

Now that we can simulate motion, we can put the user back in the picture. You have already seen how you can use buttons to modify state, and if you create some nifty flame animations, you could add states for rockets firing, and replace your red ball with a rocket ship. But if, instead, you are serious about recreating Asteroids<sup>25</sup>, you know you need keyboard controls. So we built that into `gameApp`, and it has been there<sup>26</sup> under your nose all this time, embedded in the `Tick` message

```
Tick t (_ , accelR , accell )
```

inside the piece we have always ignored. Both `accelR` and `accell` are coordinate vectors encoding the direction indicated by the arrow keys (`accelR`) and the AWSD keys (`accell`), all set up for a two-player game. The vectors even work on diagonals, so any of the values

$$\begin{array}{ccc} (-1, 1) & (0, 1) & (1, 1) \\ (-1, 0) & (0, 0) & (1, 0) \\ (-1, -1) & (0, -1) & (1, -1) \end{array}$$

are possible.

<sup>24</sup>Ball simulation: <https://cs1xd3.online>ShowModulePublish?modulePublishId=cf66af3f-e973-480a-baca-be2a5fa1abd0>

<sup>25</sup>which in 1979 was too computationally demanding to be drawn with bitmapped graphics, so they used vector graphics in which they traced out the shapes by moving the electron gun at the heart of every Cathode Ray Tube display [https://en.wikipedia.org/wiki/Asteroids\\_\(video\\_game\)](https://en.wikipedia.org/wiki/Asteroids_(video_game))

<sup>26</sup>two ships: <https://cs1xd3.online>ShowModulePublish?modulePublishId=735eb418-d43b-4015-8dca-56c746928e03>



I know you are thinking “Finally, I can program a real game!” But before you begin, it is worth pointing out that we had to organize things a bit better in order to accommodate two ships in our code. Number One, we start using pairs (`Float,Float`) to represent Cartesian coordinates and displacement vectors. This roughly cuts in half the number of fields in our `Model`:

```
type alias Model =
    { time : Float
    , posL : (Float,Float)
    , posR : (Float,Float)
    , velL : (Float,Float)
    , velR : (Float,Float)
    , winner : String
    }
```

and other variables, and keeps data together which belongs together. This is an important software design principle: “Don’t give bugs places to hide! Reduce lines of code and definitions by grouping data together.”

But this would be only half the job, and in some ways counterproductive, if we didn’t also make functions to act on these clumps of data together, which we have done with

```
timeStep (changeX,changeY) (x,y) = ( x + s * changeX
                                         , y + s * changeY)
```

and

```
isInCircleWithRCentre r (x,y) (u,v) = (x-u)^2 + (y-v)^2 < r^2
```

This really improves the readability of the `update` function:

```
...
, posL = timeStep model.velL model.posL
, posR = timeStep model.velR model.posR
, velL = timeStep accelL model.velL |> friction
, velR = timeStep accelR model.velR |> friction
...
```

Ok, now you can go make a game! And if you prefer a ball game, start with the two-paddles example<sup>27</sup>.



## 4.8 Clickable Ruler

[to contents](#)

We have seen how `notifyTap` can be used to send messages whenever something is tapped (or clicked) in your interface. But what if your stickers app needs to know where you tapped, so you know where to put the stickers? Well, there are a few other `notify*` functions. Use the `</>` button, or link<sup>28</sup> to open the GraphicSVG documentation.

 MacCASOutreach / `graphicsvg` / 7.2.0



**GraphicSVG: Beautiful vector graphics in Elm**

Create beautiful graphics and animations in your browser. Build your graphics and animations by composing and transforming basic shapes. Create beautiful full-page Elm apps with the power of the Elm Architecture, or embed our graphics into your existing `elm/html` projects.

[README](#)  
[About](#)  
[Source](#)

**Modules**

`notifi`

**GraphicSVG**

- `notifyEnter`
- `notifyEnterAt`
- `notifyLeave`
- `notifyLeaveAt`
- `notifyMouseDown`

Notice the search box on the right-hand side? If you type in “notify” there, it will show you 16 functions. The `*Tap*` functions are safe for both touch and mouse devices. The one we want now is

```
notifyTapAt : (( Float, Float ) -> userMsg) -> Shape userMsg -> ↴
    Shape userMsg
```

Just like `notifyTap`, this is a `Shape` transformer, because its last input is `Shape userMsg` and its output is `Shape userMsg`, so we can apply it to a shape—in this case the ruler:

<sup>27</sup>TwoPaddles: <https://cs1xd3.online>ShowModulePublish?modulePublishId=1b4ccaa2-6641-4404-8105-2d5fbc785ab3>

<sup>28</sup>graphicSVG: <https://package.elm-lang.org/packages/MacCASoutreach/graphicsvg/latest/>



just like we did before

```
myShapes model = [ text (String.fromFloat model.size
                           |> String.left 5)
                     |> filled black
                     , ruler
                           |> notifyTapAt ClickRuler
                     ]
```

But unlike `notifyTap`, its first input is a function (`Float, Float`) $\rightarrow$  `userMsg`. Remember, we know it is a function because it is in ()s. If `ClickRuler` were a simple constructor like we used for previous messages, we would get an error, but it is not!

```
type Msg = Tick Float GetKeyState
          | ClickRuler (Float,Float)
```

It is a constructor with data—(`Float,Float`)—attached. Any constructor with data attached can also be used as a function, which takes that data as input, and returns a value of the constructed type, which is `Msg` in this case.

It is true that using types in this way can create some confusing error messages, but they pale in comparison to the weird run-time errors you get in untyped languages when you make similar errors.

The final piece of the puzzle for capturing click/tap position is, well, capturing the position, which happens in the `update` function.

```
ClickRuler (x,y) -> { model | size = x /180 * 31 + 15.5}
```

In this case, we do a little computation, based on knowing where the mouse is within the `Collage`.

If you were always jealous of kids with name-brand rulers, you can take our code<sup>29</sup> and create your own.

## 4.9 Mouse Over

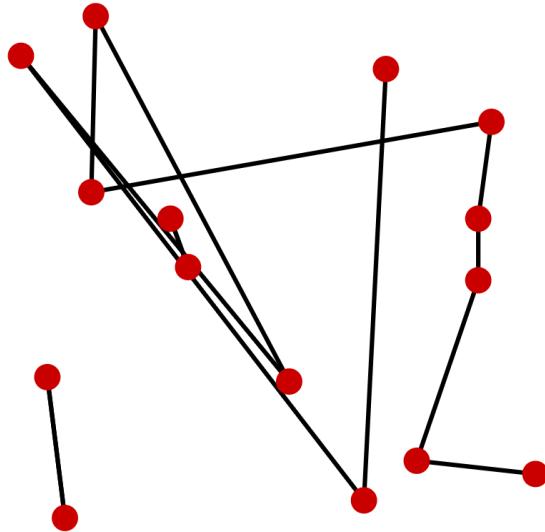
[to contents](#)

For information-rich apps, mousing over to reveal additional information or change the context can be very helpful. It is also useful for simple tooltips. But be aware that touch users will not be able to access this info.

The most straightforward is offering a tooltip, but you can imagine mousing over a path on a map to give you information on travel time and energy use, or mousing over a component in an electrical diagram to highlight all of the directly connected components, or even using a heatmap to indicate distance to the current component by reducing colour saturation, or mousing over a waterway to highlight upstream and downstream sections of its watershed in different colours.

<sup>29</sup>ruler: <https://cs1xd3.online>ShowModulePublish?modulePublishId=498e9c1e-1e3a-4161-b14c-45a91c36a142>

Here is a simple example<sup>30</sup> of a graph, with nodes and edges:



For this one we needed two new messages:

```
| Enter (Float,Float)
| Leave (Float,Float)
```

which also have coordinates attached to them, but this time we will not let the run-time system attach the coordinates based on the mouse position. We will attach them ourselves, when we attach the notification to the `Shape`:

```
|> notifyEnter (Enter pos)
|> notifyLeave (Leave pos)
```

To avoid having to type out the code for each node, we are going to steal a function `List.map` from Section 6.4. For now, just focus on the fact that it applies the anonymous function

```
List.map ( \pos -> circle 3 |> filled red |> move pos
          |> notifyEnter (Enter pos)
          |> notifyLeave (Leave pos)
        )
nodes
```

to every node in `nodes`. When the message is sent by the run-time system, we will be able to know which node the mouse is entering or leaving by its `pos`ition which we attached in the `myShapes` function. We then store that info in the `Model`

```
Enter pos -> { model | highlight = Just pos }
Leave pos -> { model | highlight = Nothing }
```

as a `Maybe` value, another thing we have to steal, this time from Section 6.4.3.

Unfortunately, this is not 100% reliable. Can you look back at the code and guess why not?

---

<sup>30</sup>hover graph: <https://cs1xd3.online>ShowModulePublish?modulePublishId=ed6324dd-20af-487c-bd4a-f19812b95aff>

It may surprise you that the `Enter` and `Leave` messages can arrive in an unexpected order. Well, you probably think we can only get one at a time, because we cannot be entering and leaving at the same time. But you cannot count on that, especially if you have a very sensitive mouse that picks up tiny vibrations as you drag the mouse over the table. As a rule, DO NOT ATTACH MULTIPLE MESSAGES to a single `Shape` if the order of those multiple messages matters. But, but, but, we need both messages, you may be thinking, but do you need them both at the same time? Of course, you only need the message which would change the current state, so we can pattern match on that state and only attach the message we need:

```
\pos -> circle 3 |> filled red |> move pos |>
  ( case model.highlight of
    Nothing -> notifyEnter (Enter pos)
    Just _ -> notifyLeave (Leave pos)
  )
```

Notice that the `case` expression evaluates to a function? Pretty nifty<sup>31</sup>!

## 4.10 Slider

[to contents](#)

Let's explore a few more mouse messages we can use to create a slider.



To have a slider, we need a value in the `Model`, in this case `model.value`, which we can use to draw the rectangle:

```
rect model.value 10 |> filled (rgb 0 0 255)
```

but remember that `rects` and other shapes draw themselves centered, which would make for a weird slider, so we have to left-align it:

```
|> move (-50 + 0.5 * model.value, 0)
```

Since the user can click on the filled as well as empty part of the slider, we cannot attach our messages to the filling. Best we attach our messages to an invisible `rect` floating above the filling—`ghost` makes it invisible:

```
, rect 100 10 |> ghost
|> addOutline (solid 0.5) red
-- clicking on the slider starts the adjustment
|> notifyMouseDownAt StartSlider
```

The first notify ends in `*At`, so we know we will get a coordinate, so `StartSlider` will need (`Float, Float`) as attached data.

```
-- moving over the slider sends a message, but the update ←
only accepts it if we are editing
|> notifyMouseMoveAt Slider
```

<sup>31</sup>fixed: <https://cs1xd3.online>ShowModulePublish?modulePublishId=67e9ce85-3e5c-4cc0-9d06-61a13b26155f>

We now violate the \*\*\*RULE\*\*\* we just made about not attaching multiple messages. In this case, it is ok if we get these messages out of order. It would cause at most a tiny change in the filled percentage.

```
-- both lifting the mouse, and mousing outside the slider ←
-- stops the adjustment
|> notifyMouseUp EndSlider
|> notifyLeave EndSlider
```

But now we do need to think about it. If we drag across the boundary of the slider, or unclick the mouse, then dragging stops. Moving back will not change that.

We could have attached the last three messages only when dragging, but it won't make a difference in this case.

```
update msg model = case msg of
    Tick t _ ->
        { model | time = t }
    Slider (x,_) ->
```

What we *do* do in this case is validate the `Slider` message, by checking the `model.dragging` state. (Note that being a `Bool` we don't need to write `model.dragging==True`, because the `(==True` doesn't do anything.) We then do a second validation on the position, to make sure our value is always a valid percentage, and clamp values to 0 or 100 if outside the valid range.

```
if model.dragging then
    { model | value = if x < -50
        then 0
        else if x > 50
            then 100
            else x + 50
    }
else
    model
```

The `StartSlider` message also needs to reset the `value`, with the same validation. (Exercise: improve this code by replacing the duplicate validation with a validation function.) Did you notice that since we only use the `x` position, we don't need to give the `y` position a variable name in the pattern `(x,_)`. By doing this, we get the compiler to ensure that we do not use the `y` position.

```
StartSlider (x,_) ->
    { model | value = if x < -50
        then 0
        else if x > 50
            then 100
            else x + 50
    , dragging = True
    }
EndSlider -> { model | dragging = False }
```

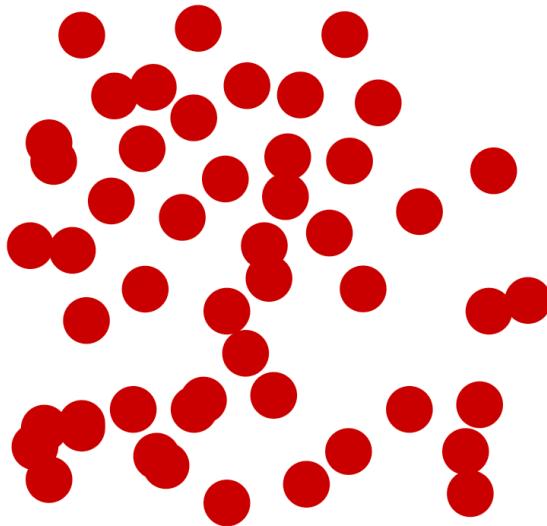
Finally, we set the `dragging` state on `StartSlider` and `EndSlider` but we didn't need to on the `Slider` message above.

There we have a working slider<sup>32</sup>. We leave it to you to make it pretty.

## 4.11 Drag and Drop

[to contents](#)

Now we can apply what we have learned to Drag and Drop, but we will have to take into account both  $x$  and  $y$  coordinates this time, and we will also need to “remember” where we “grab” the thing we are dragging, so that when we move it, the grabbed position on the `Shape` stays the same.



Since this is a bit more complicated, we create a `State` type:

```
type State
= Waiting
| Dragging
  (Float,Float)
```

in which we are either `Waiting` or, if we are `Dragging`, then we keep track of the difference between the mouse pointer coordinate and circle centre. We could ignore this difference, but the circle would jump to the click position when first clicked.

We store this information, along with the current position of the circle and the time (in case we want to animate later).

```
type alias Model
= { time : Float
, pos : (Float,Float) -- circle centre if not dragging, or ←
    the mouse click point if dragging
, state : State
}
init = { time = 0, pos = (0,0), state = Waiting }
```

<sup>32</sup>slider: <https://cs1xd3.online>ShowModulePublish?modulePublishId=fa835fc3-536d-4e98-b034-a32bc3f7275b>

To change states, we need to know if, when `Waiting`, the mouse is clicked on the circle, and when `Dragging`, if the mouse moves or they `LetGo`:

```
type Msg
  = Tick Float GetKeyState
  | NewPos (Float,Float)
  | MovePos (Float,Float)
  | LetGo
```

Now we attach these messages to two shapes

1. the circle, obviously! but only for the mouse down, not for moving;
2. a big invisible rectangle, for moving and letting go, because you could drag the mouse faster than your computer can move the circle.

```
myShapes model =
  [
    circle 20 |> filled red
    |> move ( case model.state of
```

Here we have to calculate the position differently if we are dragging, because we track the mouse position, and need to add the offset between the click position and the circle centre.

```
          Waiting -> model.pos
          Dragging delta -> add delta model.pos
        )
    |> ( case model.state of
```

We can only detect mouse clicks when we are `Waiting`.

```
    Waiting ->
      notifyMouseDownAt NewPos
    Dragging _ ->
      identity
```

The `identity` function has the same output as its input, i.e., it doesn't do anything! Why do we need it?

```
  )
]
++
( case model.state of
  Waiting -> []
  Dragging _ ->
    [rect 190 126 |> ghost
```

This `ghost` rectangle is only “drawn” when `Dragging`. We attach multiple messages to it, and we have to be careful, because we could get two of them in an unexpected order. In this case, we could get the `LetGo` first and then the `MovePos`, even though `LetGo` switches us to `Waiting`, and this rectangle and these messages won't exist. However, in between the time when `LetGo` is sent, and `update` processes it, a `MovePos` message could be generated.

```
    |> notifyMouseMoveAt MovePos
```

```

        |> notifyMouseUp LetGo
        |> notifyLeave LetGo
    ]
)

```

We define helpful functions for adding and subtracting coordinates. We will learn more about “vectors” in the next chapter.

```

sub (x,y) (u,v) = (x-u,y-v)
add (x,y) (u,v) = (x+u,y+v)

```

Everything happens in the update function. It’s so exciting!

```

update msg model
= case msg of
    Tick t _ -> { model | time = t }

```

When the mouse button is pressed, we get NewPos with the coordinates of the mouse pointer.

```

NewPos pos ->
case model.state of

```

We ignore this message if we are not Waiting for it (see above). We subtract the mouse position from the circle centre, and keep this in the Dragging state, so we can draw the circle in the right place.

```

Waiting ->
{ model | pos = pos
          , state = Dragging (sub model.pos pos)
}
_ ->
model
MovePos pos ->
case model.state of

```

When in the **Dragging** state, we get **MovePos** whenever the mouse moves. Actually, it will depend on how fast your computer is, it might *only* happen 30 times a second, not *whenever* your mouse moves. Your computer is busy and can’t be checking the mouse continuously. We update the **position** in the model.

```

Dragging _ ->
{ model | pos = pos }
_ ->
model
LetGo -> case model.state of

```

When we **LetGo** (or drag off the edge of the rectangle) we switch back to **Waiting**, and calculate the new circle centre by adding the offset to the last position we have for the mouse.

```

Dragging delta ->
{ model | pos = add model.pos delta
          , state = Waiting
}

```

```

    }
- ->
model

```

Whew! That was a lot of code<sup>33</sup>. If you are not sure you understand it all, try changing things which are still mysterious, and see what breaks!

If you are not too worried about violating causality in the time-space continuum, and you read ahead in the Containers section to learn about [List](#), then now would be a good time to try to make a game where you have not one, but a whole bunch of red circles, each of which is individually draggable, and where the goal is to put a target percentage on the left side of the centre line. If you get stuck, have a peek<sup>34</sup> at the code behind the screenshot at the start of this section.

When your game goes viral, remember where the infection came from!

## 4.12 Composing Model-View-Update Modules

[to contents](#)

What if your friend creates a mini-game, and you want to include it in your game? How do you do that? Is there room enough in one app for two [Models](#), [views](#) and [updates](#)? We can manage any number of submodules by importing one or more modules into another module.

We'll start with a version of the two ships example from Section 4.7. The best thing about building a GraphicSVG app out of multiple modules is that we can compile and test each module individually as a stand-alone app. This makes development so much easier, and allows teammates to work in parallel. But before we put them together, it is important to make sure that

1. We define a [Model](#) type and use it in type signatures like

```
myShapes : Model -> List (Shape Msg)
```

and

```
update : Msg -> Model -> Model
```

Since our [Model](#) is usually a record, and record types can be inferred without a type declaration, we don't need to do this to get our [module](#) to work alone, but it will make it much easier to get it working when imported.

2. We define functions for extracting, querying, or recreating our [Model](#) type.

In the two-ships case, we have

```
type alias Model =
  { time : Float
  , playerL : String
  , playerR : String
  , posL : (Float,Float)
  , posR : (Float,Float)
  , vell : (Float,Float)
```

---

<sup>33</sup>DragAndDrop: <https://cs1xd3.online>ShowModulePublish?modulePublishId=9cefba80-2c1f-4cdb-a9ad-2f042a6931d7>

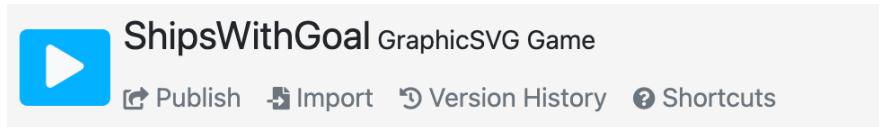
<sup>34</sup>multiple drag: <https://cs1xd3.online>ShowModulePublish?modulePublishId=559ec2cc-e0c9-4d26-b097-46d10c08845b>

```
, velR : (Float, Float)
, angleL : Float
, angleR : Float
, win : Maybe String
}
```

and can access `model.win` any time `model` is in scope, but it is a good design practice not to access record fields directly from other modules, but rather to use a function, such as

```
didWin : Model -> Maybe String
didWin model = model.win
```

to do so. Within the module, this might not be a big deal, but if we do the “easy” thing in importing modules, then whenever we change that field in the record, we will need to change all the modules which access it. Better to create and use the helper function `didWin` so that we can restrict changes to the single module. Let’s see how it works:



You remember the Publish link, we’ll need that, and to create a New (and new *type of*) share link:

[Show All](#) [New](#) [Close](#)

This time, we will publish for ourselves! But instead of choosing a fork or view link, we will create an import link.

## Publish Module

Publish to

Specific user

Publish scope

Ability to Import

The permission do you give the other user to do with your code.

Share with:

ProfAnand

Filter

Clear

ProfAnand

Publish

This will open a new page, which tells you it created a link, but that's it. You can't use it to do the import. Instead, you need to find it from the import pane, which you open with the button right next to the Publish button we just used. This will open

## Import Modules

X

Goal

Filter

Clear

Module Name	Version	Publisher	Action
ShipsWithGoal	3	ProfAnand	<a href="#">Import Module</a>
ShipsWithGoal	14	ProfAnand	<a href="#">Import Module</a>

but if you try it now, you won't have anything to import. Well, you may have too many importable modules, but when you filter out by the one you are looking for it won't be there, because you haven't created it yet, and we cannot publish our modules for importing to everyone, only to specific users or specific groups. So you should either write your own modules or try it out by forking mine<sup>35</sup>.

<sup>35</sup>two import: <https://stabl.rocks>ShowModulePublish?modulePublishId=b2b0e80f-7d56-449b-92a4-d5f3259fae3e> main: <https://stabl.rocks>ShowModulePublish?modulePublishId=b80e90c8-a07e-4de5-b155-9ec9272b43fd>

Now, when you import `ShipsWithGoal`, you are telling the WebIDE to compile the two modules together<sup>36</sup> and implicitly placing an `import` above the first line of your module. Since repeating `imports` is not an error, I'll repeat it here:

```
import ProfAnand.ShipsWithGoal
```

We now have access to the type, value, and function definitions of the imported module. To use them safely, I recommend the following steps:

1. In

```
myShapes model =
[  
  case model.state of  
    A           ->  
      button 40 "To B" ToB  
    AWon winner ->  
      button 100 (winner++" won! Play Again?") ToB
```

when we get to the state where the mini-game will be played, we can use the `myShapes` function from that module. The only difference is that we need to “qualify” the function, by prepending `YourUsername.ShipsWithGoal.` in front of `myShapes`. Just like with a local function, you can mix it in with other shapes—in this case a button.

```
B           ->  
  [ button 40 "To A" ToA  
    |> move (76,56)  
  , ProfAnand.ShipsWithGoal.myShapes model.shipsModel  
    |> group
```

In order to mix messages generated by the submodule from messages created by the main module, you need to wrap them in a constructor which we will add to the main `Msg` type below:

```
|> GraphicSVG.map ShipsMsg  
]  
|> group  
]  
  
type State = A | B | AWon String
```

2. To the existing message constructors, you need to add a *wrapper* constructor, which we will call `ShipsMsg`

```
type Msg = Tick Float GetKeyState  
          | ToA  
          | ToB  
          | ShipsMsg ProfAnand.ShipsWithGoal.Msg
```

---

<sup>36</sup>If you install Elm on your computer, this corresponds to adding to the list of modules to compile in `elm.json`.

The messages get wrapped, using `GraphicSVG.map` above, which finds all our `notify*` functions, no matter how many `groups` they are nested inside.

3. Next, we need to include `all` submodule states inside our `Model` type. This is because everything about the state of the app has to be here! There are actually two ways of doing this. We can include it directly in the `Model`

```
type alias Model = { time : Float
                    , state : State
                    , shipsModel : ProfAnand.ShipsWithGoal.Model
}
```

like we do here, or you can include it in some of the states in the `State` type. The latter makes sense if the submodule is only active in some states, and we don't need to save its state in between times when it is active. Since we may want to allow pausing the game and coming back to it, I have put it in the `Model`, but as written, we could have put it in the `B` constructor.

4. In the `update` function,

```
update msg model =
  case msg of
```

if the submodule uses animation or keyboard controls (or may in the future), we need to update the submodule state when we get a tick message.

```
    Tick t getKeyState
    -> let
```

We need to use the submodule's `update` function to do this, but the tricky thing is that we cannot pass in the `Tick` message we just received because, although the constructors are named the same, and have the same arguments, they have different types. One has the type `ShipsWithGoal.Msg` and the other `HasMiniGame.Msg`. But since the arguments are the same, we can construct the submodule's version of `Tick` using the date we just received in this module's `Tick`:

```
    newShipsModel = ProfAnand.ShipsWithGoal.update
      (ProfAnand.ShipsWithGoal.Tick t ←
       getKeyState)
      model.shipsModel
  in
```

In this case, we want to check whether a player has won, so we define a variable for the new submodule's state and then use the function `didWin` to test for a winner, and take the appropriate action if so:

```
  case ProfAnand.ShipsWithGoal.didWin newShipsModel of
    Just wn -> { model | shipsModel = ProfAnand.←
      ShipsWithGoal.initWithNames "Patatuj" "Inaqan"
                    , time = t
                    , state = AWon wn
                    }
    Nothing -> { model | shipsModel = newShipsModel }
```

```
, time = t
}
```

Note that we reinitialize the game as soon as a player wins. We could have instead initialized the game on the `ToB` transition, below. An important thing is that it happen once between a player winning and restarting the game.

```
ToA -> { model | state = A }
ToB -> { model | state = B }
```

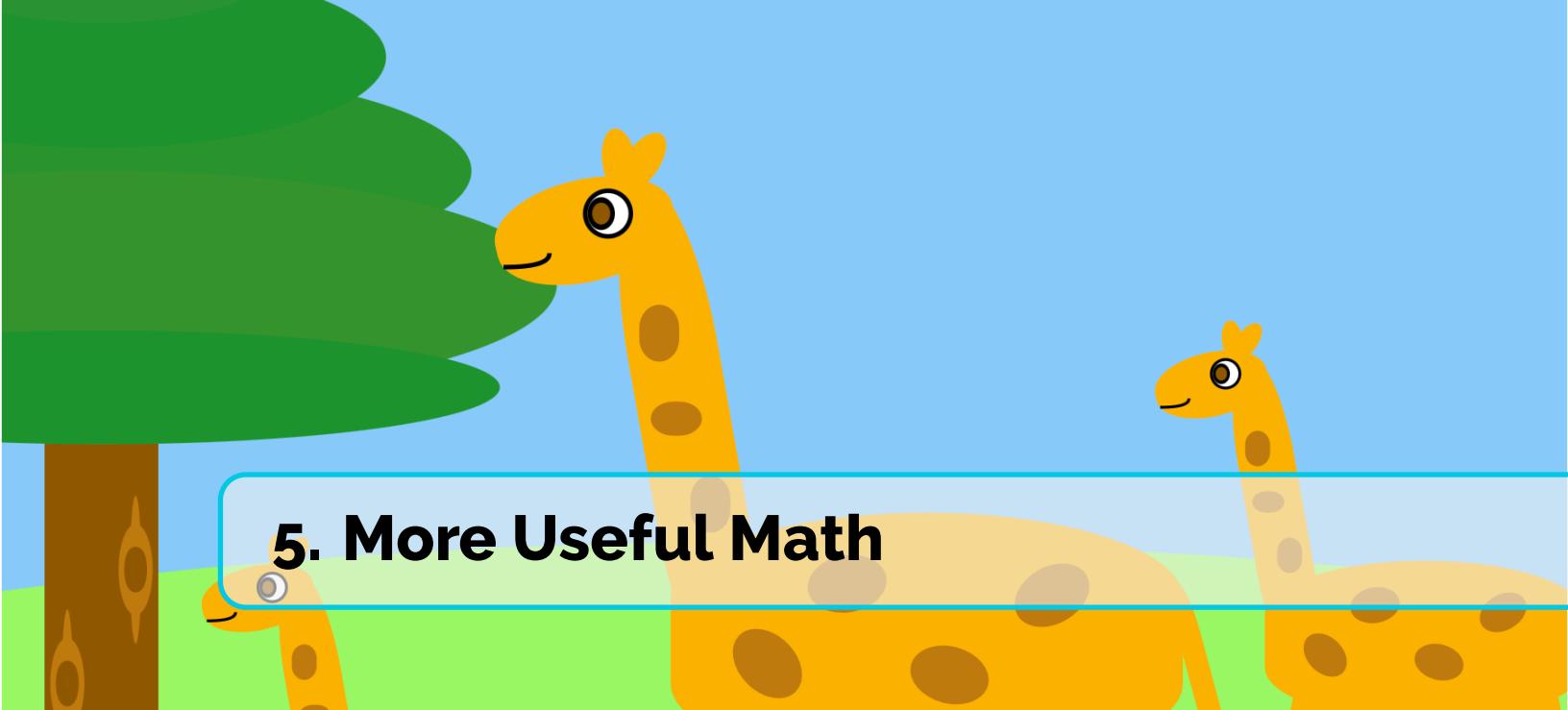
When you get a message generated by the submodule, unwrap it and call the submodule's `update` function to calculate the new submodule state, and store it in our new `model` value. In this case, we use the access function `didWin` to test if one of the players won, and apply a transformation to this module's `state`.

```
ShipsMsg shipsMsg
-> let
    newShipsModel = ProfAnand.ShipsWithGoal.update
        shipsMsg
        model.shipsModel
in
{ model | shipsModel = newShipsModel
, state = case ProfAnand.ShipsWithGoal.didWin ←
    newShipsModel of
        Just _ -> A
        Nothing -> model.state
}
```

Note that this particular submodule doesn't have any buttons or other controls to generate messages, so the only message is the `Tick` message which we handled above, so this case will never be evaluated. But it is still good practice to write this code so that if buttons are added to the submodule in the future, they will just work, because the necessary code is already in the main module.

5. Use the submodule's `init` function to initialize the copy of the submodule's `Model` contained in this module's `Model`:

```
init : Model
init = { time = 0
, state = A
, shipsModel = ProfAnand.ShipsWithGoal.initWithNames "←
    Patatuj" "Inaqan"
}
```



## 5. More Useful Math

In this chapter, we capture two more bits of useful math. The first math is called a Boolean Algebra<sup>1</sup>, which is a pattern that works like the `Bool` type, with values `True` and `False` and operations `not`, `and` (`&&`), `or` (`||`), and `xor`. If you don't know what these operations do, try filling in these tables:

<code>x</code>	<code>y</code>	<code>x &amp;&amp; y</code>	<code>x    y</code>	<code>x</code>	<code>not x</code>
<code>False</code>	<code>False</code>			<code>False</code>	
<code>True</code>	<code>False</code>			<code>True</code>	
<code>False</code>	<code>True</code>				
<code>True</code>	<code>True</code>				

These operations are really useful, and easy to learn and remember, which makes it even easier to learn new concepts (sometimes complicated ones) when they follow the same pattern. This pattern is called a Boolean Algebra. The next most complicated example is the *power set* of a set  $X$ , written as  $2^X$ , because the number of subsets is equal to  $2^{\|X\|}$  (two to the power the size of the set  $X$ ). Let's compare

Abstract Notation	Logic	Elm Logic	$2^X$	subsets of <code>bigSet</code>
0	false	<code>False</code>	<code>{}</code>	<code>Set.empty</code>
1	true	<code>True</code>	<code>X</code>	<code>bigSet</code>
$x \wedge y$	and	<code>(&amp;&amp;)</code>	$x \cap y$	<code>Set.intersect x y</code>
$x \vee y$	or	<code>(  )</code>	$x \cup y$	<code>Set.union x y</code>
$\neg x$	not	<code>not</code>	$X \setminus x$	<code>Set.diff bigSet x</code>

If you know some of the notation, like the notation for an empty set `{}` (also  $\emptyset$ ), good, otherwise, don't worry. Abstractly we have two special elements, 0 and 1. In basic logic, those are all the elements, and in Elm, they are `True` and `False`. In set theory, however, they are the empty set and the full set of which we are taking subsets. In Elm's `Set` module (which we have to import if we want to use it) they are `Set.empty` and whatever we call the set we start with. In the table, we call the full set `bigSet`.

---

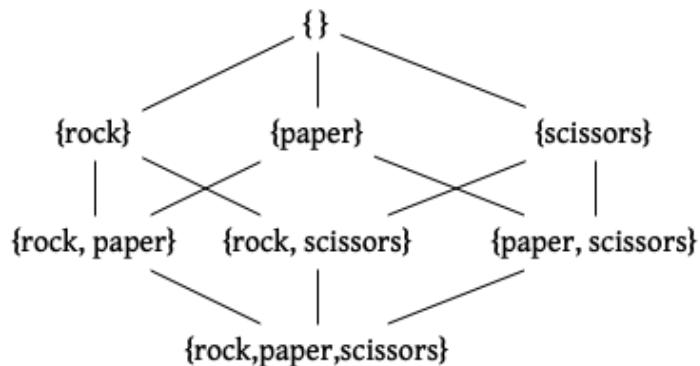
<sup>1</sup>See [https://en.wikipedia.org/wiki/Boolean\\_algebra](https://en.wikipedia.org/wiki/Boolean_algebra)

The operations in logic are *and* and *or*, also called conjunction and disjunction, which you just have to remember are double-symbol operators `&&` and `||`. Fortunately, most programming languages you are likely to use today use the same symbols for these operations, so you only have to learn them once<sup>2</sup>.

*Isn't it confusing to use 0 and 1, because some of us know them as numbers already?* Well, actually, there are many cases where symbols were reused in mathematics, especially after printing was invented, because it was just easier to use one that already existed. This got worse with early computers which had a tiny set of symbols. Today, we don't have this problem, but still 0 and 1 make sense if you think of  $\wedge$  as multiplication. When you `and` `False` with anything, you get `False`. When you intersect any set with the empty set, you get the empty set. When you multiply 0 by anything you get 0<sup>3</sup>. What about 1? Well, multiplying by 1 does give back 1!

I hope this helps you remember these new meanings for 0 and 1. But another way to think about it is to think about how we would store `True` and `False` in a computer. You may have heard that everything in a computer is stored in terms of numbers. This is kind of true, because that's how people made them, with a few exceptions like optical and quantum computers. Computers store information in chunks, and we know how much information is in a chunk by how many different values we can store. The smallest chunk can store two values, because being able to store one value doesn't count as information. It's like a weather forecast which is always sunny—you don't need to store it. Since there are only two logical values, we only need two numbers to count them. Why do we start with 0 instead of 1 in our counting? This is because a lot of calculations are simpler if we count this way, so in the interest of saving electricity and getting things done faster so we can go home and play on our computers<sup>4</sup>, computer scientists pretty much always count starting from 0.

Remembering  $\wedge$  and  $\vee$  is easy if you think about organizing a diagram of all the subsets of a set, with sets drawn below all sets they contain. Like this:



<sup>2</sup>Why do we use these, and why are they doubled??? Most programmers think this notation was invented with the C language, but, in fact they go back to B and its progenitor BCPL, for which the 1967 manual is available online, see <https://www.bell-labs.com/usr/dmr/www/bcpl.html>.

<sup>3</sup>Mostly true, unless your numbers have infinity. What is  $0 \times \infty$ ? Computer implementations of arithmetic have agreed this is *not a number* (`NaN`), see [https://docs.oracle.com/cd/E19957-01/806-3568/ngc\\_goldberg.html](https://docs.oracle.com/cd/E19957-01/806-3568/ngc_goldberg.html) for an in-depth discussion of floating-point numbers, including `NaNs`.

<sup>4</sup>This is not a good thing! Please go outside and think about how much information is stored in the structure of a single tree.

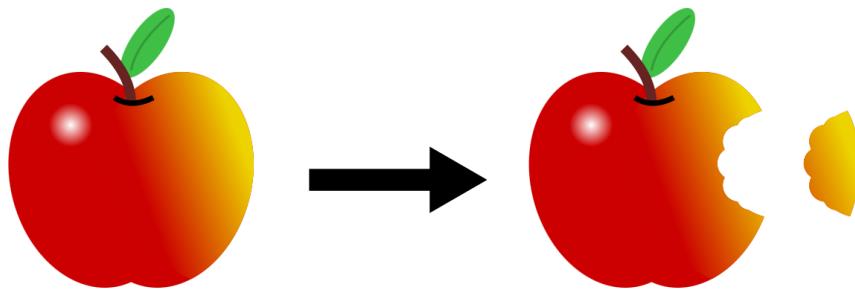
In this diagram,  $\{\text{rock}\} \wedge \{\text{scissors}\}$  is the set that connects to  $\{\text{rock}\}$  and  $\{\text{scissors}\}$  in an upward pointing  $\wedge$ . This particular diagram may look like a cube to you. All power sets look like cubes, but this set with three elements is a 3D cube, so you recognize it. Try drawing a 2D cube, or a 4D cube, and see if you recognize them.

This kind of diagram has another name, it is called a lattice<sup>5</sup>. We can define the lattice of all subsets of a set with  $n$  elements to be an  $n$ -dimensional cube lattice. So if you draw the diagram for 5 things, and your parent asks you to clean up your room, you can legitimately say that you are too busy working in 5 dimensions to bother with 3-dimensional rooms.

## 5.1 Clipping and Shape Math

[to contents](#)

Ok, that was a lot of theory! How can we use it for practical problems, like *How do we draw an apple with a bite out of it?* It turns out that the set-theory analogy works. Think about an apple shape as a set of points, which is a subset of all the points on your canvas. To have a bite, we need to have a mouth. So think about the set of points inside that mouth. The bite is the intersection of the set of points in the apple and the set of points inside the mouth. And the apple without the bite is the set of points in the apple, minus the set of points in the bite, but we could also say it is the subtraction of all the points in the mouth (whether they are in the apple too, or not).



Let's try it<sup>6</sup>! First, define **Shapes** for the apple and the bite. The colour of the bite doesn't matter. Try changing it to see.

```
apple = circle 20 |> filled red
bite = circle 10 |> filled black
|> move (20,0)
```

Next we take a bite *out* of the apple with **clip**, which is similar to  $\wedge$ ,  $(\&\&)$ ,  $\cap$  or **Set.intersect**:

```
biteOfApple = apple |> clip bite
```

Then we take a bite *away* from the apple with **subtract**, which is similar to  $\neg$ , **not**,  $X \setminus x$  or **Set.diff**:

---

<sup>5</sup>See [https://en.wikipedia.org/wiki/Lattice\\_\(order\)](https://en.wikipedia.org/wiki/Lattice_(order)) where, unfortunately, they draw the lattices upside down (to us).

<sup>6</sup>apple: <https://cs1xd3.online>ShowModulePublish?modulePublishId=d41b5773-2c23-4810-888d-6d658a9565df>

```
appleMinusBite = apple |> subtract bite
```

Now we can animate the bite being taken:

```
myShapes model =
  [ apple
    |> move (-55,0)
  , bittenApple
    |> move (40,0)
  , applePiece
    |> move (60 + 20 * sin model.time,0)
  , arrow |> move (-10,0)
  ]
```

Ok, that wasn't exactly the apple we showed you, but if you are interested, you can have a peek<sup>7</sup> at the code which does it.

We have seen `clip` and `subtract`. In total there are five functions for image logic:

```
clip      : Shape userMsg -> Shape userMsg -> Shape userMsg
union     : Shape userMsg -> Shape userMsg -> Shape userMsg
subtract  : Shape userMsg -> Shape userMsg -> Shape userMsg
outside   : Shape userMsg -> Shape userMsg
ghost     : Stencil -> Shape userMsg
```

Whereas `clip` colours in the intersection of two `Shapes`, `union` colours in the... *union—surprise!!!*

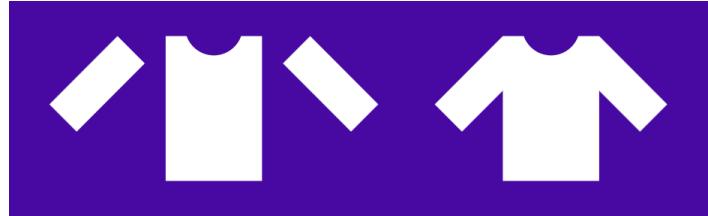
When drawing something complicated, it is useful if we can visualize a `Shape` as the union of simpler `Shapes`—a great example of Dividing and Conquering. For example, a t-shirt is basically a rectangular body and two rectangular arms, with an optional cut-out for people with heads.

```
-- we can make a t-shirt...
tshirt =
  rect 20 30 |> filled white
-- ...by adding a sleeve...
  |> union
    ( rect 20 8 |> filled white
      |> move (-10,-4)
      |> rotate (degrees 45)
      |> move (-10,15)
    )
-- ...and another sleeve...
  |> union
    ( rect 20 8 |> filled white
      |> move (10,-4)
      |> rotate (degrees -45)
      |> move (10,15)
    )
```

---

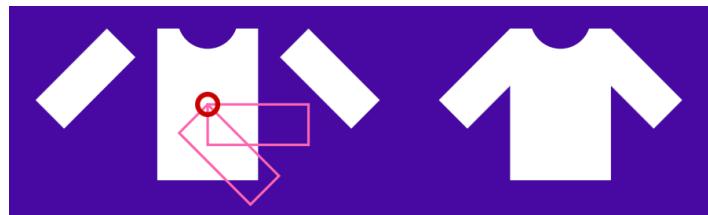
<sup>7</sup>better: <https://cs1xd3.online>ShowModulePublish?modulePublishId=0f2ae556-8168-4ce4-879f-fc684828a168>

```
-- ...and removing the neck hole
|> subtract
( circle 6 |> ghost |> move (0,17) )
```



Before we go on, you might be wondering why we have two `moves` for each sleeve, before and after the `rotate`???

The following picture may help:



The rectangles outlined in pink show the sleeve rectangle after `|> move(10, -4)`, which moves the top-left corner of the rectangle to the middle point  $(0,0)$ , so that when we do the `|> rotate(degrees -45)` that corner stays where it is, giving us the second outlined rectangle. Why move by  $(10, -4)$ ? Because those are half the width and height of the rectangle we start with. We picked that corner, because we want to sew it on to the top right corner of the main rectangle, so now we just have to move it there from the centre with `|> move(10, 15)`.

Now, if you are the suspicious type, you may be suspecting that we didn't really need `union`, because we have `group` to put `Shapes` together. If so, you have a career in espionage, because you are right! This is the code which drew the t-shirts on the right.

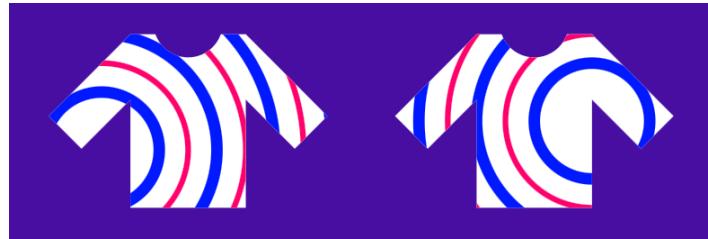
```
groupieShirt =
[ rect 20 30 |> filled white
, rect 20 8 |> filled white
|> move (-10,-4)
|> rotate (degrees 45)
|> move (-10,15)
, rect 20 8 |> filled white
|> move (10,-4)
|> rotate (degrees -45)
|> move (10,15)
]
|> group
|> subtract
( circle 6 |> ghost |> move (0,17) )
```

But if you want `union` and `subtract` to line up so you can see them being applied one after another, you need to use `union`, because `group` doesn't work the same way.

Did you go back to check that the t-shirts really all look the same? Well, we can fix that, and this is really how `clipping` got famous. We can use it to “paint” another shape with a pattern we've already drawn:

```
-- first we draw the tshirt
[ tshirt
-- then we draw the pattern...
, pattern
    -- but we only draw it on top of the tshirt!
    |> clip tshirt
]
|> group
```

Pretty clever! You can use different patterns for each shirt, or just move them around:



Before you invest in a t-shirt factory, we should fix a problem which could bankrupt your business. In the code for the patterned t-shirt, the variable `tshirt` is used twice. In the rush to fill an order for 20 different t-shirts, you are bound to copy and paste a different t-shirt shape in one place, but not the other, resulting in a pattern not matching the t-shirt. But, don't worry, we can fix this by creating a function

```
silkScreen rawShirt withPattern =
[ rawShirt
, pattern
    |> clip rawShirt
]
|> group
```

You can try it out<sup>8</sup>. Using functions to avoid repetitive code, especially when variables or segments of code are repeated, is a good programming practice. It is pretty easy to do in a functional language like Elm, even for complicated bits of code.

To finish off this section, we need to cover `outside` and `ghost`. In a way, we should have covered `outside` before `subtract`, because `outside` is the more basic operation, playing the role of `not`. A pixel is part of the `outside` of a `Shape` if it is not in the `Shape`. That's just the definition of outside in English. Notice that even in English, the definition of “outside” uses “not”.

---

<sup>8</sup>t-shirts: <https://cs1xd3.online>ShowModulePublish?modulePublishId=1e299922-0ba4-4f6b-b1fc-951a335da5ae>

If you're scared to find out what `ghost` does, don't be! We've already snuck it into the definition of the t-shirt above! It is just a shortcut function, for when we want to define a `Shape` which is only used for one of the other image operations, and never drawn itself. It just saves us from having to pick a colour for the filling or outlining which we will never see. It isn't that much extra typing, but picking a colour nobody will ever see can be very distressing to philosophy and physics students who worry about whether the colour can exist without an observer.

Now it's time to see where we stand in terms of logic and set theory? We'll redraft the table of operations, replacing the column for Elm Logic with a column for `Shape` math:

Abstract	Logic	Shape Math	$2^X$	subsets of <code>bigSet</code>
0	false	<code>group[]</code>	$\{\}$	<code>Set.empty</code>
1	true	<code>rect 192 128 &gt; filled red</code>	$X$	<code>bigSet</code>
$x \wedge y$	and	<code>&gt; clip</code>	$x \cap y$	<code>Set.intersect x y</code>
$x \vee y$	or	<code>&gt; union</code>	$x \cup y$	<code>Set.union x y</code>
$\neg x$	not	<code>&gt; outside</code>	$X \setminus x$	<code>Set.diff bigSet x</code>

Not bad! The analogy helps make sense out of `union`, right away, and `clip` after you think about it a bit. Just like logical not<sup>9</sup>, if you apply it twice, `outside` gets back the inside, where you started. We decided not to have an `empty` function giving a `Shape` with nothing in it, because `group[]` already does that, and doesn't take long to type. You can argue whether `everything` is `rect 192 128`, but in the Animation Activity, it definitely is. Why is it `red`, well we picked it because it is short and easy to see!

There is a lot you can do with these simple operations by composing them together. This is the sign of a good language design. We make a lot with a little, by stirring in a little mathematical reasoning.

## 5.2 Following A Path

[to contents](#)

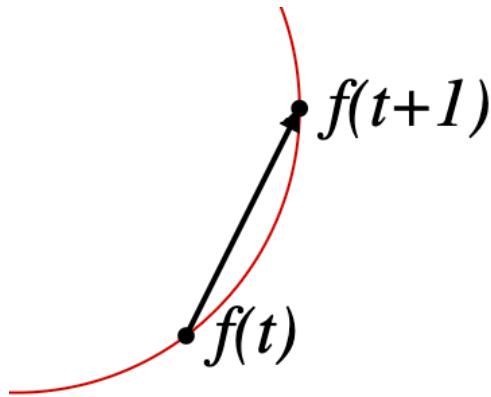
Another useful mathematical concept is that of slope or derivative. Imagine you are sitting on a point travelling on a curve defined by a function of time

$$f(t) \in R^2 \quad (5.1)$$

If we want to face forward, we need the direction of travel. We can figure that out by taking the slope of a line tangent to the curve. If we know the curve  $f$ , then we may also know the slope, but what if we are working on a library to draw the point and the passenger, and we could be called with many different functions? In that case, we can approximate the direction of the curve by looking at two points on the curve:

---

<sup>9</sup>In English it is frowned on to string “not”s together, because non-logicians find it confusing!



From this, we can calculate the slope as rise/run. In many cases, we want the angle of this slope, and fortunately, we have a function `atan2` which calculates this angle. Obviously, we will get a better answer if the two points are closer together—as long as they aren’t so close that the precision of `Floats` gets in the way. For graphics, picking a “small” value like  $1/100$  should work, but in Section 9.2 we show how we can calculate the exact slope using derivatives.

```
angleOfMotion xyFun t =
  let
    (x0,y0) = xyFun t
    (x1,y1) = xyFun (t+0.01)
  in
    atan2 (y1-y0) (x1-x0)
```

The function `atan2` is related to the trigonometric function `tan`, which would tell you the slope, given the angle, except that `atan2` works for slopes of  $\infty$ , and distinguishes between the two directions represented by a line. We are used to our executable functions being imperfect approximations of the mathematical functions, because our `Float` numbers are only limited-precision versions of real numbers, but this time, the computer science function is actually better! Almost all programming languages have `atan2` in their standard libraries, so look for it at a theatre near you.



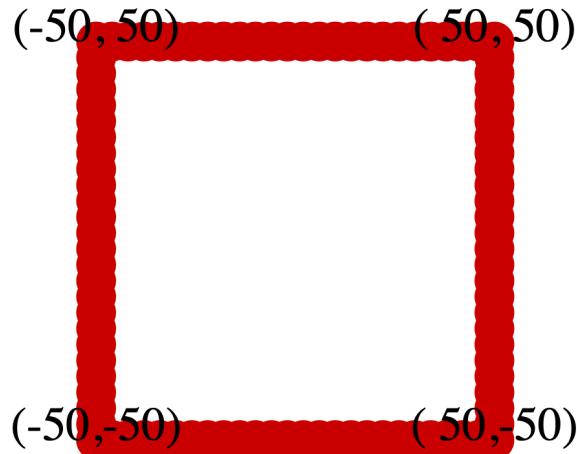
How much fun you have with it<sup>10</sup> depends on how creative you can get with mathematical functions!

Now, what if you don't know how to write your path as a function, but you could write pieces of it as a function? Good news we have

```
animationPieces : List ( Float, Float -> a )
    -> ( Float, Float -> a )
    -> Float
    -> a
```

which puts together pieces of animation. It works by putting your animations in a list, along with how long each of them should take.

Let's say we want to trace out a square with corners ( $\pm 50, \pm 50$ ):



We can do this with four segments, with a function `Float -> (Float,Float)` tracing each segment. So in this case, the type `a` in the type signature for `animationPieces` is `(Float,Float)`:

```
path time =
  animationPieces
  [ (5,\ t -> (-50           , -50 + 20 * t))
  , (5,\ t -> (-50 + 20 * t,   50))
  , (5,\ t -> ( 50           , 50 - 20 * t))
  , (5,\ t -> ( 50 - 20 * t, -50))
  ]
  (\ t -> (-50,-50))
  time
```

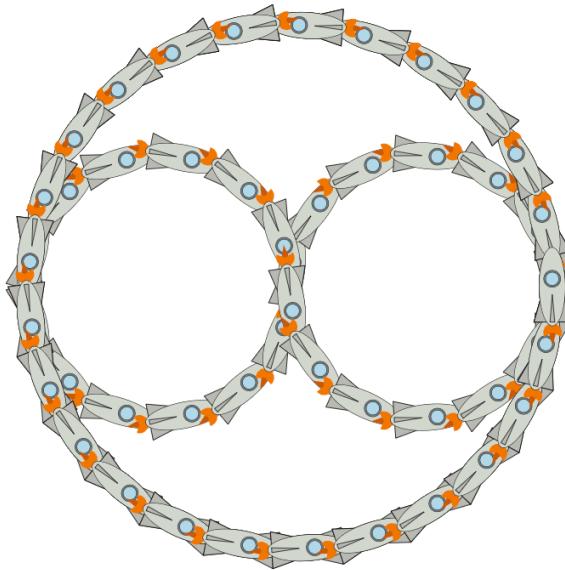
Each segment takes 5 seconds, and after the four segments, the function `\ t -> (-50, -50)` means that we stay where the last segment ends (and where the first segment began). This shows the advantages of being able to work with functions as values, putting them in tuples, and of having polymorphism, so that `animationPieces` can be used to animate positions, radii, colours, pretty much anything!

<sup>10</sup>rockets: <https://cs1xd3.online>ShowModulePublish?moduleId=2b6bbad8-c608-4dd6-ada0-6bec72bbaa67>

If you are a bit rusty creating paths, and matching up segments, we created a helper function which switches between animating and tracing out the whole path by plotting 100 time points on top of each other. In the code<sup>11</sup> you change this variable to control this behaviour:

```
wholePath = Just 20 -- trace out the whole path from 0s to 20s
-- wholePath == Nothing -- do animation
```

As a challenge, try to put what you've learned together to help the Celtic Space Agency program their first rocket trajectory:



You will need to use the fact that  $t \mapsto (\cos t, \sin t)$  traces out a circle in  $2\pi$  seconds. So it traces out half a circle in  $\pi$  seconds, and you can shift, scale and reflect in the  $x$ -axis to build all the pieces needed. But if you get stuck, you can peek<sup>12</sup> at the answer.

## 5.3 Animation via Interpolation

[to contents](#)

Let's look at the square path example again, because it is an example of a general process called interpolation, but not written in a way to make that clear. We can rewrite the formula for the  $y$ -coordinate of the first path as

$$-50 + 20t = -50 \cdot (1 - t/4) + 50 \cdot (t/4) \quad (5.2)$$

In this case, we factor out  $t/4$ s because the path takes 4s, and we factor out  $-50$  and  $50$ , because those are the starting and ending points for the animation.

In general, if we want to interpolate between two numbers,  $a$  and  $b$ , as a variable  $u \in [0, 1]$  goes between 0 and 1, we use

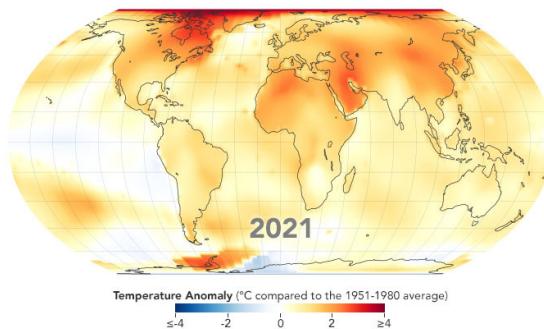
$$(1 - u)a + ub \quad (5.3)$$

<sup>11</sup>square: <https://cs1xd3.online>ShowModulePublish?modulePublishId=2e1f9fc3-2bd0-45fd-a91d-44237ff62830>

<sup>12</sup>Celtic: <https://cs1xd3.online>ShowModulePublish?modulePublishId=0fd9bf70-9bb8-4d8d-9bbd-6c8ef95dea0e>

Notice that the two factors  $1 - u$  and  $u$  add up to 1. Two functions with this property are called a partition of unity, and we can construct curved versions which lead to smooth interpolations.

When we interpolate points, we interpolate both the  $x$ - and  $y$ -coordinates. But we can interpolate a bunch of points. This is how you can create smooth animations of data like the global temperature anomaly<sup>13</sup>:



Let's look at simpler data. Say we want to animate sales data from year to year? A safe version of our interpolation function,

```
interp : Float -> Float -> Float -> Float
interp u a b =
  if u < 0 then
    a
  else if u < 1 then
    (1-u)*a + u*b
  else
    b
```

can be mapped over two sets of data to map each month's data separately:

```
List.map2 (interp model.time)
oldData
newData
```

Recall that supplying one argument makes `(interp model.time)` a function with two variables instead of three—freezing the animation time for all months in the chart. This is then used to combine the `oldData` and `newData` using `List.map2` from Section 6.4. You can try<sup>14</sup> it with your own data. After you have read more about `Lists`, why not try making a bar chart version, or adding points to the line graph.

<sup>13</sup>anomaly: <https://earthobservatory.nasa.gov/world-of-change/global-temperatures>

<sup>14</sup>interp months: <https://cs1xd3.online>ShowModulePublish?modulePublishId=e1f6b755-43b9-4260-9a75-b73ac697f598>

## 5.4 Animation in Vector Spaces

[to contents](#)

Let's look at the formula for interpolation again,

$$(1-u)a + ub \quad (5.4)$$

and write it in terms of functions

```
add
  (scale (1-u) a)
  (scale u b)
```

We see that it requires two functions

```
add x u = x+u
scale s x = s*x
```

But we could equally well write functions to act on both coordinates of a 2D point:

```
add (x,y) (u,v) = (x+u,y+v)
scale s (x,y) = (s*x,s*y)
```

We can now interpolate between two points, but you will probably want to interpolate a path along a list of points, so we are reaching into Chapter 7 this time to write a function which takes a list of points and turns it into a list of interpolated paths of the type used by `animationPieces`

```
mkAnimationPiece listOfPoints =
  case listOfPoints of
    p0 :: p1 :: rest -> (1, \ t -> interp t p0 p1 )
                           :: ( mkAnimationPiece (p1 :: rest) )
    _ -> []
```

Don't worry if you cannot write this function on your own at this point. You can see the heart of it,

```
(1, \ t -> interp t p0 p1 )
```

makes an interpolated path. If you don't believe it is that easy, try<sup>15</sup> for yourself. As a challenge, get the rocket ship to follow the path, and try to predict what will happen as it goes around corners by drawing a diagram with chords.

It turns out that a set with operations `add` and `scale` satisfying some simple rules, namely

```
add x (add y z)   ≡ add (add x y) z
scale u (add x y) ≡ add (scale u x) (scale u y)
```

(which are satisfied for numbers and coordinate pairs) is called a *vector space*. So we can interpolate in any vector space. We just have to figure out `add` and `scale` for that data type.

For people who write graphics or scientific code, it is annoying that vector spaces are not built into programming languages the way numbers are. It is not that hard for us to get this functionality in Elm. For example, we can rewrite `interp` as

---

<sup>15</sup>points: <https://cs1xd3.online>ShowModulePublish?modulePublishId=4428b3d5-b535-4f04-9691-294636ecb459>

```
interpVS addVS scaleVS u a b =
  if u < 0 then
    a
  else if u < 1 then
    addVS
      (scaleVS (1-u) a)
      (scaleVS u b)
  else
    b
```

and by making the vector-space operations arguments of the function, we can use the same `interpVS` for any vector space. We can even create complex functions for manipulating points in vector spaces, and as long as the functions pass along the `addVS` and `scaleVS` functions to each other, they can work together in complex ways, and continue to work on any future vector spaces you dream up.

To avoid the passing around of the basic operations, other languages have mechanisms to hide them. If you were writing in Haskell, you would make `VectorSpace` a class, and if you were writing in Java, you would make it an abstract class. But whether you have these mechanisms or not, a dark secret of programming with vector spaces, or many other useful mathematical objects, is that we have to trust that the implementor checked that the required properties hold. If vector spaces were baked into the language in general, or at least for a few special cases, we could have the compiler enforce the properties.

If you don't think this is a problem, take this version of the code<sup>16</sup> and try adding a bug to `addVS` or `scaleVS` and see what happens.

At this point, you can create animations where swarms of particles move through configurations that spell out different words, or form ghostly visages, but you will also want to try out `curves`, so here's a crash course.

A `curve` goes through a list of points `[p0, p1, p2, ..., pN]`, but it doesn't go in a straight line. In between points `p0` and `p1` it is pulled towards `q1`, while between points `p1` and `p2` it is pulled towards `q2`, and so on. We write it as

```
firstCurve =
  curve   p0
  [ Pull p1 q1
  , Pull p2 p2
  , ...
  , Pull pN pN
  ]
```

For example, we can make a strand of kelp using

```
seaweed0 =
  curve   (-thickness      ,-50)
  [Pull (-thickness + 10,-25) (-thickness,  0)
  ,Pull (-thickness - 10, 25) (           0, 50)
  ,Pull (           0       , 50) (           0, 50)
```

<sup>16</sup>interpVS: <https://cs1xd3.online>ShowModulePublish?modulePublishId=e9c0c402-acb1-41cd-ab85-cb8d006dbc03>

```

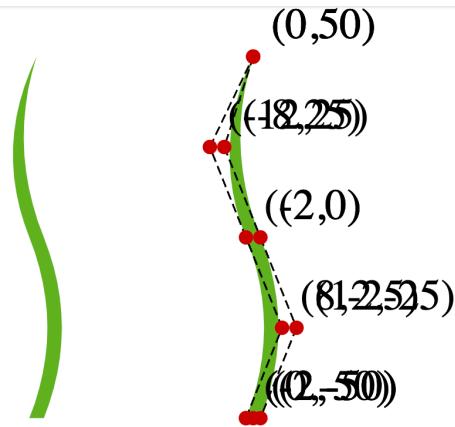
, Pull ( thickness - 10, 25) ( thickness, 0)
, Pull ( thickness + 10, -25) ( thickness, -50)
, Pull ( 0 , -50) (-thickness, -50)
]

```

To help you figure out your curves, we have

```
curveHelper : Shape msg -> Shape msg
```

which draws all the points for the curve, and connects points on the curve to pulled points with dotted lines, like the one on the right:



Unfortunately, it isn't smart enough to avoid coordinates overlapping. If seeing the coordinates is necessary, you may need to rotate by 90°.

Since curves are a bunch of points, and we know points form a vector space, you shouldn't be surprised that seaweed also forms a vector space<sup>17</sup>!

To make the code more reusable, we'll show that curves are vector spaces, starting by creating a type for the curve data:

```
type MyCurve = MyCurve (Float,Float) (List Pull)
```

(The `Pull` is already defined in `GraphicSVG`.) Then we define the operations for `Pulls`:

```

addPull (Pull p0 p1) (Pull p2 p3) = Pull (add p0 p2) (add p1 p3)
scalePull s (Pull p2 p3) = Pull (scale s p2) (scale s p3)

```

which we can use to define the operations for `MyCurves`:

```

addCurve (MyCurve p0 pulls0) (MyCurve p1 pulls1) =
    MyCurve (add p0 p1) (List.map2 addPull pulls0 pulls1)
scaleCurve s (MyCurve p1 pulls1) =
    MyCurve (scale s p1) (List.map (scalePull s) pulls1)

```

Since real oceans are constantly in motion, instead of interpolating once between two configurations, we can interpolate back and forth in a wave motion using `sinmodel.time`:

---

<sup>17</sup>Technically, only seaweed with this number of `Pull` points, but that doesn't sound as good.

```
myShapes model =  
  [ interpVS addCurve  
    scaleCurve  
    (0.5+0.5*sin model.time)  
    seaweed0  
    seaweed1  
    |> drawCurve  
  ]
```

If you think the one kelp<sup>18</sup> is a bit lonely, it's up to you to create a kelp forest!

---

<sup>18</sup>one kelp: <https://cs1xd3.online>ShowModulePublish?modulePublishId=28666739-9362-478e-8b33-17aa8a40b83f>



## 6. Core Packages

Elm is a small language, and you can get most of your work done with a small core of packages <https://package.elm-lang.org/packages/elm/core>. Let's look at what they do. In some sense, most of the packages are containers in the sense that they describe data types which contain other data, and all of Elm can be described as mathematical, but we will differentiate by how they are used in programs.

### 6.1 Core Data Types, Math, etc.

[to contents](#)

#### 6.1.1 Basics

[to contents](#)

[Basics](#) is the Math module, plus odds and ends. It defines types for numbers, [Int](#) and [Float](#), which store integers and fractional numbers, respectively. Since computer memories are finite, they cannot store the mathematical versions of these numbers, which are infinite. Mathematical integers which go all the way from  $-\infty$  to  $\infty$  can only be stored as long as they fit in whatever storage type the Elm compiler uses on your computer. It is guaranteed to be at least  $-2^{31}, -2^{31} + 1, \dots, 2^{31} - 1$ , but it could be more, which may surprise you if you have used integer types in other languages where adding 1 to the largest representable number “wraps around” to give you the most negative representable number. All the usual arithmetic works, for example, but we use `*` for multiplication and for [Ints](#) only we use `//` for divide.

```
1  - 1 -> 0
1  * 1 -> 1
1  + 1 -> 2
10 // 4 -> 2
11 // 4 -> 2
12 // 4 -> 3
13 // 4 -> 3
14 // 4 -> 3
-1 // 4 -> 0
```

```
-5 // 4 -> -1
3 ^ 2 -> 9
3 ^ 3 -> 27
abs 1 -> 1
abs -1 -> 1
negate 1 -> -1
negate -1 -> 1
```

Remember that any of these operators can be used in the prefix (normal) function form:

```
(+) 1 1 -> 2
```

Note that `-` is really two functions:

```
(-) : number -> number -> number
```

and

```
prefix - : number -> number
```

which the compiler tells apart by whether there is a space in between the `-` and the number or variable, or not. Look carefully at the examples above.

Often, when we divide integers, we also want to take remainders. Elm provides

```
modBy 2 0 == 0
modBy 2 1 == 1
modBy 2 2 == 0
modBy 2 3 == 1

remainderBy 4 -5 -> -1
remainderBy 4 -4 -> 0
remainderBy 4 -3 -> -3
remainderBy 4 0 -> 0
remainderBy 4 1 -> 1
remainderBy 4 7 -> 3
```

`Float` numbers are also not all fractions, but only a subset, and one which is more complicated to describe. If you are familiar with scientific notation, i.e., a finite decimal fraction,  $1 \leq f < 10$  together with an exponent of 10:

$$2.022 \times 10^3 = 2022$$

then the concept is the same, except that base-two (binary) numbers are used. Numbers of the form

$$s\frac{f}{2^e}$$

can be represented, where  $s \in \{-1, 1\}$  is the sign (positive or negative),  $f$  is an integer  $2^{53} \leq f < 2^{54}$  and  $e$  is an integer  $-1022 - 53 \leq e \leq 1023 - 53$ . Why 53? It is part of the standard for floating-point arithmetic<sup>1</sup>. Standards allow programmers to write programs

---

<sup>1</sup>See [https://en.wikipedia.org/wiki/IEEE\\_754](https://en.wikipedia.org/wiki/IEEE_754) for the standard agreed by the Institute of Electrical and Electronics Engineers, and [https://en.wikipedia.org/wiki/Floating-point\\_arithmetic](https://en.wikipedia.org/wiki/Floating-point_arithmetic) for the history of floating point numbers, and how the Second World War this time slowed down progress on the first implementations by Konrad Zuse.

using floating-point numbers which give the same answers on computers made by different companies.

That is a *lot* of possible fractions, and in practice, it means that for most numbers which come up in calculations, we can find an approximation good up to the 15th decimal digit. That is more accurate than most people have the patience to calculate things, and before digital computers, there were only a handful of numbers, other than repeating fractions, with more digits— $\pi$  being the most important—known by mathematicians and scientists anywhere in history!

But it is still not all fractions, so you should be aware that you are using approximations, and this can give surprising results. There is a whole subfield of study of the resulting errors called numerical analysis.

As computers were made more and more complicated, their developers first standardized how to store and process integers, which was pretty easy, then they standardized floating-point approximations for fractional numbers. Unfortunately, they stopped short of standardizing how to deal with more general numbers. So although we know how to represent numbers like  $\sqrt{2}$ , have many ways of calculating floating-point approximations, and using them in exact calculations, we do not have a standardization. In fact, very few languages have support for them, and when they do, they are part of support for a much bigger class of algebraic expressions<sup>2</sup> and software for calculating with them is called a Computer Algebra System. The oldest one with a definite start date is Maxima (originally called Macsyma), started in 1967, more than 50 years ago, and it is still available today, as open-source software. So, although it is not a new concept, nobody has seen symbolic expressions as the next step to standardize and incorporate into general-purpose programming languages. In fact, you may be surprised to learn that recent work on computer arithmetic is going in the other direction: standardizing lower-precision fractional numbers for use in machine learning.

As far as basic operations, they all work the same as with **Ints**, except division is the more expected `/`, and there is no division with remainder:

```
10 / 4 -> 2.5
11 / 4 -> 2.75
12 / 4 -> 3
13 / 4 -> 3.25
14 / 4 -> 3.5
-1 / 4 -> -0.25
-5 / 4 -> -1.25
```

An interesting While power (exponent) works equally well with **Ints** and **FLOATS**, it does not work with

```
base : Float
base = 5

exponent : Int
exponent = 2
```

<sup>2</sup>See [https://en.wikipedia.org/wiki/List\\_of\\_computer\\_algebra\\_systems](https://en.wikipedia.org/wiki/List_of_computer_algebra_systems).

```
base ^ exponent
```

```
-- TYPE MISMATCH -----
I need both sides of (^) to be the exact same type. Both ←
Int or both Float.

70| base ^ exponent
      ^^^^^^
But I see a Float on the left and an Int on the right.

Use round on the left (or toFloat on the right) to make ←
both sides match!
```

We can fix this with a new function

```
base ^ (toFloat exponent)    -- works!
```

Pay attention to order of operations with this function, because only one of these works!

```
halveA number = (toFloat number) / 2
halveB number = toFloat (number // 2)
```

Since 2 is both an `Int` and a `Float`, it is easy to think 2 will be interpreted as the other type, so it is important to pay attention to the version of divide used.

Going the other way, we lose the fractional part. We can do this by returning the closest whole number, `round`, the next higher, `ceiling`, lower, `floor` or closer to 0, `truncate`. The documentation<sup>3</sup> has many examples, but you can construct your own. Being fractional, `Floats` allow for many other types of mathematical functions, including square root (`sqrt`), logarithm (`logBase`), trig functions `sin`, `cos`, etc.

These functions are very useful for creating animations. If you have already studied trig in a math class, you are probably familiar with most of the functions, but not aware of `atan2` which takes a coordinate  $(x, y)$  and returns the angle between the line  $(0, 0) \rightarrow (x, y)$  and the  $x$ -axis. This is a lot more useful than the inverse tangent you are used to, because `atan2` works for straight up and down, and it can tell the difference between left and right, and therefore produces twice the range of angles as `atan`. One trick is that the inputs are reversed from the normal coordinates, so you use `atan2 y x`. If you need both the angle, and the distance from  $(0, 0)$ , this is called polar coordinates, and Elm provides functions for this:

```
toPolar (3, 4) -> (5, 0.9272952180016122)
(1, 1) <- fromPolar (sqrt 2, degrees 45)
```

We have already used the `degrees` function to convert degrees into radians. One radian is the angle around a circle you get if you take a string stretching from the centre to the edge of a circle (i.e., the radius) and wrap it around the circle as tight as you can. Elm also provides a

<sup>3</sup>See <https://package.elm-lang.org/packages/elm/core/latest/Basics>.

non-standard function `turns`, which converts a number of full turns into radians, which is the same as multiplying by  $2\pi$ . Note that it is up to you to know if your numbers are already in radians. For example

```
1 |> turns |> turns      -> 4*pi^2
1 |> turns |> turns |> turns -> 8*pi^3
```

This is a choice in the design of Elm. We could have created a type for degrees

```
type Degrees = Deg Float
```

and made a function to convert

```
myDegrees : Degrees -> Float
myDegrees (Deg degs) -> pi / 180 * degs
```

in which case

```
anything |> myDegrees |> myDegrees
```

would produce an error, because the input and output of `myDegrees` are not the same. Elm has an optional package `elm-units`<sup>4</sup> which has lots of types and functions like this. It definitely makes software safer if you are using physical qualities.

Another nifty fact about `FLOATS` is that they have special values for infinity, and also for invalid results. This is different from integer calculations which can crash your program if you try to divide by 0. Floating point operations have many ways of failing, and often it is easier not to check each possibility, but rather to do the full calculation and check the final result. To check for this type of failure, we use `isNaN`:

```
isNaN (0/0)      -> True
isNaN (sqrt -1) -> True
isNaN (1/0)      -> False -- infinity is a number
isNaN 1           -> False
```

and to check for infinite numbers we have

```
isInfinite (0/0)      -> False
isInfinite (sqrt -1) -> False
isInfinite (1/0)      -> True
isInfinite 1           -> False
```

We can even compare infinities, and the sign of 0 matters for calculating infinities, but not for comparing 0 against  $-0$ !

```
-1/0 == 1/(-0) -> True
0 == -0        -> True
0/0 == 0/0      -> False
0/0 == 7        -> False
```

---

<sup>4</sup>To keep angles, lengths, volumes, and any other physical units straight in your code and prevent things like the crash of the Mars rover, use <https://package.elm-lang.org/packages/ianmackenzie/elm-units/latest/>. Even if its not rocket science, do you want to make mistakes the compiler could have saved you from? Ideally, physical units would be built into all languages, so that you wouldn't need to use special functions for arithmetic, like `Quantity.times`.

and the error code, “not a number”, `NaN` is not equal to anything, even itself!

There are two really important numbers to mathematicians, and they are both defined in Elm

```
pi -> 3.141592653589793
e -> 2.718281828459045
```

And finally, Elm is a competitive language! We have

```
1 < 2 -> True
1 <= 2 -> True
1 >= 2 -> False
1 > 2 -> False
```

The type signature for these functions

```
(<) : comparable -> comparable -> Bool
```

involves not a type, but a type class `comparable`, just like `number`, which you probably accepted as either an `Int` or a `Float` without thinking about it. Comparables are a bit more complicated. The Elm documentation says that numbers, strings, booleans, tuples, and lists of comparables are comparable. Functions are definitely *not* comparable, because so many of the functions we use have infinite, or at least many billions of inputs. So testing for equality or inequality would be really expensive. The Elm compiler should warn you if you try to compare things for inequality which are not comparable, but it will not always warn you when testing for equality, because the signatures are given as

```
(==) : a -> a -> Bool
(/=) : a -> a -> Bool
```

Meaning they work for any type, as long as both inputs (i.e., sides) are of the same type. Currently, Elm will crash, which is arguably better than getting into an infinite or verrrry long loop. Given that Elm does more than any language we know to detect problems before you try to run a program, this is a bit disappointing.

## 6.1.2 Tuple

[to contents](#)

We have already made extensive use of coordinate pairs, but sometimes we do not want to deconstruct the tuple like this:

```
isOnRight point =
    let
        (x,y) = point
    in
        x > 0
```

and we'd rather use `Tuple.first` or `Tuple.second`:

```
isOnRight point = Tuple.first point > 0
```

Not only do we save typing, but we don't need to define a variable for the `y` coordinate which we won't use.

## 6.2 Strings

[to contents](#)

### 6.2.1 String

[to contents](#)

**Strings** are for text data. Since your program is also text, we use double quotes ("") to separate them. A common mistake when using **Strings** is to miss a closing quote, but if your editor has syntax highlighting, you should notice it right away. Even the listing environment figures it out:

```
halfString = "Everything started off well, ...
anotherDefintion = 7
```

If you have a lot to say, then you will probably want to use triple double quotes:

```
longerStory = """It was a dark and stormy night,
and Cindy was worried about being home alone, when
suddenly---bzzzzt---the power went out, and she
started to wish she had gone to her brothers boring
piano recital after all, but then again, she was..."""
```

Contrary to popular belief, you are allowed to divide your text into multiple sentences inside triple quotes. :) If you are going to do anything with the string, be aware that at the end of each of your lines, Elm will introduce a 'n' (newline) character.

As we will see with Other Containers, Elm's core packages are careful to use the same function names for similar functionality across packages. Here are three:

```
String.isEmpty ""          -> True
String.isEmpty "not empty" -> False
String.length "long"      -> 4
String.length "short"    -> 5
String.reverse "stressed" -> "desserts"
```

And we can convert `toList` and `fromList`, but we need to learn about `Chars` to be able to do that.

But **Strings** are not just any containers, they contain the text from the most thrilling screenplay to the most insipid text message ever sent, as well as every invoice and startup business plan. There is so much we do with **Strings** that we need powerful functions to do them.

```
String.repeat 3 "ho! " -> "ho! ho! ho!"
String.replace "broken" "fixed" "The bicycle is broken!"
           -> "The bicycle is fixed!"
```

We have a special operator for joining strings (++) , but this is just a nicer way of writing the underlying function `String.append`. Although it is less easy to read, it is useful to have when we are using `>>` and the higher-order functions we will learn about when we look at the other containers.

```
"beginning" ++ "middle" ++ "end" -> "beginningmiddleend"
```

```
String.append (String.append "beginning" "middle") "end"
  -> "beginningmiddleend"
String.append "beginning" (String.append "middle" "end")
  -> "beginningmiddleend"
```

Sometimes we have a whole list of strings to put together, and `String.concat` makes that easy.

```
String.concat ["beginning", "middle", "end"] -> "beginningmiddleend"
```

Lots of data comes in parts joined by special characters. We can easily split them apart, and put them back together:

```
String.split "/" "/var/www/html/" -> [ "", "var", "www", "html", "" ]
String.join "/" [ "", "var", "www", "html", "" ] -> "/var/www/html/"
"comma,separated,list" |> String.split "," |> String.join ":" 
  -> "comma:separated:list"
```

Exercise: If you were stranded on a dessert island with `String.join` but not `String.concat`<sup>5</sup>, could you make do? For convenience, we have some special versions of `String.split` designed to help us handling text (at least in left-to-right languages):

```
String.words "To be or not to be" -> [ "To", "be", "or", "not", "to", "be" ]
String.lines """
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;
"""
  -> [ "I shall be telling this with a sigh",
      , "Somewhere ages and ages hence:",
      , "Two roads diverged in a wood, and -I",
      , "I took the one less traveled by,"
      , "And that has made all the difference."
    ]
```

There are some more functions which know about upper- and lower-case letters and word spacing: You can be obnoxious and convert your text to shouting:

```
String.toUpperCase "I'm not shouting!" -> "I'M NOT SHOUTING!"
```

Or e e cummings-ify your writing:

```
String.toLowerCase "I'm not shouting!" -> "i'm not shouting!"
```

You can remove extra whitespace (spaces, tabs and newline characters):

```
String.trim " I'm not \n shouting! " -> "i'm not shouting!"
```

Put spaces around something to line it up in a list:

```
String.pad 10 '~' "piglet" -> "~~piglet~~"
```

---

<sup>5</sup>Since you ask, `String.concat` was heavier and sunk with the ship!

If you are not concerned about niceties of language like word boundaries, you can also `slice` `Strings` starting at any position and continuing as long as you like:

```
String.slice 2 5 "01234567" -> "234"
```

and if it is easier to count from the right, you can use negative numbers:

```
String.slice -5 -2 "01234567" -> "234"
```

This is a good example of insufficient test cases. If you think of test cases as being like fences, with each fence preventing your pets from running off in a certain direction, then these two examples are like putting up a single fence pole and wondering why your goats are so excited, only to come back later and find out they ate all of your marigolds. What is the simplest function which could satisfy those examples? How about

```
simpleSlice x y str = "234"
```

as a simple definition, which we know is totally undeserving of the name `slice`, but if you are an automated testing environment, or if you are an automatic code generator trying to generate efficient code from a specification, then you have to admit it is a pretty good definition. I hope you will think about this example when you try to write test cases—or examples for your own book—in the future!

A few more examples can help fence it in, and save your marigolds:

```
String.slice 0 -3 "01234567" -> "01234"
String.slice 0 5 "01234567" -> "01234"
String.slice 1 -4 "01234567" -> "123"
String.slice 2 20 "01234567" -> "234567"
```

These are not enough to prevent an obviously wrong function from matching all these tests, but they are probably enough for the “simplest” function to be the one we want, assuming we can agree on what simple<sup>6</sup> means. `String.slice` is pretty powerful, but you do have to think about what different combinations of parameters do. In many cases, we can get by with four special-case functions, which are more readable, because they are less powerful:

```
String.left      2 "01234567" -> "01"
String.right     2 "01234567" ->       "67"
String.dropLeft  2 "01234567" ->    "234567"
String.dropRight 2 "01234567" ->   "012345"
```

Exercise: To prove that the above functions are all special cases of `String.slice`, implement them using `String.slice`.

Sometimes, we don’t need to `split` or `join` our `Strings`, we just need to know what’s in them. For example, we need to know if a file

```
filename = "topSecret.pdf"
```

is a pdf:

```
String.endsWith ".pdf"      filename -> True
String.endsWith "topSecret" filename -> False
```

---

<sup>6</sup>Most Computer Scientists would probably use Kolmogorov Complexity, see [https://en.wikipedia.org/wiki/Kolmogorov\\_complexity](https://en.wikipedia.org/wiki/Kolmogorov_complexity).

or is `topSecret`,

```
String.startsWith ".pdf"      filename -> False
String.startsWith "topSecret" filename -> True
```

or contains `topSecret`.

```
String.contains ".pdf"      filename -> True
String.contains "topSecret" filename -> True
```

Of course, what we probably need to know is whether the file contains top-secret information or not, and we might even need to look for encrypted information! There are highly optimized libraries for this like Lucene<sup>7</sup>. The theory and algorithms for this are interesting, and whether it is finding approximate matches in DNA sequences or potential plagiarism in essay submissions, it is an active area of research and development in computer science. One part of it is indexing, and we will bring together several of the standard packages to create an indexing example which would be good enough for many smaller applications, i.e., like a quick-search for this textbook! – TODO where is it?

We have already seen that we can easily `split`, process and `join` strings together, but these operations can really add up if we are processing lots of strings. It might be more efficient to keep track of where we find substrings so that we can eliminate a lot of false matches before we actually start processing our `Strings`. For this, Elm gives us

```
String.indices "/" "/home/donald/documents/top_secret/stolen.doc" ←
    -> [0,5,12,22,33]
```

and since some people spell it `indexes`, the `String` library gives you that as a synonym.

*Conversions* to and from `Strings` are needed every time you want to display something for the user, or process their typed input or a document. Turning a number into a `String` is easy:

```
String.toInt    123    -> "123"
String.toInt   -123    -> "-123"
String.toFloat 123    -> "123"
String.toFloat  3.9    -> "3.9"
```

But going the other way is complicated. What number should "`cat`" give you? Maybe 9 for its nine lives? Instead, Elm gives you `Nothing`!

```
String.toInt  "cat"    -> Nothing
String.toFloat "cat"    -> Nothing
```

However, if you can just manage to type a valid number, then you will just get a number:

```
String.toInt "-42"     -> Just -42
String.toFloat "3.1415" -> Just 3.1415
```

To understand the output, you will need to read ahead to Section 6.4.3.

---

<sup>7</sup>For some history see [https://en.wikipedia.org/wiki/Apache\\_Lucene](https://en.wikipedia.org/wiki/Apache_Lucene).

## 6.2.2 Char

[to contents](#)

Char defines the character type for Elm, which fortunately for us supports Unicode. Unicode is designed to support all human languages, and it includes all of the current and extinct languages we know<sup>8</sup>. Single quotes (') distinguish **Chars** from **Strings** (of characters), but you can convert back and forth using

```
String.toList    "wild"          -> ['w','i','l','d']
String.fromList ['l','i','o','n'] -> "lion"
```

and if you only have one character, it is slightly more efficient to use

```
String.fromChar '1'           -> "1"
```

These functions are mildly interesting, but we will have to come back to **Char** to see what we can do with these and the rest of the library after we have learned about **List**.

## 6.2.3 Higher-Order String functions

We have already mentioned higher-order functions, which are functions with another function as an input, but having discovered **Strings**, we can really make use of them. Let's first look at their type signatures:

```
map    : (Char -> Char) -> String -> String
filter : (Char -> Bool) -> String -> String
any    : (Char -> Bool) -> String -> Bool
all    : (Char -> Bool) -> String -> Bool
```

What they all have in common is that their first arguments are in parentheses, (). For example **(Char -> Char)** means that, instead of having two **Char** inputs, **String.map** has a single function as an input, and that function has one **Char** input and a **Char** output.

Knowing the type signature pretty much gives away what **String.map** and **String.filter** do. Think about it for a minute, what can you do with a function which turns a **Char** into another **Char** and a **String**, which is a bunch of **Chars** stuck together?

Still thinking? I don't see smoke coming out of your ears, so think a bit harder. At this point, you are probably thinking, a **String** is a bunch of **Chars** and we have a function to transform **Chars** so we should probably transform the **Chars** one at a time, and reassemble them as a **String**.

```
map (\c -> '^') "a/b/c"          -> "^^^^^"
map (\c -> if c == '\' then '/' else c)
"\home\cindy\documents\fluffy.doc" -> "/home/cindy/documents/←
fluffy.doc"
```

And we could use **String.map** to reimplement some functions we already have:

```
myStringToUpper = String.map Char.toUpperCase
myStringToLower = String.map Char.toLowerCase
```

---

<sup>8</sup>The lack of future languages either indicates a dead end in our civilization coming up, or a lack of time travel.

Here we see the advantage of partial evaluation, since we do not have to come up with a name for the argument:

```
myStringToUpper picolo = String.map Char.toUpperCase picolo
```

When using standard functions like `map`, whose type signatures are known, there shouldn't be any confusion when using partial application to define a new function, but for new or seldom-used functions, it can be helpful to write out the argument, because that makes it obvious what is a function, since it has an input.

Similarly, what could `String.filter` do with the function `Char -> Bool` it takes as an input? Here are some hints:

<code>String.filter Char.isDigit</code>	<code>"H0H 0H0" -&gt; "000"</code>
<code>String.filter Char.isAlpha</code>	<code>"H0H 0H0" -&gt; "HHH"</code>
<code>String.filter ( \ c -&gt; Char.isAlpha c    Char.isDigit c )</code>	<code>"H0H 0H0" -&gt; "H0H0H0"</code>
<code>String.filter Char.isAlphaNum</code>	<code>"H0H 0H0" -&gt; "H0H0H0"</code>

Again, these are not enough examples to completely specify the function, but you should be able to add enough examples.

`String.any` and `String.all` also have an input `Char -> Bool`, function but instead of returning a `String`, they return a `Bool`. Try to guess what they do. Exercise: write down your definitions in English. Do your definitions match these examples?

<code>String.any isDigit "H0H 0H0" -&gt; True</code>
<code>String.any isDigit "1234576" -&gt; True</code>
<code>String.any isDigit "Santa" -&gt; False</code>
<code>String.all isDigit "H0H 0H0" -&gt; False</code>
<code>String.all isDigit "1234576" -&gt; True</code>
<code>String.all isDigit "Santa" -&gt; False</code>

They probably do, because the function calls can (almost) be completed into English sentences: Are `any` of the characters `isDigits` in "`H0H 0H0`"? Are `all` of the characters `isDigits` in "`H0H 0H0`"?

You are probably feeling pretty good now! You have learned to guess what functions do, just based on their types. This is a taste of what is possible when thoughtful designers use strong typing. The next two functions are a bit more complicated. Let's look at their type signatures.

<code>foldl : (Char -&gt; b -&gt; b) -&gt; b -&gt; String -&gt; b</code>
<code>foldr : (Char -&gt; b -&gt; b) -&gt; b -&gt; String -&gt; b</code>

This time, in addition to the `String` and `Char` types, there is a type variable `b`. Just like a normal variable, a type variable can be anything, but in this case, it is a type. This means that we can use one function with different types of inputs and outputs. We call this polymorphism, which is Greek for “many shaped”. Let's look at some simple examples, where `b=Int`.

<code>foldl : (Char -&gt; Int -&gt; Int) -&gt; Int -&gt; String -&gt; Int</code>
--

The simplest example is

```
String.foldl (\ c i -> 7) 0 "any string" -> 7
String.foldl (\ c i -> 7) 0 "" -> 0
```

The first function ignores the characters in the string and always returns 7, as does the invocation of `foldl`, unless the `String` is empty. Why?

On the other hand,

```
String.foldl (\ c i -> if c == '1' then 1 else 0) 2 "11b" -> 0
String.foldl (\ c i -> if c == '1' then 1 else 0) 2 "111" -> 1
String.foldl (\ c i -> if c == '1' then 1 else 0) 2 "b11" -> 0
String.foldl (\ c i -> if c == '1' then 1 else 0) 2 "" -> 2
```

ignores the `i:Int` input, and always returns 1 if the last character is a '`1`'. Why does only the last character matter? This is where the “l” in `String.foldl` comes in. “L” stands for *left to right*, and we can write out the application as follows:

```
String.foldl oneOrZero 2 "11b"
-> String.foldl oneOrZero (if '1'=='1' then 1 else 0) "1b"
-> String.foldl oneOrZero 1 "1b"
-> String.foldl oneOrZero (if '1'=='1' then 1 else 0) "b"
-> String.foldl oneOrZero 1 "b"
-> String.foldl oneOrZero (if 'b'=='1' then 1 else 0) ""
-> String.foldl oneOrZero 0 ""
-> 0
```

where

```
oneOrZero = \ c i -> if c == '1' then 1 else 0
```

In this example we have processed the `String` one `Char` at a time, starting from the left. That explains the “l” what about the “fold”? This could be in analogy with folding nuts into batter while making a nut cake.

Exercise: Write this in Elm.

Did you get

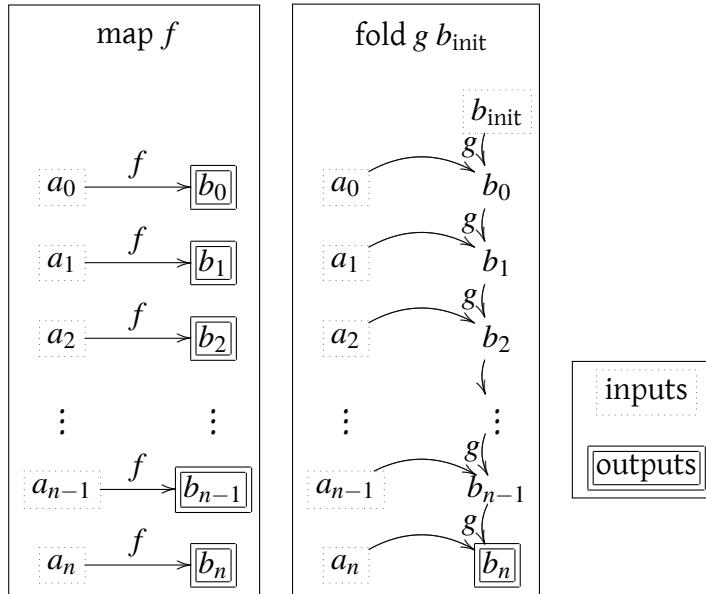
```
String.foldl spoon batter nuts
```

or

```
String.foldl batter nuts spoon
```

In case you are not a baker, think about the last time you prepared for a long trip. You put on your felt hat. You lay out your blanket, put some nuts near the end, fold it over, add some more nuts, folded again. Not how you prepare for long trips? You might have trouble getting the hang of `foldl`, and need to use recursion instead!

Or maybe this picture would help.



## 6.2.4 Left versus Right

To easily see the difference between `foldl` and `foldr`, consider these examples:

```
String.foldl ( \ c soFar -> "("++String.fromChar c++soFar++)" ) <-
  "" "abc"
  -> "(c(b(a)))"
String.foldl ( \ c soFar -> "("++soFar++String.fromChar c++)" ) <-
  "" "abc"
  -> "(((a)b)c)"
String.foldr ( \ c soFar -> "("++String.fromChar c++soFar++)" ) <-
  "" "abc"
  -> "(a(b(c)))"
String.foldr ( \ c soFar -> "("++soFar++String.fromChar c++)" ) <-
  "" "abc"
  -> "(((c)b)a)"
```

Ok, you may have to think about it a bit, stand on your head while looking in a mirror, then ask yourself: What determines the inner-to-outer order with respect to parentheses? What determines the left-right order in the output?

## 6.3 Polymorphism and Standard Interfaces

[to contents](#)

We just saw that some functions like `String.foldl` are really a whole family of functions because their type signatures contain type variables. This is called *polymorphism*, but it is the most restricted type of polymorphism. Let's look at how Elm supports it, and why we like polymorphism. Ok, we'll admit that we like polymorphism because we're lazy. We don't want to remember two function names if we can get away with one, and polymorphism lets us do this. The different types of polymorphism are

- parametric The signature contains type variables, and the different versions of the function are *parametrized* by the types assigned to those type variables. This polymorphism is *structural* because it only depends on the structure of how types are used. Any functions which are needed are supplied as arguments, i.e., for `foldl` this was the combining function. Knowing that however, we implemented a parametricly polymorphic function has to work for every type, and therefore cannot rely on any of the operations or properties of that type, allows us to deduce properties which are called *free theorems*.<sup>9</sup>
- ad hoc In this case we use the same symbol or function name for different functions paternalistically. For example, many languages allow `+` to both add numbers together and join strings, even though these are really different functions. Notably  $1 + 2 = 2 + 1$  but "`cat`"++"`dog`" ≠ "`dog`"++"`cat`"!<sup>10</sup> Computer scientists who understand math don't like ad hoc polymorphism for this reason. It creates a complicated type system and then hides it from the user, which makes it easier to get the right answer without understanding why, but makes it harder to know your answer is right.
- subtype If you think of all possible values as a set, then types are subsets. Elm's basic types are disjoint subsets, but record types can intersect each other (but don't have to). Let's look at an example. If your `model` contains

```
{ time : Float, positions : List (Float,Float) }
```

And you have functions:

```
clock model = ... model.time ...
ships model = ... model.positions ...
```

Then what are the types for `clock` and `ships`? One uses `.time` and the other `.positions`.

```
clock : { a | time : Float } -> Shape Msg
ships : { a | positions : List (Float,Float) }
                  -> Shape Msg
```

You should read these types as

“any record with a `Float` field labelled by `time`,”

or

“any record with a field `positions` of type `List (Float,Float)`.”

Both of these are subsets of

```
{ a | time : Float
      , positions : List (Float,Float) }
```

But note that neither are subtypes of

---

<sup>9</sup>Originally freed by Wadler [Wad89], but you can find the basic idea liberated from the discrete mathematical notation on Reddit [https://www.reddit.com/r/haskellquestions/comments/6fkufo/free\\_theorems/](https://www.reddit.com/r/haskellquestions/comments/6fkufo/free_theorems/).

<sup>10</sup>This property is called commutativity, and string concatenation does not have it!

```
{ time : Float, positions : List (Float,Float) }
```

because that type doesn't include records with a three or more fields.

This explains subsets and intersecting sets. How do we get disjoint sets? Think about it for a while<sup>11</sup>. Now does it make sense that records in Elm use set braces, {}?

**duck typing** In languages without types or with weak enforcement, *duck*<sup>12</sup> *typing* leaves it up to the programmer to test the capabilities of each object. The problem with duck typing is that, too often, the way you find out that your data does not have the type you thought it did is when your program crashes or produces an incorrect result.

## 6.4 Containers

[to contents](#)

Containers are things for carrying and storing other things. In real life, we have boxes, bags, baskets, envelopes, and so on. Each one has advantages and disadvantages. Elm has these too, and just like we have English words which apply to many or all containers (big, small, put in, take out, ...), we have similar Elm functions. If you just read the last section, you are thinking, this sounds like polymorphism, and it is.

In many languages, there are built-in ways to capture this similarity via ad hoc polymorphism. In Haskell, type classes define sets of functions which types of a certain class must have. In object-oriented languages, classes can inherit properties (including functions) from super classes, and some languages allow virtual classes (which cannot be used directly, but only via subclassing). Many languages also support generic programming<sup>13</sup> which goes beyond defining common functions for a class, to creating common algorithms which can then be specialized for each instance, leading to more efficient code.

In Elm, we don't have these things. What we have are a compact set of standard libraries which are pretty careful to use the same name for functions which do the same thing. This makes it easier to learn to use multiple containers, and perhaps even easier than in other languages, but it does not allow us to write functions once and use them with different containers, which all of the above approaches would allow. Given the current uses of Elm as a web programming language, this is not a very big barrier. Just like a kitchen gadget, if you buy something with too many bells and whistles, you may never figure out how to use it for basic cooking, or it may do something unexpected, spoiling your recipe. Elm is for basic cooking.

We start with the container `List`, both because we can do a lot with `Lists` and because it is an easy container to visualize.

---

<sup>11</sup>How about `{a:Int}` and `{a:Float}`?

<sup>12</sup>This is based on the saying "If it walks like a duck and it quacks like a duck, then it must be a duck." See [https://en.wikipedia.org/wiki/Duck\\_typing](https://en.wikipedia.org/wiki/Duck_typing) for a complete definition.

<sup>13</sup>genericity: [https://en.wikipedia.org/wiki/Generic\\_programming](https://en.wikipedia.org/wiki/Generic_programming)

## 6.4.1 List

[to contents](#)

A container type must be a parametric type, sometimes called a constructor type. In this case, if `element` is a type, then `List element` is also a type. It is the type of lists of elements. For example

```
[1,2,3]           : List Int
["cat","dog"]    : List String
[[1],[1,2],[1,2,3]] : List (List Int)
```

Yes, you can have a list of lists. Who would ever need that? You will be surprised how often it is useful, and even necessary, to use nested lists. Some lists are much easier to build as a list of lists, and then collapse together using

```
List.concat : List (List a) -> List a
```

We use this a lot when generating code, like in SD Draw, because some things take up one line, and some more than one line. Tip: This is also an easy way of combining the output of functions, some of which generate no elements. Just return `[]` an empty list in that case, and `List.concat` effectively removes the empty lists when combining.

But beware of type errors when nesting multiple levels deep. It is easy to generate an error like this where different elements of a list have different types. The underlying element type is the same, but the level of nesting is different. It is easy to see here, but it can be puzzling when all of the elements of the outermost list are returned by different functions.

```
-- TYPE MISMATCH ----- ID/Animation1.←
elm
```

The 3rd element of this list does not match all the ←  
previous elements:

```
4| [ [1]
5| , [1,2]
6| > , 3
7| ]
```

The 3rd element is a number of type:

number

But all the previous elements in the list are:

List number

Hint: Everything in a list must be the same type of value. ←  
This way, we never  
run into unexpected values partway through a \elm{List.map} ←  
, \elm{List.foldl}, etc. Read

to learn how to ``mix'' types.

### 6.4.2 List Queries

We can find out what is in the list with

```
member : a -> List a -> Bool
```

We have seen `List.concat` and `List.member` operating on any type of list. For joining lists together, this makes sense, we don't really need to know what is in two boxes to dump one into the other. But for membership, actually we do. How will we know if something is in a list, if the element type is not comparable? Fortunately, most types you will construct are comparable, but one isn't. If you try this

```
listOfFunctions =
  [ \ x -> 2*x
  , \ x -> 3*x
  ]

badTest =
  List.member sqrt listOfFunctions
```

you won't get a compiler error, but you will get a run-time error

Error: Trying to use `(``==``)` on functions.  
 There is no way to know if functions are "the same" in the ←  
 Elm sense.  
 Read more about this at <https://package.elm-lang.org/>←  
 packages/elm/core/latest/Basics#== which describes why ←  
 it is this way and what the better version will look ←  
 like.

Unfortunately, you won't even see this error unless you open the JavaScript console in your browser window. Now we can easily see that `sqrt` is not in the `listOfFunctions`, but, in general, it is a hard problem. In fact, it is *undecidable*, which we can take to mean there is no method of deciding whether they are the same in a finite number of steps, for any given pair of functions.

This is not great! It would be better if Elm didn't let us get into this situation, but to avoid it, it would need to restrict some of the list functions to `comparable elements`. We could do this ourselves if Elm had type classes. But in practice, Elm programs just don't crash, so Elm programmers don't seem to fall into the trap of comparing functions. If you learn by trying to break things—which can be a great learning strategy—you can definitely do so here.

Writing `List.member` in another way:

```
isAMember thing list = List.any ( \ elem -> thing == elem ) list
```

We can see exactly why `List.member` requires comparable elements. You may have noticed that we haven't learned `List.any` yet. This is a good example of effective ad hoc polymorphism—our knowledge of `String.any` is transferable.

Let's look at some more examples

```
List.any isBig [ 8, 9]      -> False
List.any isBig [ 9,10]     -> True
List.any isBig [12,13]     -> True
```

Can you define `isBig` so that it works for all examples?

```
isBig x = x > 10
```

If one is not enough, we also have `List.all`:

```
List.all isBig [ 2, 3]      -> False
List.all isBig [ 2,13]     -> False
List.all isBig [12,13]     -> True
```

Sometimes we need to check if a list is empty:

```
List.isEmpty []            -> True
List.isEmpty [7]           -> False
List.isEmpty [[]]          -> False
```

That last one was tricky! A `List` containing an empty `List` is not empty.

Of course, we have been using `Lists` since day 1, when we put our `Shapes` between `[]`s. There are a few more ways to create them:

```
List.singleton "Hello List"    -> ["Hello List"]
```

which isn't really needed, because we could type `["Hello List"]` directly, but may be helpful in defining other functions.

If nobody says Hello back, you can try repeating yourself:

```
List.repeat 3 "Hello List"    -> ["Hello List","Hello List","Hello List"]
```

I use this a lot together with other `List` functions.

Another thing we need a lot of is creating a sequence of numbers in order:

```
List.range 1 10             -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List.range 10 1              -> []
List.range 99 99             -> [99]
```

Note that the numbers have to be in order, and they have to be `Ints`, not `FLOATS`, and definitely not `Strings` or `Chars`. If you need the alphabet, you need to use `Char.fromCode`

```
List.range 1 25 |> List.map Char.fromCode
                  -> ['A','B','C','D','E','F','G','H'←
                      ','I','J','K','L','M','N','O',''←
                      P','Q','R','S','T','U','V','W←
                      ','X','Y','Z']
```

Yes, I know, we haven't learned about `List.map` yet, but `List.map` works very much like `String.map`.

Let's jump to more advanced mapping. `List.indexedMap` maps a function over two `Lists`, but you only have to specify the second one! the first one is the index, or position of the second input in the `List` starting with 0. If you want to put things in grid



you can use the index to change the  $x$  coordinate

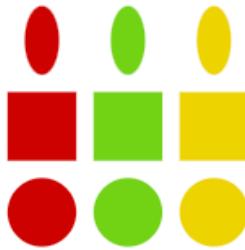
```
List.indexedMap
  ( \ idx shape -> shape |> move (10 * toFloat idx,0) )
  [ circle 4 |> filled red
  , square 8 |> filled green
  , oval 4 8 |> filled yellow
  ]
```

Which can be really useful if you mix it with text

```
List.indexedMap
  ( \ idx char -> char |> String.fromChar
    |> text
    |> fixedwidth
    |> filled purple
    |> move (0,20)
    |> rotate (degrees -22 * toFloat idx) )
  (String.toList "Hello Circle!")
```

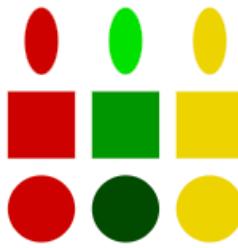
And if you need a 2d grid, you can nest `List.indexedMaps`:

```
List.indexedMap
  ( \ yIdx stencil ->
    ( List.indexedMap
      ( \ xIdx clr ->
        stencil |> filled clr
        |> move (10 * toFloat xIdx,10 * toFloat ←
                  yIdx)
      )
      [ red, green, yellow]
    )
  )
  [ circle 4, square 8, oval 4 8 ]
|> List.concat
```



Notice that the variable `yIdx` has a larger scope. We could use it to change the colours, for example by

```
[ red, rgb 0 (75 + 75 *toFloat yIdx) 0, yellow]
```



If you have programmed with a language containing loops, you might be thinking that this structure resembles a loop. A couple of things:

1. the loop structure looks simpler;
2. the {}s in other languages turn into ()s;
3. unlike the loop, we can easily move the “body” outside the loop by using a named function, rather than the lambda function; and
4. unlike the loop, the map construction guarantees that the values in the list can be computed in any order, which is really helpful if you have a multi-core processor—something even today’s least-expensive smartphone has—and want the compiler to automatically parallelize your code to take advantage of those multiple cores.

Why doesn’t Elm have a simpler way of expressing this? It is a trade off. Having simpler ways of using features in special cases can save typing, and make some code more transparent, but it always means more for the beginner to learn.

There are some other maps, `List.map`, `List.map2`, ... , `List.map5`. We have already seen simple maps. The numbered maps differ in taking functions with more than one input, up to five, to be specific. They are good for combining multiple lists. It’s hard to come up with an example with five lists. How about this?

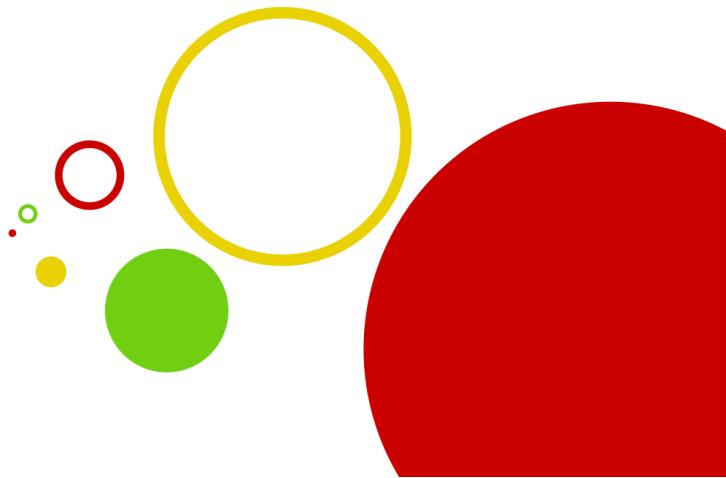
```
List.map5
( \ size clr x y toShape ->
  circle size
  |> toShape clr
  |> move (x,y)
```

```

)
[ 1, 2, 4, 8, 16, 32, 64]
[ red, green, yellow, red, green, yellow, red, green]
[ -90, -86, -80, -70, -50, -20, 65, 80]
[ 0, 5, -10, 15, -20, 25, -30, 35]
[ filled, outlined (solid 1)
, filled, outlined (solid 2)
, filled, outlined (solid 3)
, filled, outlined (solid 4)
]

```

Can you see that this gives you



Probably not! It would definitely have been easier to write this without `List.map5`, but after puzzling over it, you should have a very good idea what the numbered maps can do. When are you going to need this? Putting two lists together happens all the time. Sticking with graphics, we often need to make a “natural” scene, like a forest or stars in the night sky. If we already have a list of tree `Shapes`, we can combine it together with a random list of points. If we are talking about trees which grow at different rates, then we could use `List.map3` to add in a third list of random numbers to add a random scaling.

Hopefully, everyone loves mapping. The more we use it, the more likely compiler writers will figure out how to use the inherent parallelism to generate code to take advantage of the tens of thousands of threads of execution possible on the Graphics Processing Unit on even an old GTX card or recent iPhone<sup>14</sup>. But what about when your computations are not independent of each other?

What if we want to visualize a list of areas: 10, 37, 28, 12, 2, 19, 19, 14, 4, 28. We could of course draw a bar chart, but a bar chart is really a way of showing (linear) numbers. It might

<sup>14</sup>Yes, even an *old* GPU needs to run tens of thousands of threads in parallel to use its power: <https://stackoverflow.com/questions/6490572/cuda-how-many-concurrent-threads-in-total>, and today the same is true of a fanless M1 MacBook, and soon it will be true for smartphones. For a comparison of M1 processors: [https://en.wikipedia.org/wiki/Apple\\_A14](https://en.wikipedia.org/wiki/Apple_A14), which doesn’t take into account that arithmetic units themselves need to run multiple computations in parallel for top efficiency, as explained in the previous stackoverflow post.

be more useful for your readers to see the areas as squares or circles:



We can accomplish this with a function which keeps track of what has gone before. Like a map with `state`. Remember the `update : msg -> model -> model` function? Replace the `msg` type with an `input` type which could be a number—as in the areas example—or any other type. Now replace the `model` with a more general `state`. What we need is a function which processes a whole list of `inputs` to this generalized

```
update : input -> state -> state
```

That function is a fold, and comes in two flavours:

```
List.foldl : (input -> state -> state) -> state -> List input
           -> state
List.foldr : (input -> state -> state) -> state -> List input
           -> state
```

One which processes the inputs starting from the left and one starting from the right. Putting squares shoulder to shoulder can be accomplished by either, if we have an update function, which in this case we'll call `addArea`:

```
List.foldl
  addArea
    ((-90, red), [])
      areas
|>
  Tuple.second
```

This is a typical pattern for using either fold. The `state` type is a pair, with the first part being what we need to remember from previous inputs—the state—and the second part being a list which is *almost* a map of the input, except for the need for memory. We have to start the fold off with an initial state, in this case

```
((-90, red), []) : ( (Float, Color), List (Shape msg) )
```

Often, we don't need the `state` type except for the list, so we compose with `Tuple.second`.

Once we establish that the fold pattern works, we reduce the problem of processing the whole list to processing one input at a time:

```
addArea newArea ((currentPosition, clr), shapesSoFar) =
  let
```

We start with some advanced math to calculate the width of the square with the desired area (ignoring negative inputs):

```

width = if newArea > 0 then
        sqrt newArea
    else
        0
in

```

After this square is shouldered in, the next square will go `width` units to the right:

```
( ( currentPosition + width
```

To make things look nicer, we cycle through three colours, but in a real visualization, that would be missing an opportunity to encode another aspect of the data into the colour, like the per-capita avocado consumption.

```

, if clr == red then
  green
else if clr == green then
  yellow
else
  red
)
```

Finally! we draw the square, and move it into place. Note that moving along by `width` makes sense for the left-hand side of the squares. If were aligned by the middle of the squares, we would need to increment by the current and future squares, which would make the state more complicated. So let's not do that! Since `squares` come centered on their centres, we need to shift them by  $0.5 \times \text{width}$ , both horizontally and vertically. As an exercise, modify `addArea` so the squares are lined up on their right-hand sides. (Surprise! the code is the same, but the explanation is different.)

```

, ( square width
  |> filled clr
  |> move (0.5*width + currentPosition, 0.5*width
)
:: shapesSoFar
)
```

Let's look at another example, compositing a collage, or, as you would have called it in kindergarten, gluing shapes:

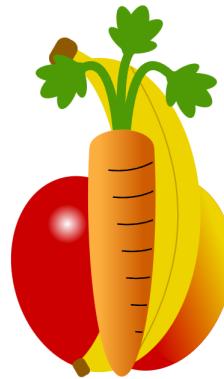
```
List.foldl glue blankPage [apple,banana,carrot]
```

means calling `glue` three times, once for each shape in the list:

```

blankPage
|> glue apple
|> glue banana
|> glue carrot

```



If you don't like the order, you can start from the right:

```
List.foldr glue blankPage [apple,banana,carrot]
```

which gives

```
blankPage
|> glue carrot
|> glue banana
|> glue apple
```



Before we finish with folding, let's think about those 10,000 cores again! Actually, let's think about millions of cores in the server farm of a pirate consortium. They really don't want to wait for a single core to fold their data. And they don't! Because they ensure that their `update` function has special properties.

Let's say they want to add up their ill-gotten cash. They could use

```
List.foldr (+) 0 [ill,gotten,cash]
```

Starting with 0 and adding one amount after another. But

$$(x_{\text{ill}} + x_{\text{gotten}}) + x_{\text{cash}} = x_{\text{ill}} + (x_{\text{gotten}} + x_{\text{cash}}). \quad (6.1)$$

This is called *associativity*, which practically means we can put the brackets anywhere. If we did have a server farm, we could divide up the task, giving two numbers to each core

to add up until we run out. Then taking the new list, and doing the same thing again. And repeating until we are down to a single number. This could use a lot of cores to reduce our work, but it only works for functions which are associative, which, if you didn't notice, also requires that the `input` and `state` types be the same.

You could call this a multi-core fold, but most languages call it *reduce*. We don't have one in Elm. Browsers don't allow us to access multiple cores anyway, but if you are thinking about a career in data science, it might be something to look out for in your next programming language.

Addition also has another property, called *commutativity*, i.e.,

$$1 + 1 = 1 + 1$$

oops, bad test case, I meant

$$1 + 2 = 2 + 1$$

I'm sure you knew what I meant. The order doesn't matter. If you are running on a distributed cluster, where you really don't know which cores will finish first and send their partial sums back, it is handy to know that order doesn't matter, so you can just add up partial sums in whatever order they happen to arrive. It would be good to have a special *reduceCommutative* which can take advantage of this property. And now that you have chopped your computation time down by a factor of 10,000, notice that you can also save one operation, since you don't need to add 0 this way!

Finished with mapping? There is a special type of mapping which makes sense for lists. Sometimes we need to map a function which turns elements into lists over a `List`, but we don't want a `List` of `Lists`! We have a function for that.

`List.concatMap : (a -> List b) -> List a -> List b`

which we could build out of two other functions

`List.concatMap = List.map >> List.concat`

This also works for mapping functions which may return empty `Lists`, like when people stop paying attention to your lecture about `Lists` and you need to collect their cell phones. Some people don't have one, so `cellPhones` returns `[]`, while others have several burner phones—don't ask why.

`List.concatMap cellPhones audience`

Similar to mapping, we also have filtering, which we saw previously for `Strings`. Say we need to remove empty lists from a `List` of `Lists`.

`List.filter (List.isEmpty >> not) [[apple],[],[apple,banana]] -> [[apple],[apple,banana]]`

But some functions, like `String.toInt`, don't return `Lists`, they return `Maybes`, so we have another version of this:

`List.filterMap String.toInt ["1","2","3.14"] -> [1,2]`

Why do we have so many ways of mapping and filtering? Because they are incredibly useful.

But where do all these `Lists` come from? We have already seen that putting `[stuff]` inside square brackets creates a list. This works for any size, as long as we know the elements when we are writing the program. Definitely works for one element, but sometimes it is convenient to have a function for that

```
List.singleton one -> [one]
```

Where are we likely to use this? When we need to mix a function which returns one element with functions which return `Lists`:

```
[ drawStudents 7, drawTeacher >> List.singleton ]
```

Yeah, it is pretty rare!

We can also put `Lists` together

```
[1,2,3] ++ [4,5,6] -> [1,2,3,4,5,6]
```

which if you prefer words to symbols can be accomplished with

```
List.append [1,2,3] [4,5,6] -> [1,2,3,4,5,6]
```

and if you only want to add one element to the beginning of a `List`:

```
1 :: [2,3,4] -> [1,2,3,4]
```

This last function is called “cons”, and has a long history in programming languages, going back to Lisp<sup>15</sup>. It has a special role, because there is only one way of decomposing a `List` into the first element (called the *head*) and the rest (called the *tail*). This is *not* true for `(++)`! So if we cannot do what we need with a combination of mapping, folding, and filtering, and we actually have to figure out a new function, we probably need to break it down by `cases`:

```
addUp listOfNumbers =
  case listOfNumbers of
    oneNumber :: restOfNumbers
      -> oneNumber + (addUp restOfNumbers)
    []
      -> 0
```

(Note that this function already exists, as

```
List.sum [1,10,100] = 111
```

as does

```
List.product [2,3,5] = 30
```

As a fun exercise, why don't you modify `addUp` to make `multiplyAcross?`) Note that `addUp` only needs to add one number! If you have two or more numbers, it calls in a friend (conveniently also called `addUp`) to handle the rest. We will talk more about this in Chapter 7.

If we don't want to use a `case` expression for this, for whatever reason, we also have functions to get the head and tail of a list:

```
List.head [1,2,3] -> Just 1
List.head [] -> Nothing
```

---

<sup>15</sup>where list processing really took off <https://en.wikipedia.org/wiki/Cons>.

and

```
List.tail [1,2,3]    -> Just [2,3]
List.tail [1]        -> Nothing
List.tail []         -> Nothing
```

Note that both return `Maybe` wrappers around the element type. More about that in the Results section, but why can we not just return the element type? Well, we have to do something for empty lists. So just like converting a `String` into an `Int`, we cannot count on `List.head` always working. Many other languages have functions to return the head of a list which will crash on empty lists—including languages which have features like `Maybe`, but inscrutably let you turn them off. Of course, everyone is in a hurry, and you may be doing a quick experiment which you know will succeed, or you don't care if your program crashes. But more often than you would think, those bits of experimental code end up in mission-critical applications. Elm doesn't let you do that. Since we do have `case` expressions which safely deconstruct lists, why do we need these? Well, they are convenient for constructing conditions in `if` expressions like this

```
if model.hasRainbow == True &&
    List.head model.customers == Just "unicorn" then
    ...
```

It does often happen that you want to take the tail and use it in some list processing, and if it happens to be empty, no big deal. Except that you cannot do that because of the `Maybe` wrapper. In that case, you can use `List.drop 1`. Notice the first argument. In general, you can `List.take` or `List.drop` any number of elements.

```
List.take 5 [1,2,3,4,5,6,7,8]    -> [1,2,3,4,5]
List.take 5 [1,2,3,4]            -> [1,2,3,4]
List.drop 5 [1,2,3,4,5,6,7,8]   -> [6,7,8]
List.drop 5 [1,2,3,4]            -> []
```

This can be handy if you are running a contest, and want to take the top three:

```
scores
|> List.take 3
```

Those are the first three in the list. To avoid a lot of people shouting at you, you probably want to `List.sort` the list first. :)

```
scores
|> List.sort
|> List.take 3
```

This works as long as your list elements are `comparable`, as we discussed above.

```
List.sort : List comparable -> List comparable
```

To keep the code simple, it is good to think ahead to how things should be compared.

```
scores = [("Abdul",7),("Betsy",6),("Crystal",8)]
```

will probably not give you the result you want, because it will sort by the name, which comes first in the pairs, whereas

```
scores = [(7,"Abdul"),(6,"Betsy"),(8,"Crystal")]
```

will give you what you want, since the judges' score comes first.

In complicated cases, you can specify your own ordering mechanism with

```
sortWith : (a -> a -> Order) -> List a -> List a
```

which takes an additional input—the comparator—which returns a value of

```
type Order
  = LT
  | EQ
  | GT
```

This could be useful if you have a weighted scoring system like

```
scores
|> List.sortWith
  (\ (judge1A,judge2A,nameA) (judge1B,judge2B,nameB) ->
    if judge1A + 2*judge2A < judge1B + 2*judge2B then
      GT
    else if judge1A + 2*judge2A > judge1B + 2*judge2B then
      LT
    else
      EQ
  )
|> List.take 3
```

But if you just want to sort on one value, even if it is in the wrong place in your pair or is part of a record you can use

```
sortBy : (a -> comparable) -> List a -> List a
```

For example, to sort according to the `.score` field in a record, of type

```
scores : { name : String
          , score : Float
          , country : String
          , grade : Int
        }
```

you can use

```
scores
|> List.sortBy .score
|> List.take 3
```

The same thing would work with `Tuple.second`. And if you end up with the result in the opposite order, we have the handy

```
List.reverse [1,2,3,4] == [4,3,2,1]
```

which would be useful if instead of a (positive) score, you recorded the times in time trials:

```

scores
|> List.sortBy .elapsedTime
|> List.reverse
|> List.take 3

```

This is a great example of how we can do a lot by composing functions, once we get to know the libraries. This is why pretty much all non-dead programming languages are picking up functional functionality!

Sticking with our contest theme, if we only want one winner, we can use

```
List.maximum : List comparable -> Maybe comparable
```

or

```
List.minimum : List comparable -> Maybe comparable
```

depending on whether we are sorting by score or elapsed time. The interesting thing is that we don't need to sort a list to find the max or min, but we do to pick out the top 3! So these functions will save you some electrons. But unlike `List.take 3`, they don't return a possibly empty `List`. Instead, they return a `Maybe`. So your code may have to be a bit more complex. Even if you are reporting the top 3, however, it is probably a good idea to announce that nobody finished the race rather than just not announcing anything, otherwise people may assume your app is broken. This is a special case of Norman's principle of *feedback*.

```

case scores |> List.maximum of
  Just (score, name) -> name ++ " wins!" |> text |> filled green
  Nothing             -> "Nobody finished :(" |> text |> filled ← red

```

By now, you are probably thinking the list of `List` functions goes on and on. Well, it doesn't there are just a few odds and ends. Sometimes we need to know how big a list is

```
List.length [apple,banana,carrot] -> 3
```

Or we want to put our list together for display

```

List.intersperse ", " ["apple","banana","carrot"]
|> String.concat
-> "apple, banana, carrot"

```

Or to separate out two `Lists` from a `List` of pairs with `List.unzip`:

```

scores
|> List.sort
|> List.take 3
|> List.unzip      -> ( [8,7,6]
                           , ["Crystal","Abdul","Betsy"] )

```

Or `List.partition` the records of students who completed their homework, from those who didn't:

```

(onTimeStudents, tardyStudents) =
  studentRecords
  |> List.partition .homeworkComplete

```

TODO TODO TODO - this should be inside containers

## 6.4.3 Result, Maybe

[to contents](#)

It is daunting to design a complex new application, and usually, we are happy to get the basic interactions to work. But what happens when things don't go according to plan? How do we handle bad data? How do we handle lost network connections?

Elm's compiler helps us eliminate most of the cases of crashing apps, simply by making sure that we handle all cases. A key part of how this works is the use of [Maybe](#) and [Result](#) types.

It truly is annoying when much or even the majority of our code is about handling errors, but think about things from the user's point of view. I'm sure you have tried to use a web application and received something like this



Contrary to urban myth, government contracts *do not* require a quota of cryptic errors. This is just a sign that even spending a billion dollars on software engineers doesn't guarantee a good job.

You probably think I'm going to say, well they didn't have access to the best tools. Well, it turns out, this unnamed corporation invented some very good tools you probably use every day, and supported the development of multiple functional languages! Let's see how the free and open-source Elm handles it.

Elm has a carrot and stick approach to handling errors: the stick is the failure to compile code which does not handle all cases, while the carrot is [andThen](#), which we are going to learn about now.

At this point, you may wonder why error handling is in the Containers section, right! Well, that is the beauty of the Elm approach: thinking of the valid values as being wrapped in a safety blanket—the blanket being the container.

Let's look at these two types together:

```
type Maybe a
  = Just a
  | Nothing
```

and

```
type Result errTy ty
  = Ok ty
  | Err errTy
```

They each have two constructors, and act very similarly to `List`s with at most one element:

List	Maybe	Result
[value]	Just value	Ok value
[]	Nothing	Err "Oops!"

In this example, the error type associated with `Result` is `String`. This is a common choice, which makes it easy to report errors to the user, but if our goal is to fix things without reporting or in addition to reporting to the user, using a `String` means that we cannot rely on the compiler to tell us if we have accounted for all possible errors in a `case` expression.

We can easily transform back and forth from `Maybe` to `List` values, via the standard function

```
List.head : List ty -> Maybe ty
```

and

```
fromMaybe : Maybe ty -> List ty
fromMaybe m =
  case m of
    Just value -> [value]
    Nothing      -> []
```

We could do the same for `Result`, or compose these functions with the library functions

```
Result.toMaybe   :           Result errTy ty -> Maybe ty
Result.fromMaybe : errTy -> Maybe          ty -> Result ty errTy
```

Note that, here, we gain/lose information in the not-normal cases. A `List` with multiple elements loses everything after the first element. An error `Result` loses the attached information about the error type (`:errTy` above). Since `Maybe` has the least information, it cannot lose anything.

In many cases, we need a value, whether the computation to produce one works or not. If it fails, we need to have a fall-back value. For this, both libraries provide `withDefault`:

```
Maybe.withDefault : ty -> Maybe ty      -> ty
Result.withDefault : ty -> Result ty errTy -> ty
```

This is typically how you use it: Get a list of numbers from somewhere:

```
commaSeparatedFloats = "1,2,3.14,,5.25,six,7,6,5,4, 3 , 2 ,1"
```

Now, split the string at the commas, `trim` the extra spaces, convert each trimmed `String` to `FLOATs`, then convert the resulting `Maybe FLOATs` into `FLOATs` by taking the recognized numbers and replacing the errors with the default value 0:

```
numbers =
  commaSeparatedFloats
  |> String.split ","
  |> List.map String.trim
  |> List.map String.toFloat
  |> List.map (Maybe.withDefault 0)
```

That was a blizzard of functions, let's take a graphics break, and show how to use some to make a bar chart which grows.

```
myShapes model =
  let
    oneBar x = rect 4 x
      |> filled green
      |> move (0,0.5*x) -- align bottoms
      |> scaleY ( if model.time < 5 then
                    model.time
                  else
                    5
                  )
  in
  List.indexedMap
    ( \ idx x -> oneBar x |> move (-60 + 5 * toFloat idx, 0) )
  numbers
```

Do try this at home<sup>16</sup>!

How container-like are these? Can we put things in them? Check. Can we take things out? Check. What about mapping? Yes, we can apply mapping in order to operate on values without unwrapping them. For example, we can calculate the area of a rectangle whose length and width *may be* given as `Strings`?

```
Maybe.map2 (*)
  (String.toFloat "4")
  (String.toFloat "3")      -> Just 12
Maybe.map2 (*)
  (String.toFloat "four")
  (String.toFloat "3")      -> Nothing
```

The full set of mapping functions allows us to continue a computation using inputs which may or may not be available. Rather than checking each input, the mapping functions operate on the values if available, and if *all* of them are available, the result is returned, wrapped in a `Just`. Mapping functions are available from simple `map`

```
Maybe.map : (a -> b) -> Maybe a -> Maybe b
```

all the way to

```
Maybe.map5 :
  (a -> b -> c -> d -> e -> value)
  -> Maybe a
  -> Maybe b
  -> Maybe c
  -> Maybe d
  -> Maybe e
  -> Maybe value
```

---

<sup>16</sup>floats: <https://cs1xd3.online>ShowModulePublish?moduleId=20ce50ca-acd4-42c4-867e-912ca8438b88>

In each case, the first argument is the combining function with the right number of inputs.

The mapping functions for `Result` work similarly, but note that the `Result.Err` case has an error code associated with it? What happens if there are multiple errors?

```
Result.map2 (*) (Result.Ok 3)      (Result.Ok 2)      ->Ok 6
Result.map2 (*) (Result.Err "Ruh") (Result.Ok 2)      ->Err "Ruh"
Result.map2 (*) (Result.Ok 3)      (Result.Err "Roh") ->Err "Roh"
Result.map2 (*) (Result.Err "Ruh") (Result.Err "Roh") ->Err "Ruh"
```

Now if you are wondering if, in the last case, could you get `Result.Err "Ruh-Roh"` as the error—yes, you could, if you defined your own result type in which the error is always a `String`. We cannot do that with the library function, because we don't know what type will be assigned to the `Err` case, so we don't know how to combine values. This is the limitation of defining containers so that any value can be put in the container.

Finally the carrot! Mapping allows you to perform computations in the case that the inputs are successful, but what about composing a sequence of steps, each of which could fail? For this, Elm provides

```
Maybe.andThen : (a -> Maybe b)    -> Maybe a    -> Maybe b
Result.andThen : (a -> Result x b) -> Result x a -> Result x b
```

Let's say we need to decode a time of the form `hours:minutes`. We have seen that we can use `String.split ":"` to chop the two pieces, and then `trim` away the extra spaces. But then we have to convert both parts to `Ints` and check that they are in valid ranges. In this case, we probably do want to check that there are exactly two valid numbers separated by one colon. Otherwise, this data is probably meant to be something else! We need to do the split before checking for two components, but we can save some typing by applying all transformations which happen to both components in the same way, before checking that we have exactly two components using a `case` expression:

```
timePair =
  case favouriteClassTime |> String.split ":" 
    |> List.map String.trim
    |> List.map String.toInt of
      [str1,str2] -> (str1 |> Maybe.andThen (inRange 0 23)
                        ,str2 |> Maybe.andThen (inRange 0 59)
                        )
      otherwise     -> (Nothing,Nothing)
```

To do this, we created our own range-testing function:

```
inRange low high x =
  if x >= low && x <= high then
    Just x
  else
    Nothing
```

If it works, we can display a clock, otherwise, we can display a broken clock<sup>17</sup>.

---

<sup>17</sup>clock: <https://cs1xd3.online>ShowModulePublish?modulePublishId=90ccf3bb-5aa6-4be8-a45c-d5f19f22a128>

We won't repeat this for the `Result` type. It is pretty much the same, but we'd need to use functions which return `Result`.

There is one other thing we might need for `Results`. Since they do have the second type, we can also map over that:

```
Result.mapError : (errTy1 -> errTy2) -> Result errTy1 a  
                                -> Result errTy2 a
```

We could use this if we define our own internal error type to help us recover from some errors. At the point where we cannot recover and need to report the error to the user, we can map the `errTy1 -> String`.

#### 6.4.4 Array, Dict, Set

[to contents](#)

`List`'s play a special role in programming, and have a richer set of functions. The remaining containers are extremely powerful, but nevertheless have simpler interfaces. Let's do them together.

First, what are they good for? Well, `Set` is easiest to distinguish. It is like a mathematical set, in that things can be in or out, and that's it. Order doesn't matter—although internally they do have to be stored in some order. Once something is in the `Set`, adding another copy results in the same `Set`. This is very different from `List`. So when should we use `Set`? When what we need to remember is the in-out relationship of a set, of course! The implementer of the library now has opportunities to optimize the code for this situation. They could implement `Set` as a `List`, but that is very unlikely to be efficient. Elm has one implementation for each container type. Other languages, however, may provide several implementations, and if performance is really important, it's best to benchmark using a test similar to your actual application.

Next, we have `Dict` which is essentially a labelled set. So things can be in or out, but if they are *in*, then they have a label attached to them. We call this a dictionary because it works like a paper dictionary, which is a set of words, each of which is labelled by a definition (or a translation to another language, etc., for specialized dictionaries). A long, long time ago, people didn't use dictionaries, because they were complicated to write and most languages didn't have good implementations. If it were up to you to build your own dictionary, you would probably come up with a hack to use an array. Having been around for a long, long time, I can tell you this led to a lot of bad performance. The sad thing is that many people are still learning to program using arrays (and loops), when arrays are no easier to learn than dictionaries, and loops are no easier to learn than mapping and filtering. Of course, you are free to reenact the bad old days when we only had arrays, just as you are free to live in a drafty tent and haul water in a leaky wooden bucket.

Which brings us to `Arrays`! Never heard of them? Think of them as `Dicts` indexed by `Ints`, or as `Lists` with some special access functions. The special case, `Array Bool` can also imitate a `Set Int`. If you need to access the contents in a random order, `Arrays` can be fast, but speed shouldn't be the first, second or third thing you think about. First you should pick the container whose access and creation functions make it easier to write code for your use case. If after you finish writing your app, you determine that it is not performing well, then you should look at the algorithms you use. Are there faster algorithms, because that

will almost give you a bigger bump than changing containers. Finally, if all those have been done, you should try benchmarking the other containers. In software development, we talk of “premature optimization” as a bad habit—especially among the best programmers—who cannot resist implementing something efficiently, even though they don’t know if it will have a noticeable impact, and even though they know other programmers will find it harder to understand and adapt their code.

If performance is a concern in your application, there are two parts to calculating time:

$$(\text{number of operations}) \times (\text{time per operation}). \quad (6.2)$$

Replacing the natural values in an algorithm with `Int` indices can be efficient, but it can also make it confusing and hard to write safe programs. So using `Arrays` may reduce the (time per operation), but if it makes it harder to understand the algorithm and make changes to the algorithm itself which reduce (number of operations), then we will often end up with worse performance, because changes in (time per operation) are very rarely more than a factor of 2, but changes to the algorithm itself can turn something which took  $2^n$  operations into something which takes  $3n$  operations. When the problem size,  $n$ , is more than 10, this is already a hundred-fold difference, and the gap will grow, and fast!

One of the great things about Elm’s libraries is that they make it as easy as possible to switch between containers if you suspect you could get better performance with another one. The same common interface function names also make learning that much easier!

So what can you do with these containers? We will examine them together, noting similarities and differences, starting with the types:

```
type Set    keyTy
type Dict   keyTy valTy
type Array      valTy
```

You can name the type arguments anything you want<sup>18</sup>, but it is helpful to think of the argument for `Set` as an index, and the argument for `Array` as a value. A `Dict` needs both, which fits with the fact that we can easily adapt a `Dict` to any place we needed a `Set` or `Array`.

We create an empty container the same way:

```
Set.empty : Set keyTy
Dict.empty : Dict keyTy valTy
Array.empty : Array valTy
```

which may cause you to think about the container as getting filled up. This is not quite right. In fact, when we add or remove something from a container, we are creating a new container, and any references to the original still work. If this required copying the old container over into a new region of memory, this would be slow. Fortunately, Elm and other languages with immutable data structures lean on<sup>19</sup> some pretty sophisticated data structures, in which the ghosts of Christmas past, present, and future all share storage space efficiently.

<sup>18</sup>Actually, any type you want for `valTy`, but `keyTy`s need to be comparable.

<sup>19</sup>The best place to start learning about these structures with a focus on immutability (purely functional data structures) is Okasaki’s book, but there is an incredible thread on StackExchange with further info: <https://cstheory.stackexchange.com/questions/1539/whats-new-in-purely-functional-data-structures-since-okasaki>.

Starting from `empty` is especially useful for starting off folds. But most often, you will probably create your containers from a `List`:

```
Set.fromList : List keyTy      -> Set keyTy
Dict.fromList : List ( keyTy, valTy ) -> Dict keyTy valTy
Array.fromList : List           valTy -> Array           valTy
```

and for further processing you may need to convert back:

```
Set.toList : Set keyTy      -> List keyTy
Dict.toList : Dict keyTy valTy -> List ( keyTy, valTy )
Array.toList : Array valTy   -> List           valTy
```

Note that you can convert back-and-forth between the other container types and `Lists` this way. If you want to convert between other container types themselves, you need to create an intermediate `List`. If in processing the list, you need the index, you can use `List.indexedMap`, but it may be convenient—and is necessary if converting to a `Dict`—to use

```
toIndexedList : Array valTy -> List ( Int, valTy )
```

to convert to a `List ( Int, valTy )`. Similarly, you may only need the values or the indices (also called keys) from a `Dict`, one of the reasons for which is that you want to convert to a `Set` or `Array`.

```
Dict.values : Dict keyTy valTy -> List valTy
Dict.keys   : Dict keyTy valTy -> List keyTy
```

Converting back and forth may seem wasteful, but in some cases, it is actually very efficient. For example, if you need the set of unique values appearing in a `Dict` the fastest way is to build a `Set` and then convert to whatever container most appropriate.

That said, Elm's libraries make it easy to process data in-place, and avoid these conversions, with processing for `Set` and `Array` looking identical, and the `Dict` functions being just a little bit more complicated because there are both index and value types involved. There is mapping:

```
Array.map : (a -> b)      -> Array a  -> Array b
Set.map  : (a -> b)      -> Set a    -> Set b
Dict.map : (k -> a -> b) -> Dict k a -> Dict k b
```

Note that the key is provided to `Dict.map` and can be used to modify the value, but the interface doesn't give you a way of changing the key. To remap keys, you have to convert to a `List`, remap there and convert back.

Filtering is similar:

```
Array.filter : (a -> Bool)     -> Array a  -> Array b
Set.filter  : (a -> Bool)     -> Set a    -> Set b
Dict.filter : (k -> a -> Bool) -> Dict k a -> Dict k b
```

and again you can use the key to help filter the `Dict` entries.

Filtering both in and out at the same time, as we saw with `Lists` does not exist for `Arrays` but works the same way for `Sets`:

```
(onTime, tardy) = studentRecords
    |> Set.partition .homeworkComplete
```

and `Dicts`:

```
(onTime, tardy) = studentRecords
    |> Dict.partition .homeworkComplete
```

Folding looks similar:

```
Array.foldl : (a -> b -> b)      -> b -> Array   a -> b
Array.foldr : (a -> b -> b)      -> b -> Array   a -> b
Set.foldl  : (a -> b -> b)      -> b -> Set     a -> b
Set.foldr  : (a -> b -> b)      -> b -> Set     a -> b
Dict.foldl : (k -> a -> b -> b) -> b -> Dict k a -> b
Dict.foldr : (k -> a -> b -> b) -> b -> Dict k a -> b
```

The fact that we have both left- and right-folds which process elements from lowest to highest, and highest to lowest respectively, gives away that part of the efficiency of `Set` and `Dict` types is that their internal data structures encode the order.

Less exciting, but also important to have, are functions to test for emptiness:

```
Set.isEmpty : Set keyTy      -> Bool
Dict.isEmpty : Dict keyTy valTy -> Bool
Array.isEmpty : Array valTy   -> Bool
```

To find the size:

```
Set.size   : Set keyTy      -> Int
Dict.size  : Dict keyTy valTy -> Int
Array.length : Array valTy   -> Int
```

But only `Set` and `Dict` let us check for membership:

```
Set.member : keyTy -> Set keyTy      -> Bool
Dict.member : keyTy -> Dict keyTy valTy -> Bool
```

And for the value-containing two, we can look up entries via

```
Dict.get      : keyTy -> Dict keyTy valTy -> Maybe valTy
Array.get    : Int    -> Array    valTy -> Maybe valTy
```

This doesn't make sense for `Set`, because it only contains keys, so if we know the key there is nothing to look up, other than membership. The `get` functions can do the job of looking up and telling us about membership, if we used them in a `case` expression. For example, if the value type is `String`, we could prepare it for display:

```
case Array.get key orders of
  Just value ->
    "The value at "++String.toInt key++" is "++value++"."
  Nothing ->
    "There is entry for "++String.toInt key++"!"
```

To add one thing into a container, we have

```
Set.insert : kTy          -> Set kTy          -> Set kTy
Dict.insert : kTy -> vTy -> Dict kTy vTy -> Dict kTy vTy
```

But what if the element is already in the container? The `Set` is not altered, but the new `Dict` will contain the new value. `Array.set` works the same way for replacing a value at an index already in an array:

```
Array.set    : Int    -> valTy -> Array valTy      -> Array valTy
```

But beware! If the index is outside the range of the `Array` the returned `Array` is the same as the input `Array`. `Dict` has the option to take the best of both worlds—the new and the old—using a combining function, as the second argument:

```
Dict.update : keyTy -> (Maybe valTy -> Maybe valTy)
              -> Dict keyTy valTy
              -> Dict keyTy valTy
```

The combining function's input `Maybe` indicates the presence of the entry in the `Dict`, and the output `Maybe` allows you to set a new (or old) value, or have that entry removed by returning `Nothing`.

Conversely, we can remove elements: To remove one thing from a container, we have

```
Set.remove : keyTy -> Set keyTy      -> Set keyTy
Dict.remove : keyTy -> Dict keyTy valTy -> Dict keyTy valTy
```

and if the element is not present, we will get back the same container as we input. Maybe it should be called `makeSureItsReallyGone` rather than `remove`.

The view that a `Dict` is a labelled set is reinforced by the language of set math,  $\cup$

```
Set.union     : Set kTy      -> Set kTy      -> Set kTy
Dict.union    : Dict kTy vTy -> Dict kTy vTy -> Dict kTy vTy
```

$\cap$

```
Set.intersect : Set kTy      -> Set kTy      -> Set kTy
Dict.intersect : Dict kTy vTy -> Dict kTy vTy -> Dict kTy vTy
```

and \

```
Set.diff      : Set kTy      -> Set kTy      -> Set kTy
Dict.diff     : Dict kTy vTy -> Dict kTy vTy -> Dict kTy vTy
```

But what about the values in the `Dict`? Well, the rule is that we keep the value from the first `Dict` if there is a conflict. Don't like it? You can get any behaviour you want with

```
Dict.merge :
  (keyTy -> valATy ->                  foldTy -> foldTy)
  -> (keyTy -> valATy -> valBTy -> foldTy -> foldTy)
  -> (keyTy ->                      valBTy -> foldTy -> foldTy)
  -> Dict keyTy valATy
  -> Dict Ty valBTy
  -> foldTy
  -> foldTy
```

which does a fold over the union of the keys, ordered from lowest to highest. Wait a minute, you’re thinking. How does a fold, which reduces a whole container of values to a single value replace set math, which produces a new `Dict`? Well, a new `Dict` is a value! And unlike `Dict.union`, which requires that the two input and one output `Dict` all have the same value type, `Dict.merge` let’s us combine `Dicts` with different value types.

To understand it, let’s give the arguments names:

```
Dict.merge : foldA foldAB foldB dictA dictB init =
  let
    ...
  in
  final
```

So, it is actually a triple fold. We fold differently depending on whether a particular key is in `dictA`, `dictB`, or both. The types give away which is which. Let’s see if you are developing an imagination for creatively using APIs (Application Programmatic Interface—the functions provided by a library)? Can you solve these challenges?

1. Given two `Int` dictionaries, how would you add up all values, including both values when a key is in both dictionaries?
2. Given two `Int` dictionaries, how would you add up the maximum value for each key?
3. Given two `Int` dictionaries, how would you take the product of the minimum value for each key?
4. Given two dictionaries, how would you return the `Set` of all keys?
5. Given two dictionaries, how would you return the `Set` of keys in exactly one of the dictionaries?
6. Given two `Int` dictionaries, how would you return the dictionary with the union of the keys, in which each value is the sum of all of the values for that key?

We finish the `Dict` and `Set` functions off with the least function—`singleton`. Making containers with one element, as we had for `List`:

```
singletonDict = Dict.singleton key value
singletonSet  = Set.singleton key
```

Note that there is no equivalent to `[]` for these containers, but if you really want to avoid `singleton` you could use

```
singletonDict = Dict.fromList [(key,value)]
singletonSet  = Set.fromList [key]
```

We also do not have an equivalent of `(::)`. Where you will miss this, is if you want to process elements of your container one element at a time, using a `case` expression. These other containers do not expose a natural order to the programmer. You can easily convert `toList` and process them as you would a `List`, but think twice about doing so, since in most cases you will be better off using a `fold`.

Notice that `Array` was not invited to this festival of set math? Well, don’t worry, it has its own set of functions suited to its linear nature. More flexible than `List.range`, we have

```
Array.initialize : Int -> (Int -> valTy) -> Array valTy
```

which takes a size, and a generator function to create the values for the `Array`. If you just want the `List.range` behaviour, you can use the `identity` function from the previous footnote:

```
Array.initialize 7 identity
                  -> Array.fromList [0,1,2,3,4,5,6]
                        ]
Array.initialize 4 ( \ x -> 0.5 * toFloat x )
                  -> Array.fromList [0,0.5,1,1.5]
```

You may like this so much, you want to create one for `List` as an exercise.

If you need the same element repeated, you can do that too

```
Array.initialize 5 ( \ x -> 5 )
                  -> Array.fromList [5,5,5,5,5]
```

or use a shortcut

```
Array.repeat 5 5
                  -> Array.fromList [5,5,5,5,5]
```

Also like `List` we can join two `Arrays` together

```
[1,2,3] ++ [4,5,6]      -> [1,2,3,4,5,6]
Array.append (Array.fromList [1,2]) (Array.fromList [3,4])
              -> Array.fromList [1,2,3,4]
```

But it is quite a bit more typing!

But whereas `List` really wants you to add and remove elements from the head, `Array` is set up to add and remove from the end:

```
Array.push 3 (Array.fromList [1,2]) -> Array.fromList [1,2,3]
```

and if you know about the `stack`<sup>20</sup> data type, you would expect to find a `pop` to take the last element off the `Array`, but instead we have to use

```
Array.slice : Int -> Int -> Array valTy -> Array valTy
```

in a special case. Slicing<sup>21</sup>, in general, means taking a linear subset of an n-dimensional array, which can include skipping odd elements, and collapsing some dimensions. Elm only has one-dimensional arrays, and does not support skipping, but it does support extracting arbitrary contiguous subarrays, specified by a first and last index. It even understands negative indexing, where negative numbers (which don't make sense as normal indices) are understood to mean counting backwards from the end of the `Array`.

```
Array.slice 0 3 (Array.fromList [0,1,2,3,4])
                  -> Array.fromList [0,1,2]
```

<sup>20</sup>stack: [https://en.wikipedia.org/wiki/Stack\\_\(abstract\\_data\\_type\)](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))

<sup>21</sup>Slicing goes back to 1957! [https://en.wikipedia.org/wiki/Array\\_slicing](https://en.wikipedia.org/wiki/Array_slicing), but make sure to try APL, <https://tryapl.org>, a language almost built around slicing and dicing arrays—you'd think it were called Array Programming Language.

```
Array.slice 0 -2 (Array.fromList [0,1,2,3,4])
                           ↪ Array.fromList [0,1,2]
Array.slice 3 -2 (Array.fromList [0,1,2,3,4])
                           ↪ Array.fromList []
```

We've covered a lot in this section, but you can also *do* a lot with containers, if you learn to use them.

## 6.4.5 Bitwise

[to contents](#)

Operations on `Ints` by mapping `Bool` operations over the individual bits:

```
Bitwise.and      : Int -> Int -> Int
Bitwise.or       : Int -> Int -> Int
Bitwise.xor      : Int -> Int -> Int
Bitwise.complement : Int        -> Int
```

And rotating those bits as if they are a list:

```
Bitwise.shiftLeftBy   : Int -> Int -> Int
Bitwise.shiftRightBy   : Int -> Int -> Int
Bitwise.shiftRightZfBy : Int -> Int -> Int
```

Shifting left multiplies by a power of 2, just like shifting a decimal number left by one digit multiplies by 10. Shifting right divides by a power of 2.

The last function `Bitwise.shiftRightZfBy` depends on the number of bits in the `Int`, and is therefore dangerous to use.

What do we need these operations for? The most likely native application in a web app would be manipulating black-and-white pixel arrays packed into `Ints`. There are, however, many storage formats which involve packed bit arrays, so if you need to access any of them, these functions will help you unpack and repack such data. In either case, it's probably best to learn to use these functions as needed.

## 6.5 Development

[to contents](#)

### 6.5.1 Debug

[to contents](#)

The `Debug.todo` module is misnamed! The renaming reflects how other people develop code in non-functional languages. I think it should have been called `Develop`, because it has functions which are useful during development. Most obviously there is

```
Debug.todo : String -> a
```

which is for things on your to-do list. You don't need this when you are just typing out your code, but you do need something like it if you want to fix compiler errors as you go, or make adjustments to the visual layout of your app piece by piece.

What `Debug.todo` does is crashes your program with a message saying where it crashed. This could be useful if you *thought* you were implementing a self-contained subset of your code for this milestone, only to discover that it isn't so self-contained, and there is no way to try it out without completing another chunk of code. No foosball tournament for you!

When should you use it? Easy, in the “model” and “update” parts of model-view-update, meaning in the `init` or `update` functions and functions called by them directly or indirectly. What about “view”, i.e., `myShapes`? Well, it might work there, but if your unfinished piece of work will produce a panel on your interface, and that panel is visible even if you don't interact with it, then you won't get very far inspecting the panels which are finished, because your app will crash before you get to see it. Instead, for functions which return `Shapes` or `Stencils` (or later `Html`) you should just put in a simple message saying this part is not finished.

If you know that the unfinished component is called Bob and will be contained in a circle of radius 20, then

```
bob model =
  [ circle 20 |> filled yellow
  , text "Bob" |> centered |> filled black
  ]
  |> group
```

will let you see how “Bob” fits into the rest of your interface—and won't crash your app!

Wow, that seems really useful. Why not do that everywhere? Because not every function returns a shape you can see, and code in the `update` function may perform vital tasks without any immediate change in what the user sees. So for those to-do's, we need to use `Debug.todo`.

Often it will be used as placeholder for cases you haven't written yet:

```
makeDots n =
  case n of
    0 -> ""
    1 -> "."
    2 -> ".."
    _ -> Debug.todo "Time for a nice cold glass of nimbu pani."
```

What about showing things which are not `Shapes`? The simplest development function turns any type into a `String` which you can display using `text`.

```
Debug.toString model      -> "{ time = 3.1415 }"  
Debug.toString (3,5)     -> "(3,5)"
```

Since we already have `String.toInt`, and `String.toFloat`, why do we need this? We don't, we could write our own *efficient* `toString` functions, but this is convenient. To reinforce that this expensive function is only for development, your code will not compile with optimizations if it contains `Debug` functions, because it cannot easily be made fast in a reliable way.

This stopwatch example<sup>22</sup> shows how useful it can be for displaying records.



```
{ elapsedTime = 7.599, runningSince = Nothing, time = 31.944 }
```

I personally like seeing the instrumented state information in the same view with the output, and even have it stick to objects, but if you try displaying a complex `Model` type, it won't fit in a reasonable space. In that case, you can open up the JavaScript console and watch for messages produced by

```
Debug.log : String -> a -> a
```

This is what old-timers call debug-printfs (if said old-timers programmed in C). There are a lot of problems with debugging complex programs this way, not least of which is that the timing of the print statements is often not what you expect, since even if your program doesn't use multiple threads<sup>23</sup>, the run-time system for your language may.

Having been warned, you can use this logging function to output the `String` argument to the console whenever that computation happens. For example, we could replace this line from the above example

<sup>22</sup>stopwatch: <https://cs1xd3.online>ShowModulePublish?modulePublishId=86fce854-0217-4f3a-b3a9-dfd51b96c57c> If you don't understand how the toggle button works, reread Chapter 4.

<sup>23</sup>Threads are like programs which run on their own timeline and send messages back and forth to each other to synchronize their work. Great if you have multiple processor cores to speed up the work, and actually simpler for some types of applications, but generally confusing for beginners finding it hard to keep up with a single billion-operation/second core.

```
, Debug.toString model
    |> text
    |> size 5
    |> filled black
    |> move (-93,-60)
```

with

```
, let debugString = Debug.toString model
in debugString
    |> text
    |> size 5
    |> Debug.log debugString
    |> filled black
    |> move (-93,-60)
```

So that whenever the model is prepared for display on the screen, it will also be sent to the JavaScript console. Why did I put it after `size`? Just to show I could put it anywhere, because it doesn't do anything with the second argument except return it.

## 6.6 Elm Platform: Communicating with the outside world

This module lets you communicate with the world outside the browser and with the browser's run-time system itself. It will be difficult for you to do this without more advanced knowledge, but you can give it a try using the documentation <https://package.elm-lang.org/packages/elm/core/latest/Platform>.

### 6.6.1 Platform.Cmd

[to contents](#)

So far, we have talked about pure interactive programs using `model`, `view`, and `update` (see Section 4.5). This is great because we can write programs that do predictable things, where an input always leads to the same output. But many more interesting programs need to be able to inject unpredictable data through communication with the outside world. Why would we want to lose control of our program, you ask? Here are some good examples of unpredictable data: data that changes depending on the time, the weather, or the position of the planets. Random numbers! All of these things must come from outside the pure functions `update` and `view`, which is what `Platform.Cmd` is for. Let's start with an easy example.

### 6.6.2 Random Numbers: Let's roll the die

[to contents](#)

In a computer, it is difficult to generate truly random numbers. This is because, at their core, computers are very predictable beasts. They always give the same output for a given input. As we've mentioned, this is called *pureness*. But, it is possible to generate *pseudo-random* numbers. These are numbers which "appear" to be random to humans or even to statisticians, but they are in fact not truly random. Instead, they use something like the current time as a "seed" to generate a random number. For anyone who is a gamer, you'll

know that a seed is a unique number which always generates the same random Minecraft world or random loot drop. This works very similarly!

Elm in particular is a pure programming language, meaning we are forced to always give the same output for a given input. Even pseudo-random numbers are hard for Elm. So, how can we generate them? Well, instead of generating one directly in Elm, we can ask Elm's runtime nicely, "Please generate me a random number, and when you're done, send me a message telling me what you came up with." This is the idea of commands. We simply ask Elm to do something for us, and it will do it and send a message when it has the result. Just like the types of messages we got when the user clicked a button earlier in the book.

Let's make a program to roll a die<sup>24</sup>! We can easily represent a die as an integer number from 1 to 6. Let's make a program with three messages:

```
type Msg = Tick Float GetKeyState
    | RollDie
    | DieRolled Int
```

In addition to our regular `Tick` message which we could use for animation, we have a message `RollDie` and one called `DieRolled`. Notice that `RollDie` has no inputs, and `DieRolled` takes an integer as an input. The `RollDie` message will be used to tell Elm to generate a random number for us. The `DieRolled` message will be the one Elm sends back to us with its freshly generated random number. Neat!

But how can we tell Elm to do things, like generate a random number? This is done through Commands. We need to use a modified `update` function. Remember, when we started with animations, we didn't even need to modify the `update` function, so it wasn't visible in the Animation activity. When we added interaction, we introduced the function:

```
update : Msg -> Model -> Model
```

It takes as input a message (of type `Msg`) and our current `Model` value and returns a new value of type `Model`. To support `Cmds`, we need to add them to the output:

```
update : Msg -> Model -> (Model, Cmd Msg)
```

Notice we are now returning a `Tuple` of values. The first one is the updated model, as before. But the second one is a `Cmd Msg`. This means every time our update function is called, we have the opportunity to send the Elm run-time system a command which will produce a message later.

So how can we tell Elm to *generate* a random number? Luckily Elm's `elm/random` package<sup>25</sup> has just the thing, a function conveniently called `Random.generate`. Let's take a look at the type signature:

```
Random.generate : (a -> msg) -> Random.Generator a -> Cmd msg
```

Whoa! This one looks complicated! Let's break it down. The first input is a *function* from some value of type `a` to some `msg` type (the type of message we'd like Elm to produce). The second input is of type `Generator a`, which is a recipe for generating something of some type `a`. Finally, the output is of type `Cmd msg`, which is exactly what we were hoping for!

<sup>24</sup>die roll example: <https://stabl.rocks>ShowModulePublish?modulePublishId=409bef9d-dd69-4f54-bb19-251dd94e575e>

<sup>25</sup>elm/random: <https://package.elm-lang.org/packages/elm/random/latest/>

So, what kinds of generators can we make? Turns out, we can create a recipe to generate just about anything you can think of! Let's look at a few that `elm/random` provides for us:

```
Random.int : Int -> Int -> Generator Int
```

means generate an `Int` ranging between two given values;

```
Random.float : Float -> Float -> Generator Float
```

means generate a floating-point number between two values;

```
Random.uniform : a -> List a -> Generator a
```

means pick uniformly from one value and a list of values;

```
Random.weighted : (Float, a) -> List (Float, a) -> Generator a
```

means pick from one value and a list of values, but give each one a weight. Hmm, why not just pick from a list of values? Because `List` can be empty, and it's impossible to pick something from nothing, so the `Random` library prevents us from trying. Clever!

Now let's figure out which one we need to generate a die roll. The simplest way to represent a die is to use an integer where the numbers are constrained from 1 to 6:

```
dieGenerator : Random.Generator Int
dieGenerator =
    Random.int 1 6
```

Let's analyze this snippet a bit. Firstly, our function is called `dieGenerator`. It has as its type `Random.Generator Int`, which means it is a generator that describes a recipe for generating an integer. The body of the function calls the `Random.int` function with the arguments 1 and 6. This will generate us a number between one and six, inclusive, with a uniform probability of each number appearing.

In order to actually call this function, as mentioned, we can use the `Random.generate` function in our update function. Since we're going to use this many times, we'll store this in its own definition:

```
getNewDie : Cmd Msg
getNewDie =
    Random.generate DieRolled dieGenerator
```

As mentioned, the `Random.generate` function takes two arguments: a function describing how to convert our returned random value into a `Msg` and the generator itself, where the type of generator matches the input type of the function. Recall that the data constructor `DieRolled` can also be used as a function of type `Int -> Msg`, so the types work out perfectly.

Now let's take a look at how we can actually use this in our update function. Our update function has three cases, one for each of our messages (the `Tick` message has been omitted for brevity here):

```
update msg m =
    case msg of
        ...
        RollDie ->
```

```
( model, getNewDie )

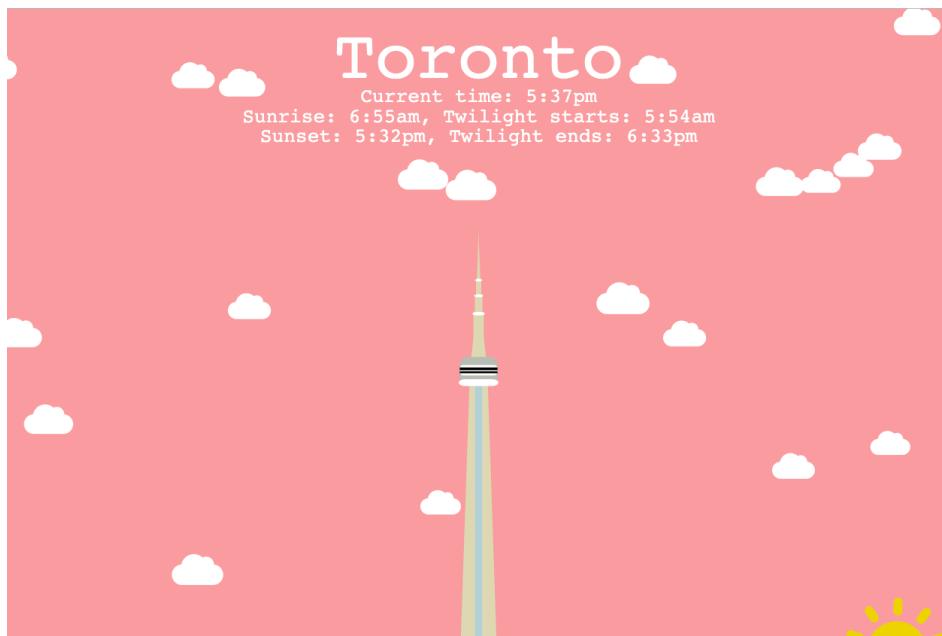
DieRolled newSide ->
( { model | dieSide = newSide }
, Cmd.none )
```

The `RollDie` message can be sent from a button in our program. It will initiate the command to generate our new die roll. Notice that the second element in the tuple is the command we created just now. The `DieRolled` message actually contains the newly generated random number. We can store that random number in the model and then use that to display the die face. Anytime we don't want Elm to do any special commands, we can use the `Cmd.none` function to say "please don't perform any commands right now."

### 6.6.3 Task: Making the world go around

[to contents](#)

Our next example will highlight the use of the `Task` library. `Tasks` are commands which could fail. One common way they are used in Elm is for sending HTTP requests. We will create a nice interactive wallpaper which will display the sun's position in real time. For this, we will use an API provided by <https://sunrise-sunset.org/>. The final program<sup>26</sup> will render this nicely for us.



Let's go over the main things we need to get this working.

The server provides us data in JSON, an standardized, human-readable data format:

```
{
  "results":
```

<sup>26</sup>sunset example: <https://stabl.rocks>ShowModulePublish?modulePublishId=f7d1af67-9998-4ecc-a0c6-43fd57f6b87e>

```

        "sunrise" : "2015-05-21T05:05:35+00:00",
        "sunset"  : "2015-05-21T19:22:59+00:00",
        "nautical_twilight_begin": "2015-05-21T04:00:13+00:00",
        "nautical_twilight_end"   : "2015-05-21T20:28:21+00:00",
    },
    "status": "OK"
    < ... values we don't need ... >
}

```

JSON is so widely used in web services because it is adaptable to any type of records, including nested records.

Elm's `elm/json` library<sup>27</sup> does most of the work of converting JSON-formatted data into Elm data types, but first, we need a type to store the response, with the fields we're interested in:

```

type alias SunTimes =
{
    sunriseTime : Posix
, sunsetTime  : Posix
, twilightBegin : Posix
, twilightEnd   : Posix
}

```

Next, we will need a way to *decode* the Json response into our Elm type. For each field, we can use the `D.at` function to “drill down” into the returned Json and extract the field we need. Then, we will use the `Iso8601` library<sup>28</sup> to parse the time into a time value Elm can understand. Finally, the `D.map4` function maps all these fields into our `SunTimes` type alias.

```

sunsetDecoder : Decoder SunTimes
sunsetDecoder =
    D.map4
        SunTimes
        (D.at ["results", "sunrise"] Iso8601.decoder)
        (D.at ["results", "sunset"] Iso8601.decoder)
        (D.at ["results", "nautical_twilight_begin"] Iso8601.decoder)
        (D.at ["results", "nautical_twilight_end"] Iso8601.decoder)

```

Now that we have our decoder, we need a message that will be received when the Http request is completed. This is a message that can either tell us that the Http request succeeded, in which case we will get the information we want, otherwise it will tell us there was an error. The `SunsetResponse` message thus takes a `Result` which can be an error or `SunTimes` value.

```

type Msg = Tick Float GetKeyState
        | SunsetResponse (Result Http.Error SunTimes)

```

<sup>27</sup>elm/json: <https://package.elm-lang.org/packages/elm/json/latest/>

<sup>28</sup>rtfeldman/elm-iso8601-date-strings: <https://package.elm-lang.org/packages/rtfeldman/elm-iso8601-date-strings/latest/>

...

Next, we need to create our request that will be used to fetch the data. We will use a GET request, as this is what the API calls for. We provide two fields: the URL of the API endpoint<sup>29</sup> and another field called `expect`. The `expect` field describes the type of data to expect as well as the message that should be sent when the request completes.

```
getSunsetTimes : Cmd Msg
getSunsetTimes =
    Http.get
    {
        url = "https://api.sunrise-sunset.org/json?lat=43.6424728&lon=-79.3876551&formatted=0"
        , expect = Http.expectJson SunsetResponse sunsetDecoder
    }
```

In order to let our user know when loads are slow or there is a network error, we need to store this state in the `Model`, for which we create a new algebraic data type:

```
type SunTimesLoadStatus
= NotLoaded
| ErrorLoading Http.Error
| Loaded SunTimes
```

The `update` function needs to be able to receive the message and store the data in the model. In this case, we can pattern-match on the two possibilities, which are that the request either succeeded or failed:

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        ...
        SunsetResponse (Ok sunsetTimes) ->
            ( { model | sunTimes = Loaded sunsetTimes, lastSunsetFetch = model.now } , Cmd.none )
        SunsetResponse (Err err) ->
            ( { model | sunTimes = ErrorLoading err, lastSunsetFetch = model.now }, Cmd.none )
        ...
    
```

Finally, to actually issue the `Http` request, we simply include our `getSunsetTimes` command in our initial function<sup>30</sup>:

```
main : EllieAppWithTick () Model Msg
main =
    ellieAppWithTick Tick
```

<sup>29</sup>In the actual code, we add a modifier to the URL to help reduce the chance of fetching the wrong day due to timezones.

<sup>30</sup>In the full program, , <https://stabl.rocks>ShowModulePublish?modulePublishId=f7d1af67-9998-4ecc-a0c6-43fd57f6b87e> you will find additional commands, like getting the current time and time zone, which follow the simpler pattern we've already seen for random numbers.

```
{ init = \flags ->
    ( init
    , Cmd.batch
        [ getSunsetTimes
        ...
        ]
    ...
}
```

Enjoy your new wallpaper<sup>31</sup>!

## 6.6.4 Platform.Sub

[to contents](#)

Subscriptions are used to receive messages about things that can happen at any moment. For example, we can get messages every 5 seconds, get notified when a user moves their mouse, or get notified when a user hides or shows the tab that the Elm program is running in. For this example<sup>32</sup>, we will subscribe to the user changing the size of the screen, and use this to adaptively change the size of a shape on the screen.

To do so, we will create a program which stores the current screen size:

```
type alias Model =
    { time : Float
    , width: Int
    , height: Int
    }
```

Next, we will create a message to send when the screen updates. It takes two integers, representing the width and height of the screen, respectively:

```
type Msg = Tick Float GetKeyState
        | WindowResize Int Int
```

The `elm/browser` package provides us the necessary function to subscribe to window size changes:

```
Browser.Events.onResize : (Int -> Int -> msg) -> Sub msg
```

This takes a function from two `Int` values to a `msg` value, and produces a subscription we can use to subscribe. We can use this to create a new subscription with our message:

```
resizeSubscription : Sub Msg
resizeSubscription =
    Browser.Events.onResize WindowResize
```

In the update function, we store the values in our model:

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
```

<sup>31</sup>sunsetting: <https://stabl.rocks>ShowModulePublish?modulePublishId=35f7604d-7bfd-45f6-8d2c-509889dda527&fullscreen=true>

<sup>32</sup>resizable oval example: <https://stabl.rocks>ShowModulePublish?modulePublishId=1ecc2048-0ea9-4326-b461-5808388226a8>

```
...
WindowResize w h -> ( { model | width = w, height = h }, Cmd.←
    none )
```

Then, we can use these values in our `myShapes` function:

```
myShapes model =
[
    oval (toFloat model.width) (toFloat model.height)
        |> filled pink
]
```

There is one more thing we need to do: we need to get the current screen size when we start the program. This is because if we don't, we won't get a resize message until the user actually resizes the screen—which may never happen. To do so, we can use a command using a `Task` provided by the `elm/browser` library<sup>33</sup>:

```
initScreenSize : Cmd Msg
initScreenSize = Task.perform
    (\vp -> WindowResize
        (round vp.viewport.width)
        (round vp.viewport.height))
    ) Browser.Dom.getViewport
```

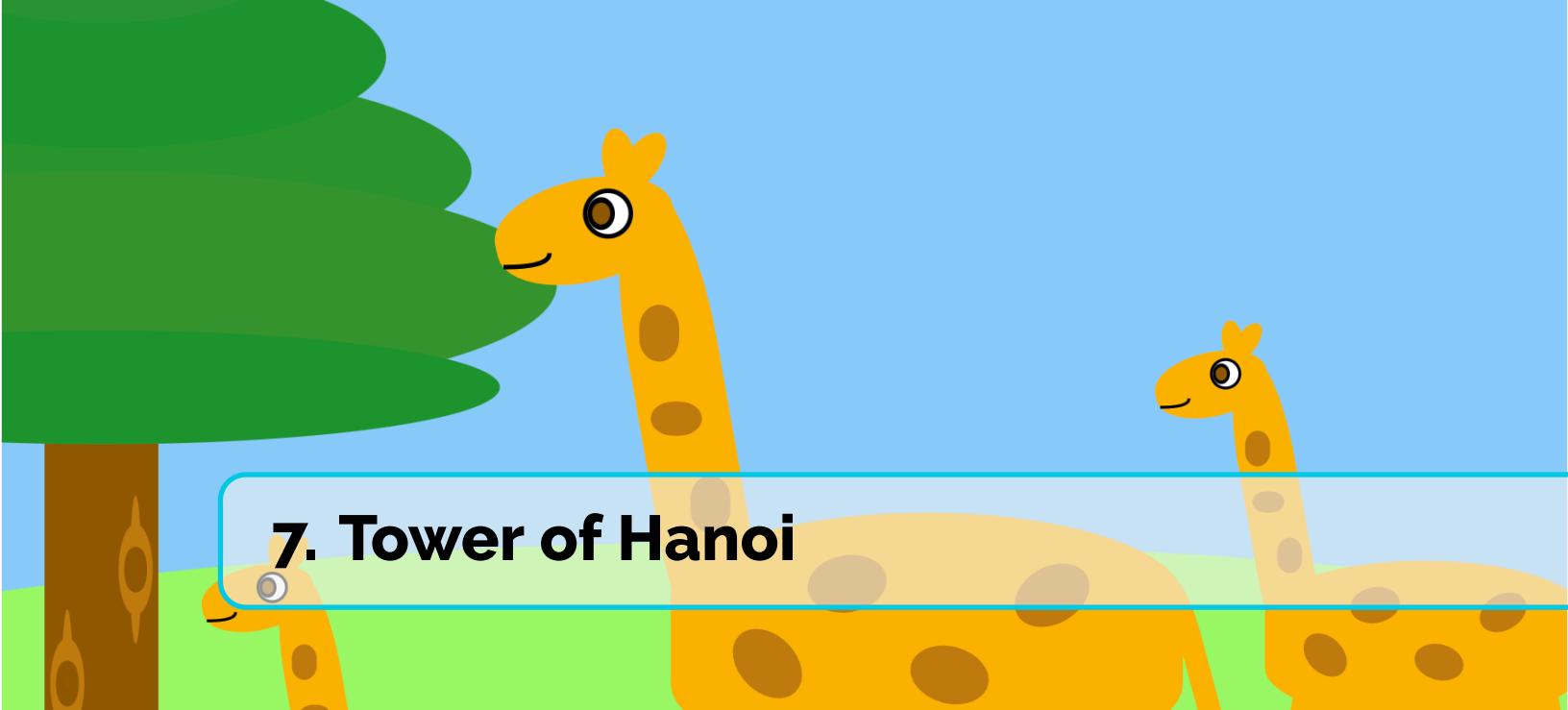
Since this needs to happen once, we put it in our `init` function.

Try it yourself<sup>34</sup>! Resize the screen and observe as the oval changes to match the size of the screen.

---

<sup>33</sup>elm/browser's `getViewport`: <https://package.elm-lang.org/packages/elm/browser/latest/Browser-Dom#getViewport>

<sup>34</sup>resize: <https://stabl.rocks>ShowModulePublish?modulePublishId=0def7b8c-3bf8-4804-a918-f8a4df8bbdd9&fullscreen=true>



## 7. Tower of Hanoi

Let's look at a simple children's puzzle whose solution involves some really powerful ideas.



In the Tower of Hanoi, your task is to move a tower made by stacking blocks of diminishing size, one on top of another. You can move one block at a time to one of three bases. At no time can a larger block rest on a smaller block. All the blocks have different sizes.

When seeing this problem for the first time, what questions should you ask yourself? For which sizes of tower, if any, can we solve the problem? How many moves are required to move the tower with a given number of blocks?

Take some time to think about the problem, and solve it if you can. If your solution is good, you should have no trouble explaining the solution to a friend.

How did you explain your solution to a friend? Did you use notation you have used before in algebra or geometry? Did you make up your own symbolic language? Could you explain your solution entirely in English? Did you need diagrams?

If your math teachers have done a good job, you started by writing down what you know about the problem, and maybe even assigning variable names. Here's what I came up with:

$$n = \text{number of blocks}$$

---

$\{A, B, C\}$  = set of places  
 $\{1, 2, \dots, n\}$  = the blocks

What is harder, moving the blocks, or counting how many moves are required to move the whole tower? Counting may seem harder, but it is actually easier if you use the Divide and Conquer principle to break the problem down. Start by assuming we know the answer, and just need to figure out an efficient way of calculating it. Let  $m_n$  be the number of moves required to move a tower of size  $n$  from one place to another. That gives us simple notation.

Divide and Conquer is a method for people with lots of friends. The hard work is to divide up the problem, then we call on our friends. After all, what is a smaller problem among friends! In this case, if we are asked to move a tower of size  $n$ , we can assume our friends have already figured out how to move a tower of sizes 1 through  $n - 1$ . The key is the  $n - 1$  solution.

Take a few minutes and try to describe a solution to the tower  $n$  problem, assuming that you can call on friends to move a tower of size  $n - 1$ .

Well, if friends are willing to move a tower of size  $n - 1$ , you can wrap paper around the bottom wrung of the tower, and call them over to move the visible  $n - 1$  tower. After the celebratory dance party, you need to move the one on the bottom—admittedly the heaviest—which we will call the  $n$ th block. Then you'll need to disguise the big block again, and get your friends to move the  $n - 1$  tower onto it. If they ask why they couldn't have moved the tower here in the first place, just tell them that it is the fault of your landscape designer, who couldn't make up their mind. How many total steps? We just take their work on the first day, add our one move, and then their work on the second day.

$$m_n = m_{n-1} + 1 + m_{n-1}$$

Let's apply this formula to a specific case:

$$\begin{aligned} m_3 &= m_2 + 1 + m_2 \\ &= (m_1 + 1 + m_1) + 1 + (m_1 + 1 + m_1) \\ &= (1 + 1 + 1) + 1 + (1 + 1 + 1) \\ &= 7 \end{aligned}$$

Now, to get 7, we have used the hopefully obvious fact that we can move a tower with 1 block in 1 move. But, remember, there are no obvious facts for computers! If we forget to tell the computer about the special case  $m_1 = 1$ , it will just keep going, like the sorcerer's apprentice:

$$\begin{aligned} m_3 &= m_2 + 1 + m_2 \\ &= (m_1 + 1 + m_1) + 1 + (m_1 + 1 + m_1) \\ &= ((m_0 + 1 + m_0) + 1 + (m_0 + 1 + m_0)) + 1 + ((m_0 + 1 + m_0) + 1 + (m_0 + 1 + m_0)) \\ &= (((m_{-1} + 1 + m_{-1}) + (m_{-1} + 1 + m_{-1})) + 1 + ((m_{-1} + 1 + m_{-1}) + (m_{-1} + 1 + m_{-1}))) + 1 + (((m_{-1} + 1 + m_{-1}) + (m_{-1} + 1 + m_{-1})) + 1 + ((m_{-1} + 1 + m_{-1}) + (m_{-1} + 1 + m_{-1}))) \\ &= \dots \end{aligned}$$

continuing until the end of time or it ran out of memory.

Here is another good lesson. No matter what silly thing a computer does, people have to take the blame, because people are doing the thinking, coming up with a set of rules which the computer faithfully carries out. Until we figure out rules for common sense—and nobody is working on this, since nobody has a definition to work from—computers will not have it, and while that may come sooner than many expect, you can bet it won't come before your next assignment is due—so you figure out the rules.

We could express our procedure as follows: Given an  $m_n$  to calculate, use repeated substitutions to reduce  $m_n$  to a number, where we replace

$m_1$  with 1

and for any values of n bigger than 1, we replace

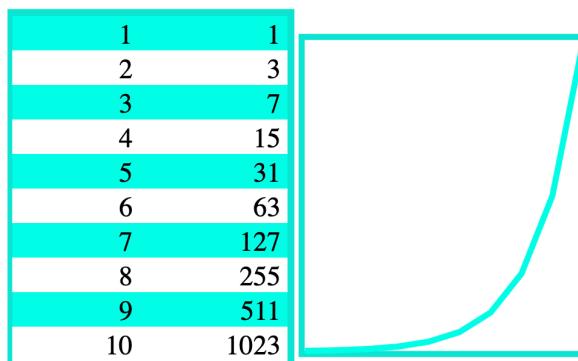
$m_n$  with  $m_{n-1} + 1 + m_{n-1}$ .

There isn't really any point in writing all of this out, since a person would probably figure that out, but it happens to be very close to how we would use Elm to tell a computer the same thing.

```
m n = case n of
    1           -> 1
    otherwise -> m (n-1) + 1 + m (n-1)
```

Note a few differences between standard algebraic notation and Elm. Instead of using subscripts to differentiate different values  $m_1, m_2, \dots$ , we use a function `m` depending on an input `n`. The next difference is that we capture all of the substitution rules together using a `case` expression. Finally, unlike in algebra, multiplication requires a symbol,  $*$ , and Elm will not accept a definition like `(x+a)(x+b)` in which writing expressions next to each other indicates multiplication. Some programming languages do allow this, including TeX, the language in which this book is written. But most, including Elm, do not.

At this point, we have pointed out the differences, but there are more similarities than differences. For example, parentheses change the order of operations. The main difference between how you learned them is that we defined our own functions in Elm on the first day—not after years of school! Maybe it wouldn't have taken as long if we had Elm functions to play with, for example, to draw our  $m_n$ s:



---

Elm can be pretty useful for this type of visualization, and you can experiment<sup>1</sup> with your own functions. It is especially helpful before rushing a new algorithm into production to know that the number of steps grows exponentially like this—just in case your boss asks why you’re playing around.

To solve the full problem, we will use the same Divide and Conquer strategy, so let’s spell out the strategy explicitly—we need to take three steps:

- Divide: Within the original problem, identify smaller subproblems of the same type, unless it is small enough to solve immediately.
- Conquer: Solve the subproblems using the same strategy.
- Combine: Combine the solutions to the subproblems into a solution to the original problem.

Sounds simple, but pay close attention to the requirement that the subproblems be of the same type and be smaller, so as not to subdivide infinitely. In our first example, the problem was to count the number of moves required to move a tower of size  $n$ . All of the problems involve moving a tower, only the size changes, and the towers are readily ranked by size.

Here’s an example which fails: Mow a field by dividing it into two equal pieces and getting two friends to each mow half. When they are finished, the field is mowed. The problem: mowing a piece of a field is always the same problem, and if your friends are as lazy as you, they will follow your lead and get two friends to mow halves of their halves. The problem is that there is no smallest size of field, so nobody would ever get around to doing any mowing. If we slightly change the problem, and start with a field of 32 hectares, and in the divide step we specify that when the patch to be mowed is 1 hectare, the unlucky friend of a friend will have to go buy a goat and supervise it to ensure even munching, then the Divide and Conquer strategy would work.

Here’s another way it could fail: You sneak into the kitchen and steal a gingerbread person, but after escaping detection for a day, you start to worry about potential incarceration, and want to dispose of the body before any detectives come snooping. So you get some friends and break off the arms, legs, and head, and give each friend a piece of anatomy to disappear. It may be the perfect crime, but it is not Divide and Conquer, because the subproblems (arms, legs, head, torso) are not the same as the whole person. You cannot break a leg off an arm, nor an arm off a head. So what we have is a strategy which disposes of one body, but it won’t hide criminality at a large scale, involving, say, boxes of gingerbread people.

Back to the Tower of size  $n$ :

- Divide: The top  $n - 1$  blocks are a tower in their own right, but smaller by one. A tower with 1 block cannot be divided, but it can be moved in one step.
- Conquer: Calculate the number of steps to move the tower of size  $n - 1$ , call it  $m_{n-1}$ .
- Combine: To move the tower of size  $n$ , we need to move the smaller tower, move the base and move the smaller tower again, so we need  $m_{n-1} + 1 + m_{n-1}$  steps.

What about moving the tower? You may have devised your own notation for moves, but some notation is required to avoid writing tortuously long sentences in English.

---

<sup>1</sup>Steps: <https://cs1xd3.online>ShowModulePublish?modulePublishId=f4b3adcd-7c00-47dc-8693-611975893cb2>

Move Tower of size  $n$  from  $x$  to  $y$  using additional place  $z$ .

Divide: If the tower has size 1, move that piece from  $x$  to  $y$ , which we will write as  $(x,y)$ , otherwise the top  $n - 1$  blocks are a smaller tower.

Conquer: Apply this procedure to produce a sequence of steps for moving the top tower from  $x$  to  $z$  using  $y$ . Call that sequence  $s(x,z,y)$ , and apply it again to find a sequence  $s(z,y,x)$  to move a tower of size  $n - 1$  from  $z$  to  $y$  using  $x$ .

Combine: Join together the two sequences with one additional move:  $s(x,z,y)$  followed by  $(x,y)$  followed by  $s(z,y,x)$ .

To write this in Elm, we need to define our set of places, which is a distinct type of thing, apart from numbers, letters, words, buffalos, and every other thing we need in our program. We define a new type of thing like this:

```
type Place = A | B | C
```

For two things together, we can use a pair, ie  $(A,B)$ , in the same way that we use  $(7,3)$  for Cartesian coordinates on the plane to group together the horizontal and vertical positions. Since we need many moves, we can express our Tower programs as `List (Place,Place)`. Note that not all programs make sense. The program `[(A,A)]` doesn't do anything! And just because we write a command `(A,B)` doesn't guarantee that when we get to this point in the program, that there will be a block at position `A` or that the block at position `B` will be wider than the one we are moving.

But ignoring these issues, we can solve our tower problem:

```
moveTower n x y z =
  case n of
    1 ->      [(x,y)]
    otherwise -> (moveTower (n-1) x z y) ++ [(x,y)] ++ (moveTower<-
      (n-1) z y x)
```

Woah! That was a long chapter, all to explain three lines of Elm! This is typical of the Divide and Conquer strategy. Once you figure out how to divide up your problem, it is usually easy to write the code, especially in a language like Elm. The tricky part is that the divide step is not always obvious. Let's make sure we remember the three parts: in the code above, underline the parts which correspond to dividing. On the code below, underline the parts corresponding to conquering.

```
moveTower n x y z =
  case n of
    1 ->      [(x,y)]
    otherwise -> (moveTower (n-1) x z y) ++ [(x,y)] ++ (moveTower<-
      (n-1) z y x)
```

Now underline the parts corresponding to combining:

```
moveTower n x y z =
  case n of
    1 ->      [(x,y)]
    otherwise -> (moveTower (n-1) x z y) ++ [(x,y)] ++ (moveTower<-
      (n-1) z y x)
```

Now you get to play with it<sup>2</sup>!

## 7.1 Dividing and Conquering Map

[to contents](#)

We've previously seen how simple using `map` is. But how simple is it to define? Can we use Divide and Conquer? What are the subproblems? Try to write them out on your own.

Divide:

Conquer:

Combine:

How does your solution compare to ours?

Divide: Separate the first element from the rest of the input list. The rest of the list is still a list, so that's our subproblem. We can do this if the list is not empty.

Conquer: Apply this procedure to get a new list whose elements are the result of applying the function argument to the rest of the list.

Combine: Take the solution to the subproblem, which is a list, and prepend the result of applying the function argument to the first element.

When we implement this in Elm, we also need to take care of the base case, which here is when the list is empty, and so has no first element.

```
map fun list = case list of
    (x :: xs) -> (fun x) :: (map fun xs)
    []           -> []
```

<sup>2</sup>Hanoi Sim: <https://cs1xd3.online>ShowModulePublish?modulePublishId=c758ef3e-5e41-4c97-b392-70ed376abc91>



## 8. Composing Music in Elm

Because Elm compiles to JavaScript, it is possible to combine it with various JavaScript modules to create some interesting programs, from 3D scenes using WebGL, to much more. In this chapter, we focus on ElmMusic, a project that uses [WebAudioFont](#) to allow users to compose and play music, as well as add accompanying animations using GraphicSVG. ElmMusic is available on [cs1xd3.online](#).

Warning! Notes aren't splitting the beat  
Measure has not enough notes

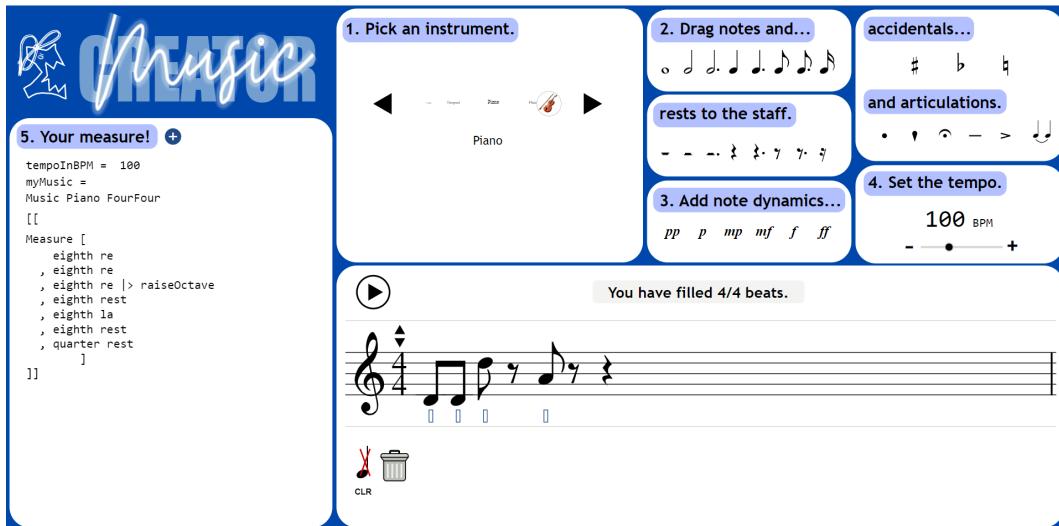


An example composition created using ElmMusic.

### 8.1 Music Creator & Basics

[to contents](#)

Much like GraphicSVG's Shape Creator, we also have a [Music Creator](#) for ElmMusic, which allows you to easily write individual measures for a composition using a drag-and-drop interface.



To use the Music Creator, you first select your desired instrument, then drag notes and rests to the staff as desired. Depending on the vertical position you drag notes to, it will set the pitch accordingly. When writing code, you can also use the `raiseOctave` and `lowerOctave` functions to adjust a note's octave. You can also chain these functions together using pipe operators to adjust pitch by multiple octaves at once. It is also worth noting that ElmMusic uses Solfège notation (do, re, mi, etc.) to represent pitch, but it is possible to write Elm functions to translate pitch between Solfège and your own custom letter notation, for example.

After adding notes and rests, you are able to add note dynamics, accidentals, and articulations to individual notes. Dynamics added to a note will carry forward to subsequent notes until another dynamic is specified. Accidentals will do the same, but only within the same measure. Articulations, however, will only apply to individual notes, so if you want several notes to be staccato, for instance, you will have to add the articulation separately to each note.

## 8.2 The Elm Music Slot

[to contents](#)

As mentioned before, ElmMusic can be accessed on [cs1xd3.online](https://cs1xd3.online), using the ElmMusic activity. When creating an ElmMusic module, you will be presented with code that produces a standard C Major scale if you compile the module. The following section will go through and explain what the sample code means.

### 8.2.1 Validation & Tempo

Near the top of the code, you will see three definitions grouped together: `ifValidateBeaming`, `ifValidateTimesignature` (sic), and `tempoInBPM`.

The first two are boolean values; the former enables validation for beaming and grouping rules (music has rules for how notes in a measure should be grouped, which depends on the strong and weak beats determined by the time signature you choose), while the latter

enables time signature validation, which will warn you if the number of beats in a measure doesn't match the time signature you chose.

The last definition simply determines the speed of the music, in beats per minute (so the default of 120 BPM means two beats are in each second).

### 8.2.2 myMusic

The next part of the sample code we will examine is

```
myMusic : ElmMusic.Music msg
myMusic =
    Music Marimba ThreeFour
        [ [ myMeasure
            [ quarter rest
            , half rest
            ]
        ]
    ]
```

The type `ElmMusic.Music` has one constructor, also called `ElmMusic.Music`, but imported so you can use it without the `ElmMusic.*` qualifier. It has three data types attached:

- `ElmMusic.Instrument` for the instrument which will play this music;
- `ElmMusic.TimeSignature` for the simple time signatures we support, i.e., the number of beats per measure on top, and denominator of the note worth one beat;
- `List (List (ElmMusic.Measure msg))` for the list of measures (notes between vertical bars).

The `Instrument` part of the `Music` type determines what instrument will be played when you compile your module and press the play button.

By default, the `Instrument` is set to `Piano`, but other instruments are available as well:

```
type Instrument
    = Piano
    | Recorder
    | Flute
    | Violin
    | Trumpet
    | Sitar
    | ElectricGuitar
    | Marimba
    | Bass
    | Cello
    | Tuba
    | Timpani
    | Custom Int Clef
```

The last choice is `Custom Int Clef`. This allows you to select a custom instrument from all of the ones supported by `WebAudioFont`. The `Int` parameter is the MIDI number which can be obtained from the `WebAudioFont` instrument catalogue, while the `Clef` parameter

can be set to either `TrebleClef` or `BassClef`. So, for example, if you wanted a String Ensemble as your “instrument,” that might correspond to an `Instrument` value of `Custom 517 TrebleClef`.

The `TimeSignature` part of the `Music` type determines how many beats are in a measure, and which type of note is worth one beat. The most commonly used time signature in music is  $4/4$  time, which corresponds to `FourFour` in the Elm code. There are a number of different time signatures supported by ElmMusic, but the ones that work the best are  $4/4$  and  $3/4$ , and the others should be considered experimental:

```
type TimeSignature
= FourFour
| ThreeFour
| ThreeEighths
| TwoFour
| FiveFour FiveFourGrouping
| SixFour -- 3 + 3
| SixEight
| FiveEight FiveEightGrouping
| ThreeTwo
| TwoTwo
| SevenEight SevenEightGrouping
| NineEight
| TwelveEighth -- 3+3+3+3, feels like 4-4 for stress, with ↪
  triplets
| OpenTwo -- half note gets a beat
| OpenFour -- quarter note has a beat (determines playback ↪
  and grouping)
| OpenEight -- eighth note gets a beat
```

The three open time signatures at the bottom have no limits on how many beats are in a measure; they merely define what type of note is worth one beat. You may also notice that some time signatures have additional parameters; these parameters determine which beats in a measure are stressed, which also affects how notes are grouped. The list of groupings is as follows:

```
type FiveFourGrouping
= FF23
| FF32

type SevenEightGrouping
= SE223
| SE232
| SE322

type FiveEightGrouping
= FE23
| FE32
```

Lastly, the `Music` type requires a nested list of `Measures`, where each `Measure` contains a list of notes as a type parameter. Both the provided sample code and the Music

Creator are good ways to learn how to write a measure in code, so we will not go into too much detail here. Some useful information to know, however, is that there are functions for notes from whole notes all the way to thirty-second (i.e. 32nd, not 30 seconds) notes, and you can also “dot” notes to increase their duration to 1.5 times the original.

### 8.2.3 GraphicSVG Definitions & Lyrics

At the very top of the sample code in an ElmMusic module, you will notice three peculiar functions: `twi`, `nkle`, and `emptyVideo`, each of which takes in a time. These are actually lists of GraphicSVG shapes that can be used to add animations to your music composition. `twi` and `nkle` (which combine into “twinkle”) are two sample animations provided for you to use, but you can also create your own; just keep in mind that any custom animations you create should accept a parameter for time—not `model.time`—even if you don’t use it for anything.

Animations can be added to your music using the `addVideo` function on a note in one of your measures (e.g. `quarter do |> addVideo twi`). Once an animation is added, it will continue to play until the `addVideo` function is used again (so if you want a section without any animations playing, you can add `addVideo emptyVideo` to the desired note).

You can also add lyrics to your music using the `sing` function, e.g.

```
quarter do |> sing "text"
```

If you haven’t added lyrics to music before. It is best to attach one syllable to each note, and use dashes to split words on syllables.

```
quarter do |> sing "wa-"  
quarter do |> sing "ter-"  
quarter do |> sing "mel-"  
quarter do |> sing "lon"
```

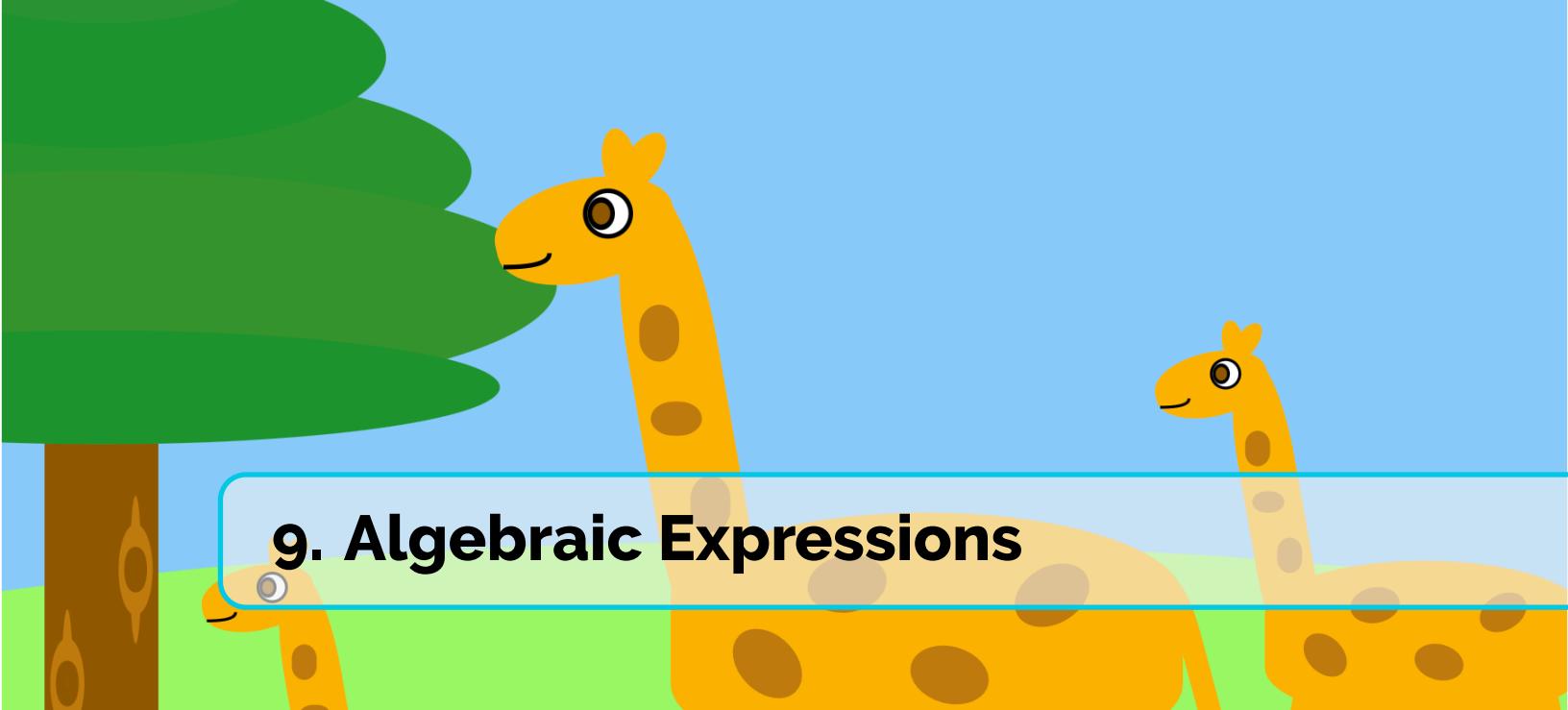
## 8.3 Next Steps

[to contents](#)

At this point, you should be ready to create simple compositions using ElmMusic. To create more advanced compositions, you can combine your music knowledge with Elm techniques to help you when composing a piece. For example, since the `Music` type is essentially a list of measures with some additional parameters, and the `Measure` type is just a list of notes, you can use list functions such as `List.map` or `List.repeat` to save some time when composing, for example using `map` to apply a desired articulation to many notes at a time, or using `repeat` to avoid having to copy and paste several identical measures.

There are lots of experiments you can do once you start writing your own functions. You could write a function to add new articulation to an input measure, and then string them together into an alien rhythm using `List.concatMap`. You could write a function to repeat a melody line, increasing the volume (dynamics) each time, or changing the pitch.

You can also use cs1xd3.online’s module sharing functionalities to create libraries (for instance, a library of functions to map pitch between Solfège and letter notation) that you can then import into other ElmMusic modules if desired.



## 9. Algebraic Expressions

Every language needs *nouns* and *verbs*. Things, and what happens to them. In Computer Science, we have data, and functions.

In this module, we define a language of the types of algebraic expressions used in Calculus.

```
type Exprs = Const Float
| Var String          -- in math we often use Char
| Sqrt Exprs          -- we can take sqrt(anything)
| IntPow Exprs Int   -- the easy case of exponent
| Exp Exprs           -- e^expr
| Ln Exprs
| Mult Exprs Exprs
| Add Exprs Exprs
| Neg Exprs
```

To save ourselves some headaches later, we do not allow expressions like  $x^y$ , but restrict our expressions to integer powers  $x^n$  where  $n$  is an integer. In mathematics, we usually use a single letter for a variable, but then we also end up using subscripts, superscripts, Greek and other alphabets and accents. Since Elm **Strings** are made up of Unicode characters, you can pretty much do any of those things, as long as you can remember the keyboard tricks to type them all in. I'll stick to English words, contracted and sometimes squished together. We also need the **Var** constructor, which signals that this particular **String** is the name of a variable. In pen-and-paper mathematics, we would never write something like that. The same symbol can mean different things in different contexts. We just need to remember which type of math we are doing. This is even more clear for constants, where 0 could be a number zero, but it could also be a function which is always zero, or a vector which is zero.

Since this data type refers to itself in some of its constructors, it is a *recursive data type*. This means that the size of values is unbounded, since each value is a tree structure.

Because we have alternative constructors, and multiple types associated with construc-

tors, this data type also has both *sums* and *products* of types. We call this an *algebraic data type*. In some programming languages, this is harder to do, but it is so easy to do in Elm that it hardly gets mentioned. This is the data-structure. It is, however, commonly called an *algebraic* data type. Either way, is the mirror-image of the Divide & Conquer algorithms and you can expect to see them here.

To make it easier to create examples, we predefine some variables

```
x = Var "x"
y = Var "y"
z = Var "z"
```

so you can refer to them without needing to repeatedly type `Var`. For example

```
xxy = Mult x y
```

You can now type in some of your favourite expressions that bring back fond memories of High School math class. Here are some of our favourites:

```
example = Add (Const 7) (Mult (Const 4) (Add (Sqrt x) (Exp y)))
example2 = Add (Var "x") (Var "y")
example3 = Mult (Var "z") (Const 0)
example4 = Add (Sqrt (Mult (Var "x") (Const 0))) (Var "y")
example5 = (Sqrt (Const 0))
example4Again = Add (example5) (Var "y")
```

Once you've typed those in, I bet you have a hard time reading them. The `Exprs` language is just not as easy to read as the mathematical expressions we are used to, but we can partially fix that by adding a function to translate `Exprs` into a string of characters which look a lot like pencil-and-paper expressions.

Expressions can get long and complicated, but we can always break them up into similar pieces, and using Divide and Conquer means that we only have to figure out the simplest possible expressions, and then let the function do all the work:

```
pretty : Exprs -> String
pretty e = case e of
    (Var name) -> name
    (Mult expr1 expr2) -> "(" ++ (pretty expr1) ++ ")" ++ " * " ++ "(" ++ (pretty expr2) ++ ")"
    (Sqrt expr) -> "sqrt(" ++ (pretty expr) ++ ")"
    (Const c) -> show c
    (IntPow expr exponent) -> "(" ++ (pretty expr) ++ ")^(" ++ (show exponent) ++ ")"
    (Exp expr) -> "e^(++ (pretty expr) ++)"
    (Ln expr) -> "ln(" ++ pretty expr ++ ")"
    (Add expr1 expr2) -> (pretty expr1) ++ " + " ++ (pretty expr2)
    (Neg expr) -> "- (" ++ pretty expr ++ ")"
```

## 9.1 Simplification

[to contents](#)

```
simplify : Exprs -> Exprs
```

$x \cdot 0 \mapsto 0$  (commutative)

```
simplify e = case e of
    (Mult x (Const 0)) -> Const 0
    (Mult (Const 0) x) -> Const 0
```

$x^1 \mapsto x$

```
(IntPow x 1) -> x
```

$x \cdot x \mapsto x^2$  Since we cannot match the same value twice in the Divide step, we have to match  $x^y$  and add a condition that they be equal ( $x == y$ ).

```
(Mult x y) -> case x == y of
    True -> IntPow x 2
    False -> Mult (simplify x) (simplify y)
```

$x^n * x \mapsto x^{n+1}$  is complicated, because we cannot use  $x$  twice on the LHS

```
(Mult (IntPow x n) y)
-> case x == y of
    True -> IntPow x (n+1)
    False -> Mult (IntPow (simplify x) n) (simplify y)
```

$x + 0 \mapsto x$  (commutative)

```
(Add x (Const 0)) -> x
(Add (Const 0) x) -> x
```

$x * 1 \mapsto x$  (commutative)

```
(Mult x (Const 1)) -> x
(Mult (Const 1) x) -> x
```

$\sqrt{x^{2n}} \mapsto x^n$

$\sqrt{x^{2n+1}} \mapsto x^n \sqrt{x}$

```
(Sqrt (IntPow x exponent))
-> case exponent % 2 of -- remainder
    0 -> IntPow x (exponent // 2)
    _ -> Mult (IntPow x (exponent // 2)) (Sqrt (simplify x))
```

In the next rules, we check to see if an operation is applied to a constant, in which case we can calculate the result right away, because there are no variables involved.  $\sqrt{Const c} \mapsto Const \sqrt{c}$

```
(Sqrt (Const c)) -> Const (sqrt c)
```

The same works for the other operations

```
(Add (Const c1) (Const c2)) -> Const (c1 + c2)
(IntPow (Const c) n) -> Const (c^n)
(Exp (Const c)) -> Const (e^c)
(Ln (Const c)) -> Const (logBase e c)
(Neg (Const c)) -> Const (-c)
```

$\text{NaN} + x \mapsto \text{NaN}$  (this one is tricky)

```
(Add (Const x) y)
-> case isNaN x of
    True -> Const x
    False -> Add (Const x) (simplify y)
```

If none of the above patterns is matched, we have to look deeper in the expression tree to find something to simplify. This is the *Divide* step! Since there are multiple data constructors which contain subexpressions, we need multiple rules to implement the Divide step. All of the rules are similar to  $e_1 * e_2 \mapsto \text{simplify}(e_1) * \text{simplify}(e_2)$

```
(Mult expr1 expr2) -> Mult (simplify expr1) (simplify expr2)
```

but each handles a different outermost operation:

```
(Add expr1 expr2) -> Add (simplify expr1) (simplify expr2)
(Sqrt expr1) -> Sqrt (simplify expr1)
(IntPow expr1 n) -> IntPow (simplify expr1) n
(Exp expr1) -> Exp (simplify expr1)
(Ln expr1) -> Ln (simplify expr1)
(Neg expr1) -> Neg (simplify expr1)
(Const c) -> Const c
(Var name) -> Var name
```

Now each time we apply the `simplify` function, it searches the expression tree for a pattern for which it has a rule, and applies that rule. Most expressions can be simplified by more than one rule, and sometimes the second pattern we simplify is partially created by applying the first rule. We need a way to keep applying rules until there are no more rules which apply. The easy way to do this is to apply `simplify` and check to see if the new expression is the same

```
simplifyUntilItStops x = let xNew = simplify x
    in case xNew == x || isConstNaN x of
        True -> x
        False -> simplify xNew
```

where we use a helper function which checks for a NaN input, since NaN is defined to be not equal to anything, including itself!

```
isConstNaN x = case x of
    (Const x) -> isNaN x
    y           -> False
```

## 9.2 Derivatives

[to contents](#)

Derivation is an operation which turns an expression defining a continuous function and produces a new expression defining the slope of the tangent to the function at each point. We call this the derivative. In Elm, everything we do is a function, and we can translate the last sentence into this type

```
deriv : String {- variable with respect to which we take the ←
  derivative -}
  -> Exprs
  -> Exprs
```

adding in the fact that we take the derivative with respect to a particular variable (which is defined by a string)–this is something we may expect a person to assume.

We define the function by encoding all the basic rules about derivatives one case at a time. If the function is constant, of course, the slope is zero:

```
deriv varName e
= case e of
  (Const c) -> Const 0
```

A variable, on the other hand, has derivative 1 if it is the right variable:

```
(Var name) -> case varName == name of
  True -> Const 1
  False -> Const 0
```

and the derivative of a sum is the sum of the derivatives:

```
(Add e1 e2) -> Add (deriv varName e1) (deriv varName e2)
```

Did you notice Divide & Conquer being applied in the last rule? The remaining rules will also implement D & C, but the combine steps get more complicated.

Let's start with the product rule:

```
(Mult e1 e2) -> Add (Mult (deriv varName e1) e2)
  (Mult e1 (deriv varName e2))
```

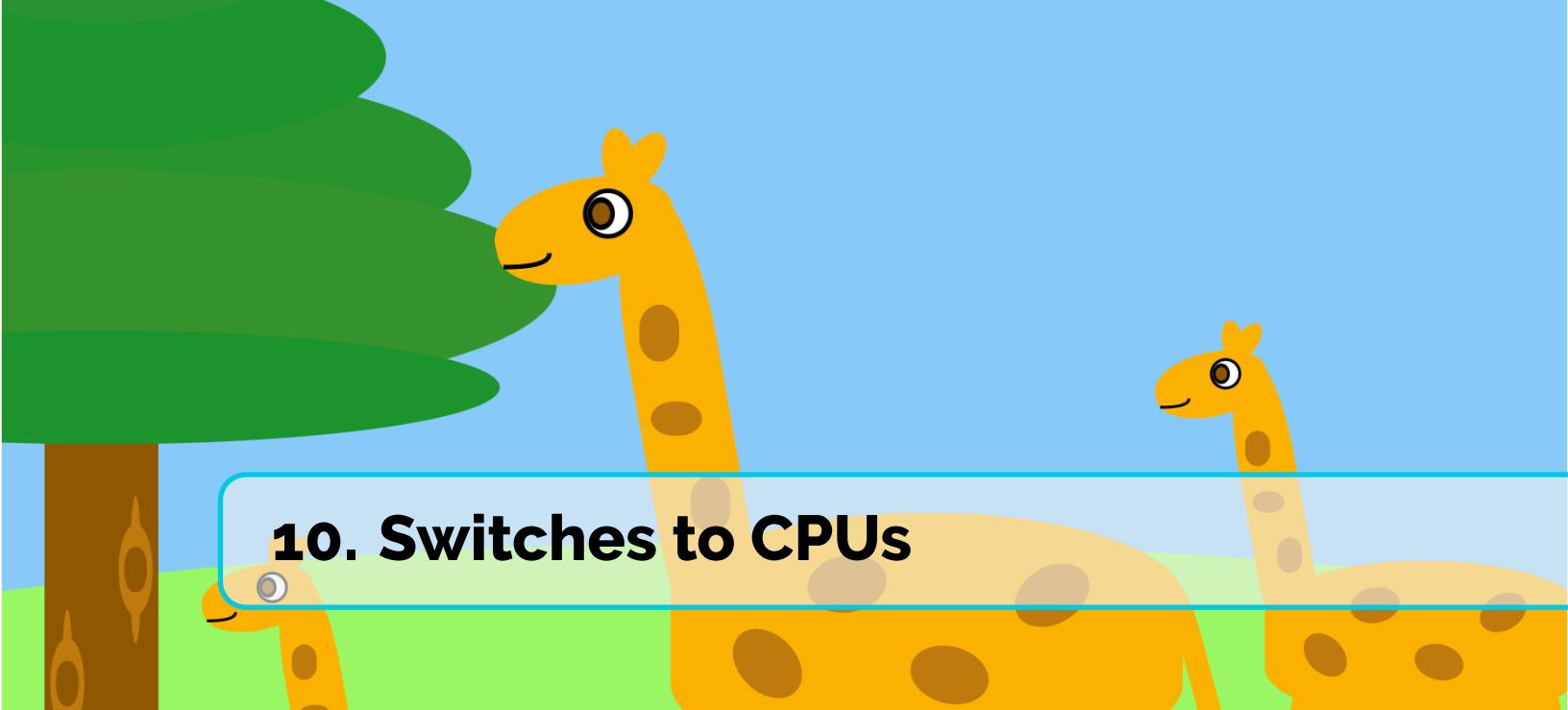
The remaining cases are rules which are familiar from Calculus, and some of them we call the Chain Rule.

```
(Neg e1) -> Neg (deriv varName e1)
(Sqrt e1) -> Mult (Const 0.5) (Mult (IntPow (Sqrt e1) (-1))
  (deriv varName e1))
)
```

where, to see the application of the chain rule, we could write  $f(x)$  for `e1` and  $f'(x)$  for `deriv varName e1`:

$$\frac{\partial}{\partial x} \sqrt{f(x)} = \frac{1}{2} \frac{1}{\sqrt{f(x)}} f'(x).$$

```
(IntPow e1 n) -> Mult (Mult (Const (toFloat n)) (IntPow e1↔
  (n-1)))                                (deriv varName e1)
(Exp e1) -> Mult (deriv varName e1) (Exp e1)
(Ln e1) -> Mult (IntPow e1 (-1))
  (deriv varName e1)
```



## 10. Switches to CPUs

*Do you remember a time when your parents thought you were sleeping, but you were really making secret plans via flashlight signals with your accomplice? You were encoding information by turning the flashlight ON and OFF (or maybe by using long and short flashes).*

Inside every iPad or smartphone is a computer crammed with billions of wires, tiny versions of the wires connecting a flashlight's battery and light, and all the pictures, texts, and numbers you put into that device are encoded into groups of *ON* and *OFF* states.

Every spelling mistake you fix or sketch you make happens by flipping switches.

How can we do all that with *ON* and *OFF*? Well, first of all, we have billions of switches in here, and we are very organized!

To start, let's agree that *ON* will mean 1 and *OFF* 0. Just as with normal decimal numbers, we can group these bits (short for binary digits, *bi* meaning two) to make bigger numbers. Counting in binary:

decimal	0	1	2	3	4	5	6	...
binary	0	1	10	11	100	101	110	...

You get the idea!

Starting with something as simple as *on/off*, we have built up a system of numbers. Letters are easy, just number them 1 for A, 2 for B, and so on, and we can encode text. If you write fancy text, you will need Unicode<sup>1</sup>, which is an agreed encoding of any symbol you are likely to need—even symbols for the undeciphered Linear A script of Minoa (ancient Greece), and mediaeval alchemical symbols. You can put them in `Strings` and access them using `Char.fromCode`.

Images are also easy to encode. Chop them into little squares, and encode the intensity of the primary colours in each square as a number. If you really are reading this on an iPad, you can try this out with our *free* app Image 2 Bits, for black = 1 and white = 0 images.

---

<sup>1</sup><http://www.unicode.org>

One thing I cannot explain is why all of our technology based on *binary* encoding is called *digital*<sup>2</sup>!

## 10.1 CPU

[to contents](#)

So encoding information as on/off choices is not hard, but how do we process that information? While decoding images and text can be an interesting mental exercise, the real advantage of the information revolution is not just the better storage of information, but the ability of computers to process it for us. Think about it, even looking something up on the internet involves many computers processing that information. The computer at the source needs to separate the information you want from all the other information, then it needs to find a path through the web of the internet to send that information back to your computer, then your computer needs to make the answer visible or audible to you. All so you can ask the internet “Are there really butterflies in my stomach?”

Today, people who own a smartphone carry computers with billions of switches around in their pockets! How do you organize a billion switches? Well, we follow the Divide and Conquer principle. Break the process down into successively smaller chunks, solve small problems, and then assemble those solutions together. A large chunk of those switches are organized to store information. That information is organized into bits, 8-bit integers called bytes, one or more bytes to store **Chars**, multiple **Chars** to store **Strings**, 64-bit groups to store **FLOATS**, and so on.

Wires transmit electric currents to transfer this data from one part of the processor to another. Specialized circuits of switches perform basic operations like adding, subtracting and multiplying **Ints** and more complicated circuits to do the same for **FLOATS**. These assemblies are called Arithmetic Logic Units or ALUs, Together with storage called *registers*, and a control system to decide which operations to perform on which data, the total is called a Central Processing Unit or CPU.

Let’s try to model this in Elm. Early CPUs only supported integer values, so let’s follow their lead, and define:

```
type alias RegisterValue = Int
```

CPUs today may have hundreds of registers, but for a long time, we could get by with a modest number. Our CPU will be a throw-back to a time with eight registers. Since everything the CPU explicitly “remembers” is in those registers, this is the largest part of the state of the CPU:

```
type CPUState = CPUState ( RegisterValue, RegisterValue
                           , RegisterValue, RegisterValue
                           , RegisterValue, RegisterValue
                           , RegisterValue, RegisterValue)
```

But we know that a CPU must execute a program to do anything interesting, and in the process of execution, it must implicitly remember its position in the program:

---

<sup>2</sup>Wikipedia blames it on the mathematician George Stibitz.

### CurrentInstruction

which must be reduced to switches—something best done by encoding the position as an `Int`.

To allow the program to execute instructions conditionally, so that we can implement `if-then-else` and `case` expressions, we also need to remember a condition. A single `Bool` value would suffice, but CPU designers have found it worthwhile to store the equivalent of `Order`:

### ComparisonResult

And, finally, assuming there is a `finally` for our program, we need

### (Maybe HaltedBecause)

to indicate whether the CPU has reached the end of the program, stopped because there was an error,

```
type HaltedBecause = ReachedHalt
| IllegalRegisterNum
| IllegalAddress
| IllegalInstrAddress
```

or, in the case of `Nothing`, is still running our program.

Our processor will only make simple numerical comparisons, which we can model with Elm's `Order` type

```
type alias ComparisonResult = Order
```

When a CPU adds a number, it is calculating a function. Since Elm is a functional language, it is easy to create values of type function. Each basic-operation function would have the type

```
basicOp : CPUSState -> CPUSState
```

Being able to do this actually requires some sophisticated programming in the Elm compiler. At the hardware level in the CPU, we must content ourselves with simpler functions, which we call *instructions*, and to store a program in a workable format, we need a simple encoding of those instructions as 32-bit or 64-bit integers.

We can simulate this in Elm with a

```
type Instruction
```

With the following instructions:

```
= Load RegisterNumber RegisterNumber RegisterNumber
```

is an instruction to move a value from main memory to a register. By convention, the first listed register number is the destination, and the address is the sum of the second two register values. If the address does not exist, the processor must halt with an error.

```
| Store RegisterNumber RegisterNumber RegisterNumber
```

is an instruction to move a value from a register to main memory. By convention, the first listed register number is the source register, and the destination address is the sum of the second two register values. If the address does not exist, the processor must halt with an error.

	<b>LoadImmediate</b>	<b>RegisterNumber</b>	<b>RegisterValue</b>
--	----------------------	-----------------------	----------------------

Sometimes, a value is always the same, so we don't need to bother putting it in main memory, we have this special instruction to load the value contained in the instruction.

	<b>Add</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>
--	------------	-----------------------	-----------------------	-----------------------

This is the first instruction which calculates a new value, in this case, by adding two register values. By convention, the first listed register number is the destination, and the values in the second two registers are added.

	<b>Multiply</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>
--	-----------------	-----------------------	-----------------------	-----------------------

Similarly, the first listed register number is the destination, and the values in the second two registers are multiplied.

	<b>And</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>
	<b>Or</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>

As with **Add** and **Multiply**, **And** and **Or** take two inputs and put an output back in the first register, but instead of treating the bits in the register as a number, they treat them as a bunch of bits and perform logical operations on each of the bits. For example

$$\begin{array}{r} & 0 & 0 & 1 & 1 \\ \text{And} & 0 & 1 & 0 & 1 \\ \hline = & 0 & 0 & 0 & 1 \end{array} \quad \text{and} \quad \begin{array}{r} & 0 & 0 & 1 & 1 \\ \text{Or} & 0 & 1 & 0 & 1 \\ \hline = & 0 & 1 & 1 & 1 \end{array}$$

shows the results of **Anding** and **Oring** 4-bit numbers, although our **RegisterValues** are 32-bits wide.

	<b>Not</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>
--	------------	-----------------------	-----------------------

**Not** reverses the bits.

$$\begin{array}{r} \text{Not} & 0 & 1 & 0 & 1 \\ \hline = & 1 & 0 & 1 & 0 \end{array}$$

	<b>Rotate</b>	<b>RegisterNumber</b>	<b>RegisterNumber</b>	<b>Int</b>
--	---------------	-----------------------	-----------------------	------------

while **Rotate** spins them around to the right,

$$00001000 \xrightarrow{\text{rotate } 1} 00000100, \quad 11000000 \xrightarrow{\text{rotate } 6} 00000011, \quad 00000001 \xrightarrow{\text{rotate } 1} 10000000,$$

putting them back in on the left when they fall off the end. (Again shown with shorter 8-bit numbers rather than the normal 32-bits.)

Doing different things depending on input data, or different user actions is accomplished at the CPU level by being able to compare numbers and then jump to a different instruction in the machine program rather than continuing, only if the conditions are true.

Compare	RegisterNumber RegisterNumber
---------	-------------------------------

does the comparisons of two register values, and puts the result of the compare into the **CPUState**.

Branch	(List ComparisonResult) RegisterNumber
--------	--

optionally jumps to an instruction number given by the specified register if the current condition in the **CPUState** is in the **List of ComparisonResults**.

And finally, when our program is really finished, we can save electricity and turn our CPU off with the **Halt** instruction.

Halt
------

For safety, there should always be a **Halt** at the end of the list of instructions, even if the CPU will never get there because of **Branches**. In particular, if all the conditions are listed, the CPU will always jump to the indicated location in the program!

## 10.2 Traces

[to contents](#)

Just like an accident investigator for the Transportation Safety Board, if something goes wrong with a machine program, we need to retrace the steps which led to the illegal instruction or incorrect result. For the simplified CPU we are studying, all we need to know are the initial CPU state and memory state and the program, because our CPU is *deterministic*—it will always arrive at the same result when starting with the same inputs<sup>3</sup>.

Given the program,

```
number instruction
(0, LoadImmediate 7 2),
(1, LoadImmediate 2 1),
(2, LoadImmediate 3 3),
(3, LoadImmediate 4 (-1)),
(4, LoadImmediate 5 16),
(5, LoadImmediate 6 6),
(6, Multiply 2 7 2),
(7, Compare 2 5),
(8, Branch [LT] 6),
(9, Halt)
```

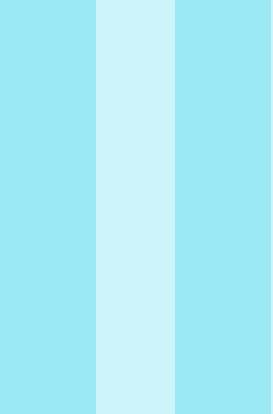
we can write out the state of the CPU at each instruction, and obtain a *program trace*:

---

<sup>3</sup>If you are an aficionado of computer games, you may be puzzled by the apparent random events which happen in your games! Even on deterministic CPUs, we can fake randomness by using some data which changes as an input. For a game, the start time of the game, measured in seconds from 1970, is generally random enough, but for more serious matters, like choosing random seeds to encrypt our secret messages, most CPUs have even better ways of generating random numbers.

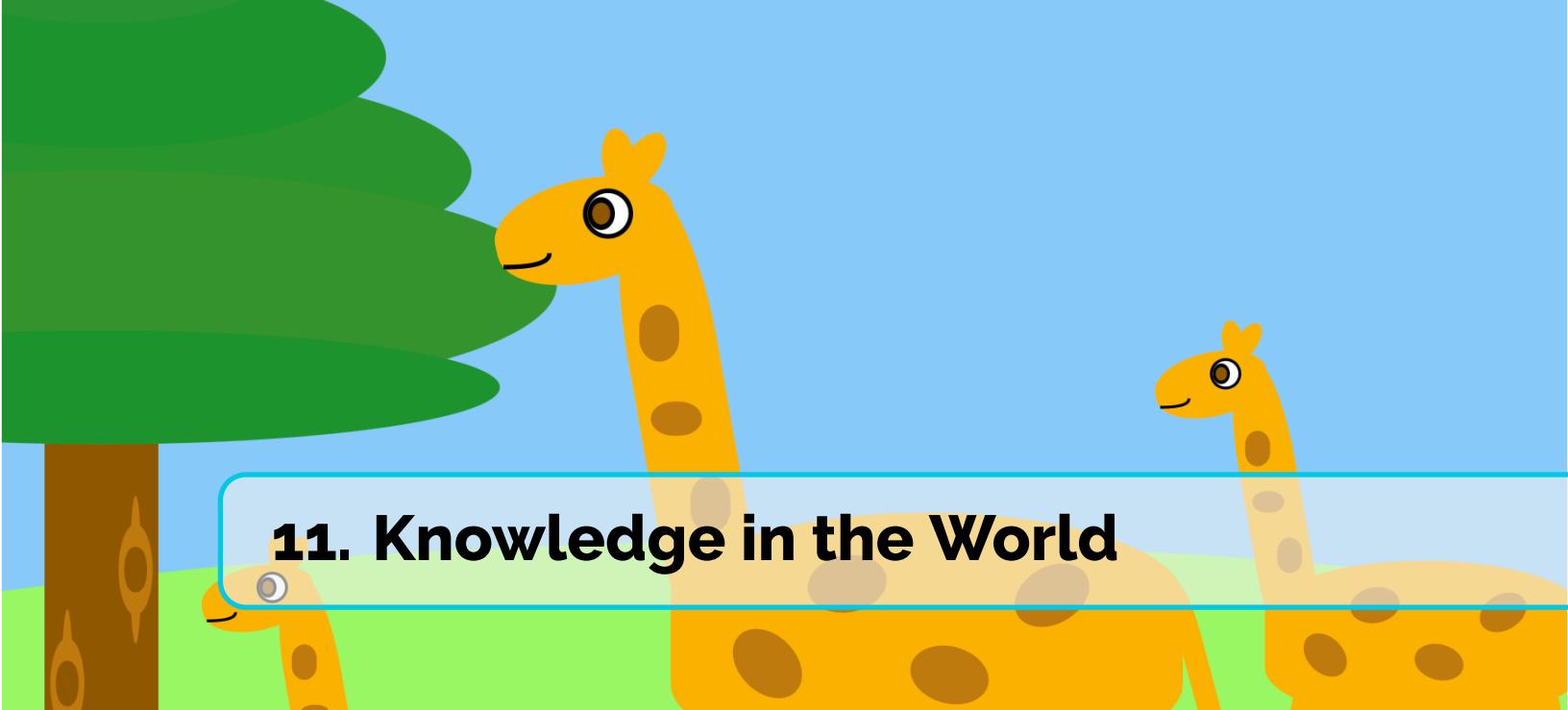
clock time	current instructn	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Condition
1	0	0	0	0	0	0	0	0	0	EQ
2	1	0	0	0	0	0	0	2	0	EQ
3	2	0	1	0	0	0	0	2	0	EQ
4	3	0	1	3	0	0	0	2	0	EQ
5	4	0	1	3	-1	0	0	2	0	EQ
6	5	0	1	3	-1	16	0	2	0	EQ
7	6	0	1	3	-1	16	6	2	0	EQ
8	7	0	2	3	-1	16	6	2	0	EQ
9	8	0	2	3	-1	16	6	2	0	LT $\overline{10}$
10	6	0	2	3	-1	16	6	2	0	LT
11	7	0	4	3	-1	16	6	2	0	LT
12	8	0	4	3	-1	16	6	2	0	LT
13	6	0	4	3	-1	16	6	2	0	LT
14	7	0	8	3	-1	16	6	2	0	LT
15	8	0	8	3	-1	16	6	2	0	LT
16	6	0	8	3	-1	16	6	2	0	LT
17	7	0	16	3	-1	16	6	2	0	LT
18	8	0	16	3	-1	16	6	2	0	EQ
19	9	0	16	3	-1	16	6	2	0	EQ

Creating a trace is a good way of checking your intuition about what a program does. It is also recommended, along with two sudokus and ten servings of vegetables and fruit as part of your daily diet, and is a favourite question to put on tests and exams, because it really tests your knowledge of the CPU state and transition functions—and it takes a while to fill out, giving the teacher time for a nap.



# Norman's Principles

<b>11</b>	<b>Knowledge in the World .....</b>	<b>182</b>
<b>12</b>	<b>The Principles .....</b>	<b>186</b>



## 11. Knowledge in the World

Everyone knows that “the customer is always right.” And as a corollary, if software doesn’t do what the customer needs, it is our fault. So we better do what the customer wants. Right? Well, it’s a bit more complicated, because, unlike with physical products like hoes or horseradish, the customer usually doesn’t know what they want software to do because they haven’t seen it before. You haven’t invented it yet!

Even if you’d never heard of “human-centred design”, you would probably be in favour of it, but what is it? The International Standards Organization have a standard for it (ISO 9241-210:2019(E)). It is advertised as

This document provides an overview of human-centred design activities. It does not provide detailed coverage of the methods and techniques required for human-centred design, nor does it address health or safety aspects in detail.

Translated into English, this means they are not there to help you do your job, they are there to tell your manager how to tell you aren’t doing it.

This may sound hopeless, but fortunately, there are people with actual tips you can put to use. The most important person in this field is Don Norman. When you have time, it is well worth reading his book “The Design of Everyday Things.” He is an engineer, a cognitive scientist, and an expert on design. He came up with much of the vocabulary we use to talk about design, and he grounded it in cognitive science. What is that? Cognitive Science<sup>1</sup> is the study of intelligence and behaviour. Suppress your immediate objection to the lack of definition for “intelligence” and focus on behaviour—what people do—and work backwards from there to why they do it. Or thinking glass half empty, why do people fail to use our software the way we think they should?

---

<sup>1</sup>If you want to know more about this Cognitive Science, I definitely recommend “Grasp: The Science Transforming How We Learn” [SY21], because you will be able to relate to the applications, since you are a learner.

## 11.1 Memory

[to contents](#)

The way we have designed our software puts demands on them beyond their abilities. What are these abilities? Actually, the important ones mirror the capabilities of a Central Processing Unit (CPU). In both human and electronic computers, we perform simple operations on data, and the access to that data limits the rate at which we can process it. By the time you get to university, you have a truly astounding amount of information in your head, but put you in a quiz show under the bright lights, and you can hardly remember any of it. In fact, a lot of the information stored in your head is stored as procedural memory, so you could only with a great deal of difficulty use it on the quiz show. The only way to retrieve it is to start doing the task related to that memory, and suddenly it is like riding a bike—perhaps literally—you can just do it again without being able to explain it.

It is easier to map tasks onto a computer program when we have learned them step-by-step. A prototypical example is long<sup>2</sup> multiplication.

$$\begin{array}{ccccccc}
 & a_5 & a_4 & a_3 & a_1 & a_0 \\
 & \times & b_4 & b_3 & b_2 & b_1 & b_0 \\
 & a_4 \times b_0 & a_3 \times b_0 & a_2 \times b_0 & a_1 \times b_0 & a_0 \times b_0 \\
 & + & + & + & + & + \\
 a_4 \times b_1 & a_3 \times b_1 & a_2 \times b_1 & a_1 \times b_1 & a_0 \times b_1 \\
 & + & + & + & + \\
 a_4 \times b_2 & a_3 \times b_2 & a_2 \times b_2 & a_1 \times b_2 & a_0 \times b_2 \\
 & + & + & + & + \\
 a_4 \times b_3 & a_3 \times b_3 & a_2 \times b_3 & a_1 \times b_3 & a_0 \times b_3 \\
 & + & + & + & + \\
 a_4 \times b_4 & a_3 \times b_4 & a_2 \times b_4 & a_1 \times b_4 & a_0 \times b_4
 \end{array}$$

Generations of children are taught to memorize single-digit multiplication facts, but not multi-digit multiplication. Instead, they have their first out-of-brain experience. Long multiplication is an algorithm designed to get around the limited working memory of the human brain by moving primary and intermediate results from brain to paper. Norman called this “putting knowledge in the world,” and he made it the organizing principle behind visual user interface design.

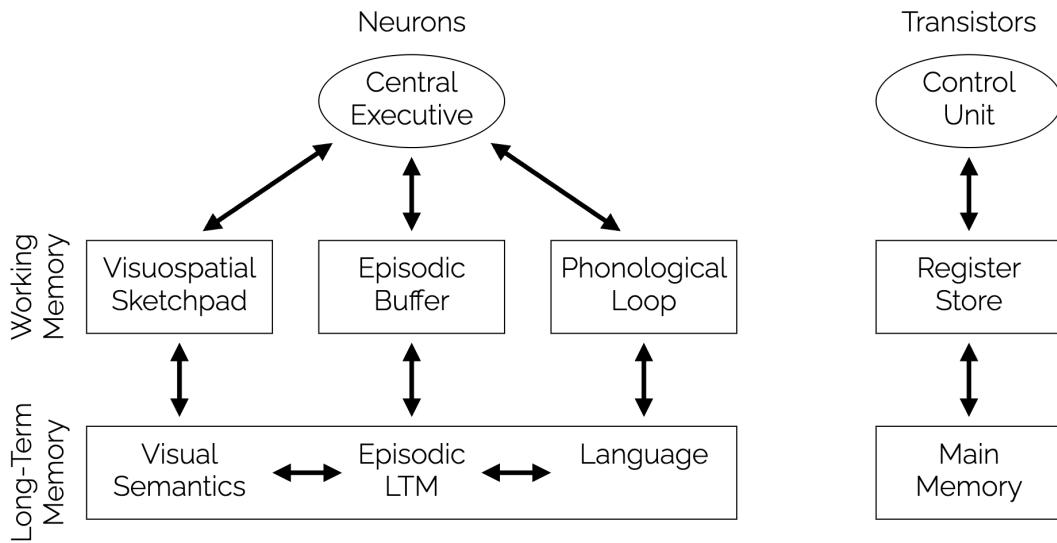
How bad is the limitation of working memory? Well, a human is doing well if they can keep seven things in their working memory, and how long you can keep it there depends partly on what it is. It is probably not an accident that that One Time Passwords seem to all be six digits long. If someone decides that six digits are still too easy to guess, they will have to group them, and make it easier to remember them in pieces. You might argue that phone numbers are now over seven digits, depending on the country. But mostly they aren’t really, because most people use very few country and area codes, so remember them as units. You could even argue that they become part of their procedural memory. That leaves us with seven numbers, and even there, there used to be patterns by town or village.

---

<sup>2</sup>Computer Scientists call it “schoolbook” multiplication when they use it as a basis for binary multiplication circuits, and large-integer software multiplication subroutines.

In 2001, Alan Baddeley was awarded for his work on working memory, and in “Is Working Memory Still Working?” he evaluates the progress<sup>3</sup>. Not surprisingly, he finds that it holds up pretty well. :) When Baddeley says that the theory holds up, he means that it continues to be useful in explaining the results of experiments, not that we can explain what is going on with the  $10^{14}$  synapses<sup>4</sup> in our brains.

The basic model which has held up, with a few improvements, can be summarized by this diagram on the left<sup>5</sup>:



And on the right, we've added a slightly more abstracted view of a computer. The parallel starts with the Central Executive, which decides what to focus on, and what to do with that information, and the Control Unit, which interprets the instructions of a program and controls which data goes where in the execution of those instructions. In the human brain, we don't know where the instructions are, or expect that they are stored in the simple and uniform way they are stored in a digital computer, so we don't show that aspect of the computer. Next, the Register Store varies between architectures, but typically stores 32 values, whereas the Phonological Loop stores about 7 words. To keep things in the Phonological Loop, we need to keep repeating them, or doing something with them<sup>6</sup>.

In both diagrams, this is where the bottleneck occurs. Try to do something too complicated, and we will fail. In the computer case, with its explicit machine instructions, we will not be able to write a program, because we won't be able to put the data in registers. In the brain case, we may be able to describe what we need to do, but will get muddled when we try to do it, because when we get to step 9 or 10, we just don't remember the things we need for that step, even if we just computed them or looked them up.

<sup>3</sup>Baddeley: [https://journals-scholarsportal-info.libaccess.lib.mcmaster.ca/pdf/0003066x/v56i0011/851\\_iwmsw.xml](https://journals-scholarsportal-info.libaccess.lib.mcmaster.ca/pdf/0003066x/v56i0011/851_iwmsw.xml) Popular science version: <https://www.psychmechanics.com/types-of-memory-in-psychology-explained/>

<sup>4</sup>Compare  $10^{14}$  to the (relatively) tiny number of pathways mapped in pictures of the connectome <https://en.wikipedia.org/wiki/Connectome>.

<sup>5</sup>make your own: <https://cs1xd3.online>ShowModulePublish?modulePublishId=8897cec7-2172-4dc1-8e4c-36ad9a6525d5>

<sup>6</sup>In digital memory, dynamic RAM (random-access memory) also needs to be refreshed, but we typically use faster, but more expensive, static memory for the register store.

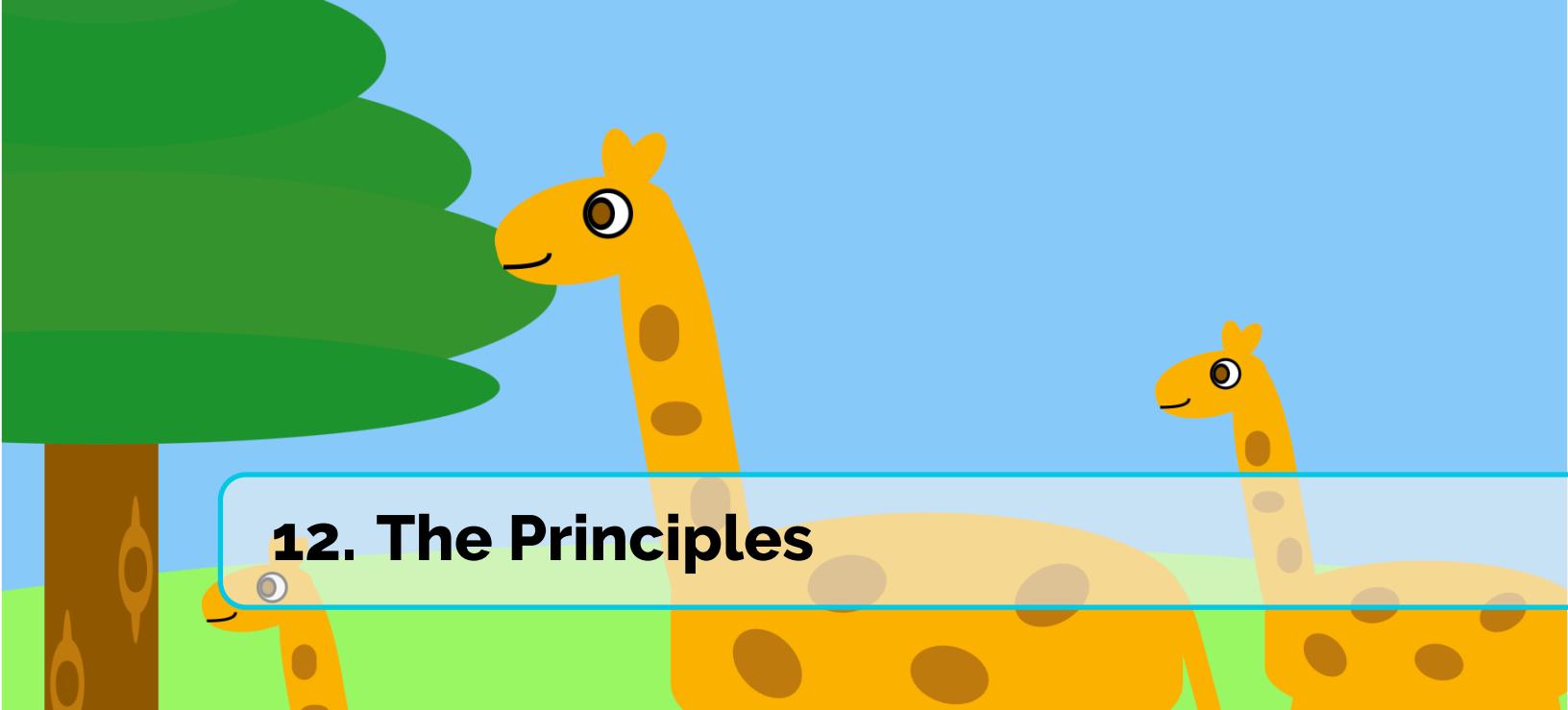
The easiest way to handle this bottleneck is to break the process down hierarchically, with “macro” steps composed of a manageable number of computational steps. We do not attempt to remember or store the data between macro steps, but either store it in Main Memory, or write it down on paper. You will notice that Paper does not appear in the diagram on the left. In theory, we could have committed data to Long-Term Memory between macro steps, but whereas storing data to Main Memory is expensive (typically taking 1000X longer to store and retrieve) it is reliable. Storing to human Long-Term Memory is not under our conscious control, and try as we might, we never<sup>7</sup> remember all of the things that we want to, so it is just a lot easier for us to write things down. If we use meaningful diagrams when we write things down on paper, we can also tap into our Long-Term Visual Semantics Memory. Your teachers have been tapping into this ability for a long time. When did you first start using a number line? multiplication table? As you progress in just about any field, you will learn new diagrams, whether they are Entity-Relationship Diagrams for databases or reaction diagrams in chemistry or predator-prey diagrams in ecology, you can pretty reliably separate experts from novices by their use of diagrams.

This is far from a complete picture of what we know about human cognition, or the complexity of a 30-billion transistor microprocessor, which employs all kinds of tricks to get around this basic bottleneck: we can get information fast, or we can get lots of information, but it is very hard to have both at the same time.

That was a pretty rough sketch, but hopefully enough to put “learn more about cognitive science” on your to-do list—another great example of putting “knowledge in the world” where it won’t get forgotten. For now, you just need to remember our limited working-memory capacity, and see how they largely explain Norman’s Principles.

---

<sup>7</sup>You can train yourself to be better at remembering things, just search for “memory palace” on the internet. Other than Sherlock Holmes, I don’t know anyone who uses it.



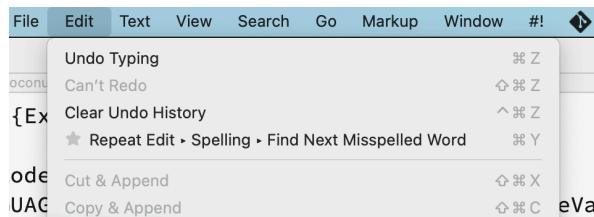
## 12. The Principles

Now that we understand who we are designing for—people with seriously small working memories—Norman’s principles are easy to explain.

### 12.1 Visibility

[to contents](#)

Make actions visible. Ideally, have a button or control for every possible action, and grey out the controls which are not active in the current state. For complex applications, you cannot do this, but you can arrange controls on tabs which slide in from the side, or pop over another control, or you can use menus, and the main menu will always be visible, reminding the user that there is a tree of submenus waiting to jog their memory, or help them discover additional actions.

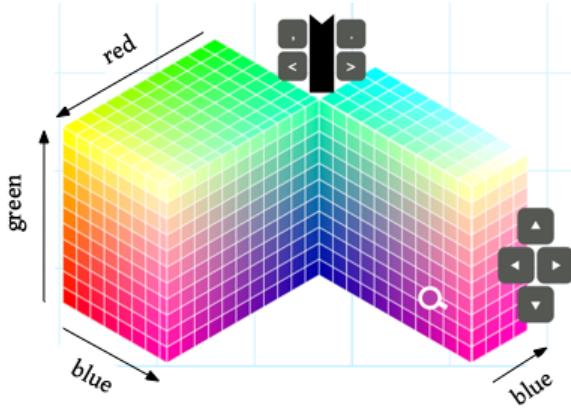


Some applications take the view that “power users” prefer keyboard shortcuts, but that brings us to...

### 12.2 Discoverability

[to contents](#)

Even power users start out having to learn the shortcuts, so you need to make possible actions discoverable. The best way to do this is to indicate keyboard shortcuts on menus, or by placing them in widgets, like

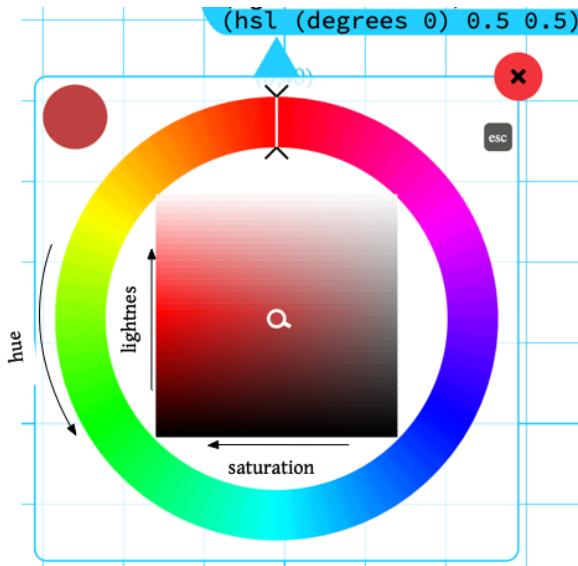


In this example, the arrangement of the keycaps is not accidental, it is an example of ...

## 12.3 Mapping

[to contents](#)

We can tap into the Spatial Semantics, making it easy to remember controls or figure out new controls in analogy with older controls or physical objects. In the case of the arrow keys, they are arranged according to the direction they point, and they are placed next to the plane on which the spyglass moves to select the colour. Both the internal arrangement and the positioning on the screen are examples of mapping. People expect similar things to be grouped together. That's how they are grouped in grocery stores, libraries, and that is even how you knew who were friends when you started school. The word "map" has many related meanings, and they all apply to "mapping". So there are "mind maps" of concepts, where proximity equals relationship. But we also use "map" to describe a mathematical function, and this allows for powerful visual mapping relationships. You probably don't even think about the fact that every function plot assumes a linear (or logarithmic, ...) relationship between the  $x$  or  $y$  coordinate on the page and the numerical values of the inputs and outputs of the function. If I had a favourite principle—which I don't since I'm a totally objective University Professor—it would be mapping. It is just a lot of fun to figure out new relationships and to use them to explain things better, whether to our users, or even to ourselves.



Since we included the RGB picker under discoverability, for balance, we need to include the HSL picker. This second picker includes two types of numerical mapping. We have an angular mapping for hue, and linear mappings for saturation and lightness. If you've never seen a double or triple rainbow, and didn't realize that the rainbow of colours are periodic, mapping hues to the angles of a circle closes the loop for you. :)

## 12.4 Signifiers

[to contents](#)

“Signify” means to be a sign of something. Your task as a designer is to make things look like what they do. I love this example from Apple:



which, thanks to Aesop, we all associate with the fable of the Tortoise and the Hare. There are lots of fast animals and lots of slow ones, but picking ones with strong associations works best, because those associations will be easy for your user to retrieve. Of course, these associations are culturally dependent, so you may not be able to use one set of signifiers for everyone. They are also generationally dependent, largely because many of the physical tools used by older generations have been replaced by software. On the other hand, newer generations recognize signifiers, like datebooks and envelopes, without having seen them non-virtually. Your grandchildren may well continue to click on a floppy disk icon to save their document, even though they will never encounter one in real life, and given the short span of time that floppy disks were used, and the relative lack of interesting material for historical movies, they never see one on video either. But the one thing we can say about the persistence of the floppy disk icon is that the visual designers are being consistent!

## 12.5 Consistency

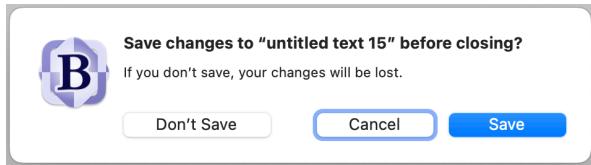
[to contents](#)

You can be consistent in your use of signifiers, shortcuts, and gestures within your application, but also across applications on the same platform. Without looking back, can you guess which shortcuts were used in the Edit menu for “Undo Typing” and “Can’t Redo” (a greyed-out version of “Redo Typing” since there were no undos to redo)? If you guessed Control-Z, then you are probably a Windows or Linux user. If you guessed Command-Z, then you are a MacOS user (or recognized the system font, and are familiar with MacOS). If you guessed Control-Shift-Z or Command-Shift-Z for “redo” then you know more than the basic shortcuts. If you know that Command-E means “Use selection for search”, then you are a power user!

If you did know those keys, it is very unlikely that you just learned them from the Visibility example. You almost certainly have used them across multiple apps, and you saved a lot of time by learning them once (maybe when you noticed they were consistent in the third app...) and using them many times. It is likely that you don’t even say “Control-Z” to yourself when undoing, because you have created procedural memories for these actions—called muscle memories, because it seems like your muscles can perform them without involving your brain<sup>1</sup>.

Even more important than consistency with other apps is consistency within your app. This means that a button to do X should be in the same position whenever it appears.

Here is the dialogue which appears when trying to close an unsaved document:



Notice that the “safe” option—save it first—appears at the right. This is actually part of Apple’s Human Interface Guidelines<sup>2</sup>. Apple’s guidelines not only ensure consistency across applications, but many of them ensure consistency across devices from 1984 to today. Unfortunately, even Apple seems to have forgotten its own guidelines when developing for web and mobile platforms.

## 12.6 Constraints

[to contents](#)

The last example is also an example of a Constraint. In this case, it is not preventing you from closing your unsaved document, but it won’t let you do it without a warning, and if you don’t really read the dialogue and press return, the resulting default action is to save the document. Constraints are about not letting the user make choices which cause data loss, or waste their time. Yet the inputs on all dialogues and forms, and don’t accept inputs

<sup>1</sup>Actually, we know this is incorrect. Your brain is definitely involved, but not your consciousness. Don Norman describes his theory for different levels of mental processing [Nor13], and explains that good designs short-circuit higher-level executive function in order to speed up processing. You experience this as working effortlessly.

<sup>2</sup>HIG button: <https://developer.apple.com/design/human-interface-guidelines/components/menus-and-actions/buttons>

you know will cause an error later—especially if the only way to fix it is to undo until you get back to the same dialogue.

Combining the principles of constraint and mapping, you should scale any controls so that illegal values are out of reach. Since we all live in countries with seven days of the week, it is not surprising that we only get seven choices in our dialogues, but it takes a bit of work to make sure that you cannot enter February 29th into a date picker on leap years. Similarly, if your air conditioner only cools down to -5°C, then make that the bottom of your slider.

Combining constraint and visibility: grey out buttons for actions in states which do not accept those actions. Going back to the first example, “redo” was greyed out because there were no undos to backtrack. Since the greying out of buttons happens in response to an action, this is also an example of feedback.

## 12.7 Feedback

[to contents](#)

Let the user know you heard them! If you are building an editor, then hopefully feedback is immediate as text appears and disappears as soon as the user types, or is highlighted as they hold Shift and use the arrow keys, or drag with their mouse.

But sometimes it takes time to perform an action. You could be processing a gigabyte of data, or waiting for an internet query to come back. In those cases, you need to do something to avoid the user banging on the keyboard or mouse, and we have a great convention: the progress bar. Contrary to popular belief, most progress bars do not actually indicate progress, they are programmed to move along according to some estimate of how long a task will take. You are probably quite familiar with installations which seem to be zipping along, and then stall, and then zip along again. No matter how suspect, you probably do feel better with a progress bar than a spinning ball, and a spinning ball is a lot better than no change in the interface.

Another feedback convention which has been created for web apps is colour change on mouse-over of a button. Some people who used the original 7.8MHz Macintosh are a bit puzzled about the need for this. If it was possible for a processor 500 times slower, and without any graphics processor support, to provide adequate button feedback—for users who had never seen a mouse before—why do we need it now?

Maybe those greybeard users complained to Google, because the wizards behind Material Design 3.0<sup>3</sup> have decreed that this animation will result in an 8% change in opacity for a black (really smoky) overlay. As you can see:



the result is a subtle colour variation which shouldn’t irk anyone old enough to have used the original Macintosh—because they will barely see it (and even more elderly users—if you can imagine that—will not be able to see at all, since our visual acuity declines with age). So to sum up,

1. web designers create buttons people don’t recognize as buttons and laggy apps which make people question whether they clicked on a button or not,

<sup>3</sup>material design: <https://m3.material.io/foundations/interaction-states>

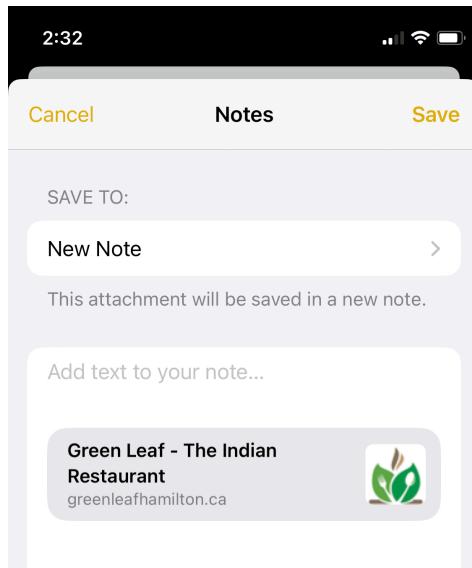
2. they then add a feedback to let you know your mouse is over a button, and
3. they complicate the API to automate the new action contributing to bloated applications, and finally,
4. they make the new action invisible to the users most likely to be confused by new-fangled interfaces.

Clearly, capitalism is working well, as evidenced by our billion-dollar companies wisely investing their monopoly-generated rents.

Since I know some of you will feel I'm picking on those underdog billion-dollar companies, I should point out that the arrival of the web app did pose a challenge for visual designers. Do you design your app using the conventions for pre-app web pages—knowing that those conventions were designed for hyperlinked documents, and not apps? Or do you design your app to match one of the desktop OSs, or attempt to detect the OS and then match that OS? But what about information appliances (devices with a browser, but no visible OS)?

No wonder they gave up and went for “fashionable”. Unfortunately, fashion meant American fashion, and this was a particularly dark (well, actually beige) time in American fashion<sup>4</sup>. Could anything be worse than beige? Well, constantly changing in order to seem “modern” would rank pretty highly. There is another billion-dollar company who touts their ability to update their cars’ software over the air. How do you think your 70-year-old car owner is going to react to their car’s interface changing just when they need to drive to the hospital?

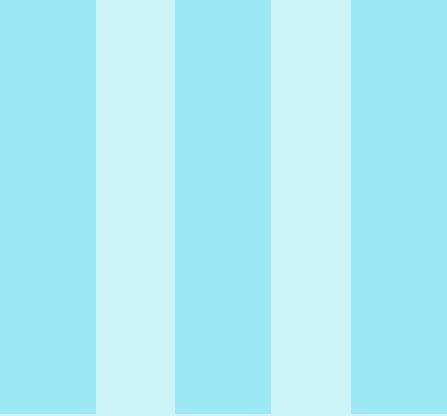
Which brings us back to Apple, and their interface for saving notes:



To get here—no antiquated floppy-disk icon in sight—we have to share (with ourselves, I guess). Notes can get lost, so it is helpful to be able to put them in folders. But how do you access the folders? Folder icon? Something which predates the floppy disk, and everyone understands? Nope. I still cannot come up with an explanation for this interface, and I don’t mind admitting that it took me years to figure out that Notes even supports folders!

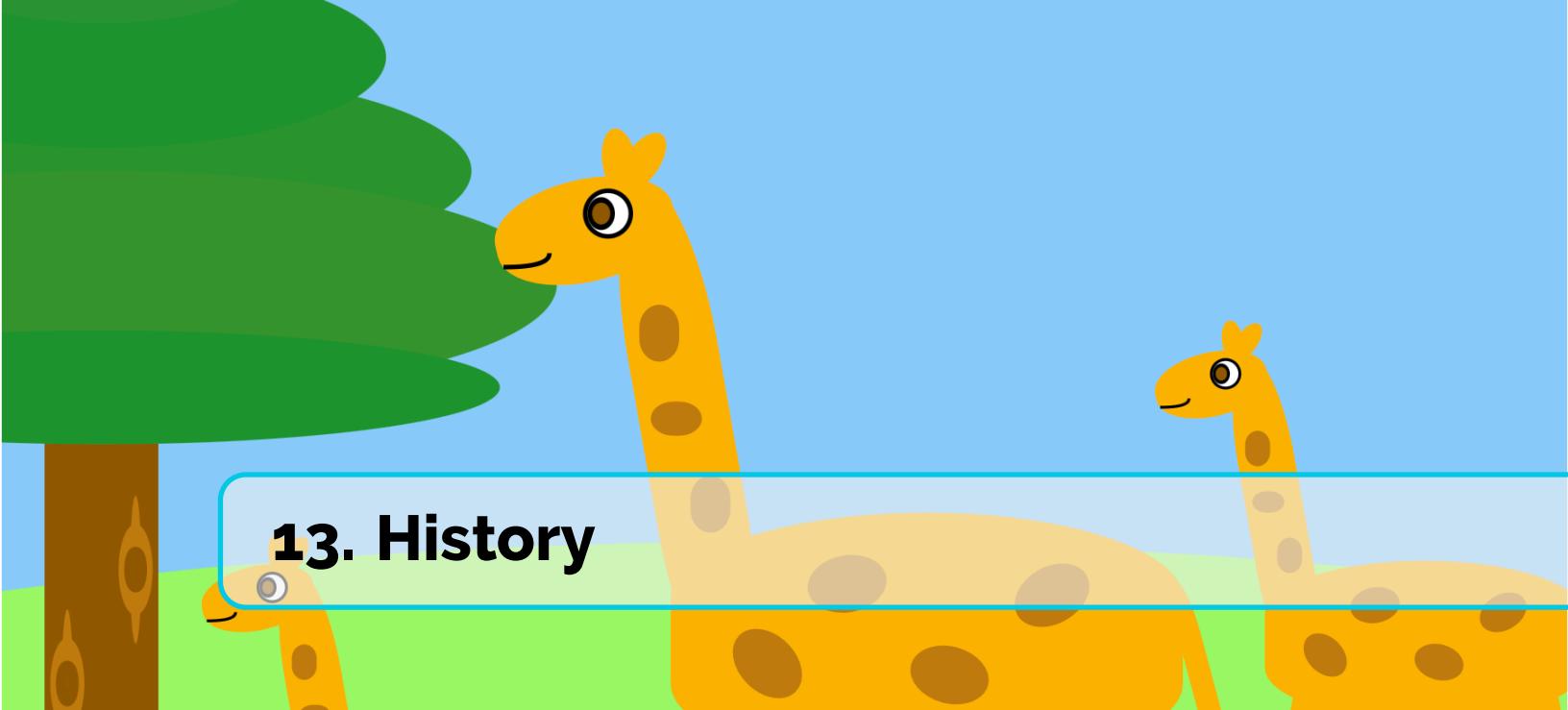
<sup>4</sup>beige “design”: <https://www.atlasobscura.com/articles/how-beige-took-over-american-homes>

At this point, you may be saying, if trillions of dollars of capitalization cannot guarantee good design, what hope do we have? Think instead of this as an opportunity for a new startup!



# Design Thinking

<b>13</b>	<b>History .....</b>	<b>194</b>
<b>14</b>	<b>Example: This IS your Grandfather's Gaming App .....</b>	<b>202</b>
<b>15</b>	<b>Design Thinking Templates .....</b>	<b>290</b>
<b>16</b>	<b>Example: Math Visualizer .....</b>	<b>316</b>



## 13. History

When you hear “design” you probably think about architecture and fashion, something requiring genius or at least inspiration. In fact, design is just another skill. Some of us seem better at it than others, but everyone can get better through practice and reflection.

You may have heard of people talk about something being “more art than science”. What do they mean by that? And when people say “it just can’t be taught,” are they serious? As with many sayings, there is some truth underlying them. In everyday language, when people talk about something being a science versus an art, they are probably thinking about the organized way in which science is taught in school: hypothesis, method, experiment, and conclusions. So something that is more art than science is something which does not fit this systematic approach. The funny thing is that if you think back to art class, you will probably remember lots of systematic instructions on how to mix colour, apply glue evenly, handle exacto knives safely, draw with hidden lines to get perspective right, or use shading to convey 3D geometry.

Conversely, even a stack of neatly penned lab notebooks stretching from here to the moon would have amounted to nothing without good hypotheses. How does good hypothesis generation work? Was it even covered in your science classes? Well, you could say it is more art than science!

We could chuckle about the contradictions in popular perceptions of art and science, but it’s nothing to laugh at. Progress in science is slowing down<sup>1</sup> just as we face the toughest challenges in known history, including slowing and adapting to climate change and containing new diseases<sup>2</sup>.

### 13.1 Herbert Simon and Design Science

[to contents](#)

Fortunately, there is a science to this, and even a Nobel prize winner (Herbert Simon in 1969). Simon’s focus was making good decisions. Starting out in political science, he learned

---

<sup>1</sup>cf. “Are Ideas Getting Harder to Find?” [Blo+20] <https://web.stanford.edu/~chadj/IdeaPF.pdf>

<sup>2</sup>Just look up avian flu in birds, and drops in frog and pollinator populations.

behavioural economics, logic, cognitive science, and optimization, and helped establish the field of artificial intelligence—all to understand decision making.

One of Simon's big worries was the way in which “professional” engineering and management programs were throwing out practical doing in favour of knowledge and skills related to established scientific theories. Established theories made it easy to teach and to test knowledge. Grading creative problem-solving is pretty hard, and very hard for inexperienced teachers to do fairly. When education was rapidly expanding, it was just easier to abandon design, which is about defining problems and making decisions, rather than solving problems by following a recipe. This is not to say that Simon devalued basic skills and knowledge, because he argued just as strenuously for improving their teaching.

Simon's first contribution to decision making will probably seem obvious to you, but it was *not* obvious to economists at the time: people make decisions based on limited information. Even today, with the internet giving us access to more information than we could digest in 100 lifetimes, we often cannot find the information we need to make decisions. Polluters do everything they can to hide the impact of their pollution. Drug companies will only fund studies which could turn a profit. The topics we most need information about are often the ones we cannot even search for because we don't know the right search terms, and the search engines we rely on make money not by giving us the best answer, but by advertising something people want to sell us. So is it surprising that people make decisions without the needed information?

To counter this, Simon had a simple formula for good decision making,

1. Intelligence: identify the environment, where does the problem come from?
2. Design: invent possible solutions, and build prototypes.
3. Choice: pick the best choice after weighing all the results.

For a good summary of Simon's ideas, see [Sim19].

While his concepts are credited as the start of modern design theory, we need to acknowledge that much of the theory, and most of the practice, were developed by working designers who probably did not know about Simon's or anyone else's theories of design. Many teams also developed their own methods independently. Mechanical engineers, architects, graphic designers, and user-interface designers all developed their own versions of many of the techniques we will describe. This is natural when design practices grow over time as people face harder and harder design problems. Even today, people often reinvent ideas because they don't even know the words for what they are inventing.

But while many creators of design culture didn't know about Simon, it is worth understanding his ambition to apply scientific methods to this problem. In fact, before Simon won the Nobel prize, he won the Turing Award<sup>3</sup> (the top prize in Computer Science) together with Allen Newell<sup>4</sup> for foundational work on artificial intelligence, the psychology of human cognition, and list processing. It all comes back to how humans make decisions with limited information. He reasoned that if they could develop a system of computation with symbols which reached similar decisions to human decision makers, the theory behind the system would be validated as an explanation of how humans think. Not only were they

<sup>3</sup>[https://amturing.acm.org/award\\_winners/simon\\_1031467.cfm](https://amturing.acm.org/award_winners/simon_1031467.cfm)

<sup>4</sup><http://act-r.psy.cmu.edu/misc/newellclip.mpg>

successful in creating such systems, they interested many other researchers at Carnegie Mellon University to take up the problem, or to use computer simulations to answer new questions. Some of them took things to the next level and developed cognitive tutors which incorporate a computational model of thinking called ACT-R<sup>5</sup>. In this way, the tutor builds a model of why a student makes a mistake, so it is possible to correct the underlying problem, just as an experienced human tutor would do. These models are too complex to explain in this book, but you can get an idea of how this could work with a simple model for team-based design. But to explain that model, we need to introduce another big discovery which happened at that same time as Simon's early work on models for cognition.

Simon was born in 1916, in the middle of World War I, grew up during the Great Depression, and finished his university education during World War II. It is hard for anyone growing up today to understand how the richest countries in the world at the time faced such extreme poverty and really struggled to do anything about it. Economics at the time had little to offer in the way of solution. As a university student, Simon got involved in the Cowles Commission for Research in Economics looking into the application of mathematics to economics (called econometrics) which produced 11 Nobel prize winners. Simon, however, did not stop with economics, he learned about another growing field: operations research. Also called logistics, or industrial engineering, operations research is about making the best plans for building, transporting, or sharing things, so you can see how this connects to economics.

## 13.2 George Dantzig and Operations Research

[to contents](#)

Operations research was one of three massive leaps forward in mathematics which helped defeat fascism in World War II, along with the digital computer and algorithms for making and breaking codes (called cryptography and cryptanalysis). Computer science might have been much slower to develop without these achievements. Operations research seeks to maximize production given limited resources. During the war, another scientist, George Dantzig, had the job of planning logistics for the US army, and was challenged by his team to automate the massive number of calculations required. Before automating, he had to figure out the right abstract form for the problem. This is a common problem in computer science. Problems come in all kinds of forms, but we need to see the link to the right abstract problem so that we can solve them. The person we credit with introducing mathematical modeling to operations research and developing the first efficient algorithm for solving Linear Programming problems was George Dantzig. If you know about inequalities, a linear program has the form

$$\max \quad f(x, y, z, \dots) \tag{13.1}$$

$$0 \geq g(x, y, z, \dots) \tag{13.2}$$

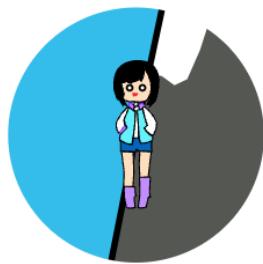
$$0 \geq h(x, y, z, \dots) \tag{13.3}$$

involving a set of variables  $x, y, z, \dots$  and functions  $f, g, h$  which are linear in the variables. Linear comes from “line” and it means that the graphs of the functions are lines, and it

---

<sup>5</sup><http://act-r.psy.cmu.edu/about/>

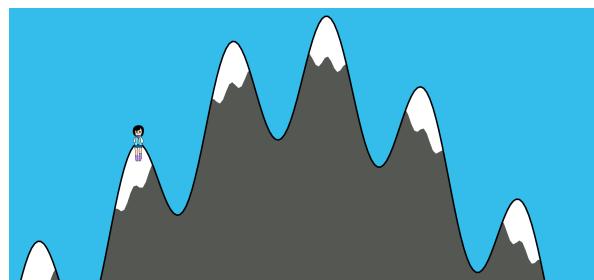
turns out that is exactly the case when the formula used to calculate each function is a sum of products of a constant and a variable, and sometimes a final constant. If you know about functions—great—otherwise, don’t worry, because we will explain functions and linear functions in this book. Now we’ll see how abstract optimization problems can simulate the process of searching for solutions. If your design problem is simple, like designing a box, you have a few things to change, like the height, width, and depth of the box. As a metaphor for finding the best solution, think about trying to find the highest mountain. The height of the land can be represented by a function, and the highest mountain will be at the point with the greatest height. Each designer has limited knowledge, because every person on earth has limited knowledge. In the mountain scenario, that is like walking around in the fog where you can only see a few steps ahead, or in the dark where you can only see as far as your flashlight.



In that situation, you start walking in the direction with the steepest slope. Eventually, you will reach a peak.

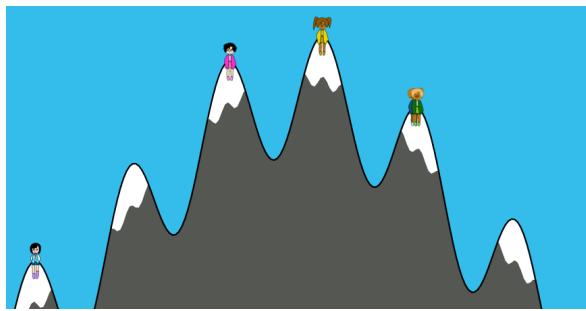


But since you can only see what is around you, you don’t realize that your peak is actually a lot lower than another peak.



Hmm, we need a new strategy. We could ask our friends to help, but if your friends go to the same school, live in the same neighbourhood, play on the same sports team, speak the same language, etc., then you start out about the same place in the idea landscape, and chances are, you will end up at the same peak. We could put another picture here, but since you are all so alike, we won't be able to tell how many people are standing with you on your peak.

Another approach is to build a diverse team, with people who were born and grew up in different places *and times*, speak different languages, play different sports and musical instruments, and read different books. With different backgrounds, you approach the problem from different starting points and will come up with different solutions.<sup>6</sup>



The difference between this optimization problem and the problems you tackle inside your head is that most problems have *discrete* choices in addition to *continuous* choices. Discrete variables require leaps. Instead of building with wood, you could build with bricks or straw. We can count the possibilities. Continuous variables can be expressed by numbers, including fractions. So we can make steps of any size. If building from straw, the amount of straw we use is such a variable. Adding discrete choices into an optimization problem makes it much harder to solve, and even more complicated to write down.

One detail missing from this description, but important in how the problem gets solved in your head, or in a computer, is that the amount of information we can work with at one time is limited in both cases. Simon introduced the idea of *bounded rationality*, recognizing that people never have all the information they need to make perfect decisions. His colleague at Carnegie Mellon University, John Anderson, built the concept of *working memory* into his computer simulations of problem-solving, which is the level of detail he needed to get real-world results in the form of cognitive tutors. Computers are more diverse than people, who can juggle 7 pieces of information at a time. Early computers could handle 8 or 16 numbers, and some could only handle numbers from 0 to 15! Most computers today can handle 32 numbers (of much larger size), and designers can change this number, but it is a trade-off between speed and the size of their working memory, which we call the *register file* or just *registers*.

Although we still recognize George Dantzig for inventing the simplex algorithm for solving linear programming problems, which are still of immense importance to operations research, it turns out that Leonid Kantorovich had formulated linear programs in 1939 while working on the optimization of wood production in the Soviet Union. If not for the Second

<sup>6</sup>See for yourself: <https://cs1xd3.online>ShowModulePublish?modulePublishId=1d26d9d8fea2-4825-8cee-2e8004fd3882>.

World War, he may have continued to work on this problem and shared ideas with George Danzig. Their parents were born in neighbouring regions of the eastern Baltic Sea, but the Danzigs emigrated to the United States.

From this we can draw an important lesson: it is important to find out how other people are approaching your problem, even though you will have to renew your search every time you learn about new words to describe it. Today, a program is a description of steps a computer can take to solve a problem. But this was not always the case, and a “Linear Program” is really a problem description, not a recipe for solving it. It is one of the few times we use the older meaning of Program.

## 13.3 Theory + Practice

[to contents](#)

“In theory, there is no difference between theory and practice, but in practice there is.” This quote was first used in 1882 by student Benjamin Brewster.<sup>7</sup> It captures the frustration many of us feel about elegant theories which we cannot see how to put into practice. Universities have struggled with the relative importance of theory and practical results, and Simon’s interest in education was sparked by what he saw as an abandonment of theory in the newly created engineering schools. Progress takes time, and it is hard to judge the usefulness of a theory when it is created, but it is also easy to get carried away creating theories without any hope of application. It is not surprising that a student came up with this wry comment, because students are often left to navigate between the worlds of theory and practice. University professors and researchers are secure in their positions and judged on the advancement of theories. Practitioners have figured out a way of getting things done, and are often not interested in “fixing” what works, and ignore long-term questions like sustainability or the emergence of competing technologies. That leaves new graduates between a rock and a hard place.

Almost all universities now have separate divisions for Science and Engineering, who often compete with each other for students, attention, and funding. Computer Science, the relative newcomer, lives uncomfortably in this artificial landscape, because although it has “science” in its name, the vast majority of its graduates are employed in practical development projects. We would argue that there is room for both theory and practice, and it is particularly exciting to live in the interface between them. Unlike most other sciences, the cost and time commitment for doing a lot of computer science experiments is very low. So it is possible to see the real-world impact of theory. In fact, every time we debug a program, we are creating a theory of operation for the program and testing the actual program against the theory. We can do that implicitly, by viewing programs as fun logic puzzles to figure out, but for big programs, this approach breaks down, and we need more theory. That theory is often borrowed from mathematics, like the optimization theory we just talked about. Linear algebra (the theory of vectors) is the foundation for all image processing and medical imaging, as well as most newer developments in Artificial Intelligence. Number theory is the foundation for cryptography, while logic underlies all of digital electronics, from the button to call an elevator to the billions of switches inside a smartphone.

<sup>7</sup>According to Quote Investigator <https://quoteinvestigator.com/2018/04/14/theory/>

We will talk more about some of these applications, but what about Simon's decision-making questions? Those questions, and many more complex questions which followed, have caused a real revolution in thinking about economics, cognitive science, and artificial intelligence. Although we cannot connect our low-level understanding of neurons in the brain to complex behaviours like choosing what music you will listen to while reading this book, cognitive scientists (including Simon himself) have developed many models which show how simpler circuits and processes can lead to more complex behaviour. These are not full theories of the mind, but they create building blocks which later generations can try to fit together to build such a model one day. These are questions of science: what are the mechanisms of the mind, and how do malfunctions in these mechanisms lead to bad decisions or even mental illness? It is an endless loop of theory, predictions, experiments, and interpretations. Design Science has proven through successive insights into how we make decisions, individually and collectively as part of a team, that it is a successful science.

And in practice? Have insights from Design Science led to recipes for designing better products, better meeting end-user needs? Have they reduced the number of failed projects, or reduced the cost of developing a new product? Have they created education pathways so that anyone can be a successful designer? I'm sure you are expecting me to say "yes" at this point! The answer is more complicated. We do not have a clear answer, in the way that we know that the earth is round. Scientists have been trained to consider a question answered when we have multiple controlled trials<sup>8</sup> with statistically significant results. But we do have evidence that cognitive science foundations do lead to practical innovation. ACT-R is the best example. ACT-R was developed by John Anderson to show that symbolic computation with limited resources can simulate observed behaviour[And07], but turned out to have practical application as the heart of cognitive tutors. We can perform sufficiently controlled experiments to measure the impact of this theory on learning, in this case, the learning of high school algebra. But if exploration is a metaphor for design, then teaching algebra is like exploring in a zoo. Algebra already abstracts away most of the complexity of real-world problems. This is why it is so powerful, but also why it is not representative of general decision making.

What about design in general, and Design Thinking in particular? We have strong evidence that some of its underlying theories are sound. We do not have any evidence against it. Should we use it to make decisions, or wait for better evidence? This is where a misunderstanding of statistics and the scientific method can be a force for evil! In many ways, our democratic societies are paralyzed by a nonsensical expectation that we can always perform these types of experiments before making decisions. Ok, but did I really mean to write "evil" in the sentence above? Yes, I did, because we have several examples where the sci-

<sup>8</sup>A controlled trial is a type of experiment where we decide in advance that some people get the test treatment, and others get a fake treatment. It is "double-blinded" if both the doctor and patient wear "blindfolds" and do not know which treatment a patient is getting. Design Thinking could be studied in this way. We would find two companies where nobody knows how it is supposed to work, teach one company DT and the other a made-up look-alike, with lots of colourful sticky notes, but not used in any of the ways we think are helpful. It would be a very expensive test, especially if we wanted it double-blinded, because we would have to teach the teachers the fake method together with reasonably sounding justifications which would nevertheless be total nonsense. Nobody is willing to pay for this.

entific method and normal scientific discourse have been used as a weapon for profit and against human health. The first was the fight to protect people from cancer and lung disease caused by tobacco. The second is the fight to protect people from man-made climate change.

As individuals, we make countless decisions every day without very much evidence. How do we do it? When we do it well, it is because we never stop learning, or said another way, we change our minds a lot! So in parallel with the evolving science, our practice of design has also evolved through learning. Going back to the design as exploration analogy, we talk to the adventurers who set out without a map, and made it back alive: the architects, fashion designers, engineers, business people, and computer scientists. Although they cannot explain why their methods work, and will often take you on a roundabout rather than a direct path, they have been successfully designing for thousands of years. Over the last 50 years, this process has been used to find common paths taken by different designers, often describing the same landscape using different vocabulary. We can also learn from their hair-raising tales of close escapes from projects which crashed and burned. We now have names for some of the monster squids and dragons inhabiting uncharted waters at the edge of the map: prematurely picking a solution; investing in fully baking a solution without prototyping every step of the way; assuming your user is like your friends; and groupthink. For a business point of view on Design Thinking, see Jean Liedtka<sup>9</sup>, who explains why it is being adopted by the smallest startups, and the world's most valuable company (as of August 2022),

## 13.4 The Double Diamond

[to contents](#)

The map which has helped steer clear of these hazards and chart successful courses through many small-scale software projects is the Double Diamond. It was created by the British Design Council, borrowing from Béla Bánáthy's work on designing social systems [Ban13].

Before we start describing the Double Diamond, it is important to make it clear that experiments are not exclusive to Design Science. They are just as important for Design Practice, but whereas scientific experiments aim to answer questions about the natural world (which includes the human mind), practical experiments aim to answer questions about the artificial world we are creating. For software developers, these questions are often about what our users really want, since they often give us confused or conflicting demands in the face of infinite possibility. But design questions may also be about the limits of technology or of social systems to adapt to change.

Practical experiments are called "prototypes". Just as failures of scientific experiments can raise doubts about accepted theories and force us to develop better ones, failures of prototypes should tell us that we do not understand our users' needs, and we need to rethink our goals.

<sup>9</sup>See <https://jeanneliedtka.com> including a very readable article at Harvard Business Review.



## 14. Example: This **IS** your Grandfather's Gaming App

To guide learners through the Design Thinking process, we have created a sequence of worksheets which do three things:

1. broken the process down into steps, so that you can concentrate on learning one step at a time;
2. created tables and graphs to indicate the level of detail recommended at each stage;
3. included a map in the top-right corner to remind you where you are in the process.

You will notice that these design choices have the effect of reducing pressure on working memory! But there is still a lot to take in, so rather than jumping into the slides right away, we present an example. This way, you can absorb some of the details, and see the overall structure, which is definitely harder to see when you are wrapped up in your own problem. The example is presented as if the slides were filled in one at a time, and are, in fact, based on the slides filled in by dozens of project teams all working on the problem of designing an app to help detect and monitor treatment of Parkinson's disease and other neurodegenerative diseases. Their goal was to be able to say "This *IS* your grandfather's gaming app!"

We encourage you to read through the entire chapter quickly to take in the landscape. The discussion after each slide is focussed on this instance of using Design Thinking on this problem. The interviews and some ideas were invented for this presentation, based on real interviews and ideas from multiple teams, as well as our experience teaching Design Thinking to hundreds of teams.

After your first read-through, we hope you will have the chance to learn from your own mistakes by going through the slides with your own team and problem. You will have detailed descriptions of each slide in the next chapter, but you can always come back and look at that part of this worked example.

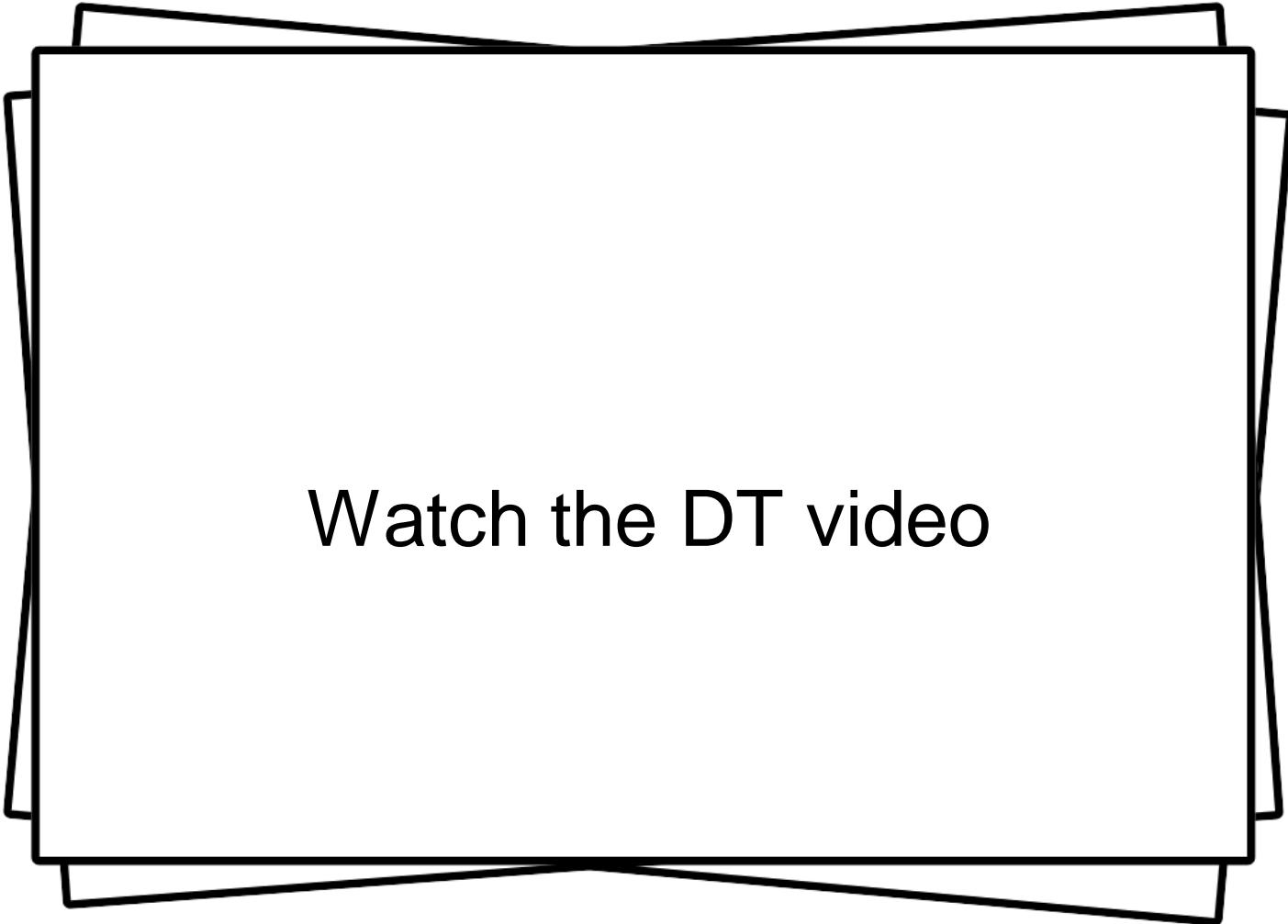
I hope we don't need to say this, but we will anyway! This is not like a worked example in Algebra—there really are no right answers. By seeing one path through to the end, we hope you will be inspired to create your own.



# DT Slide Worksheets

Winter 2022  
TEAM CS1XD3

This completed slide template is motivated by the goal of helping people affected by Parkinson's disease. This idea is a general starting point for the project, and the template helps to narrow down this idea by defining a problem and solution statement, and supporting the development of initial prototypes. The only other specification of the project is that the solution will be software-based, as this is a computer science course.

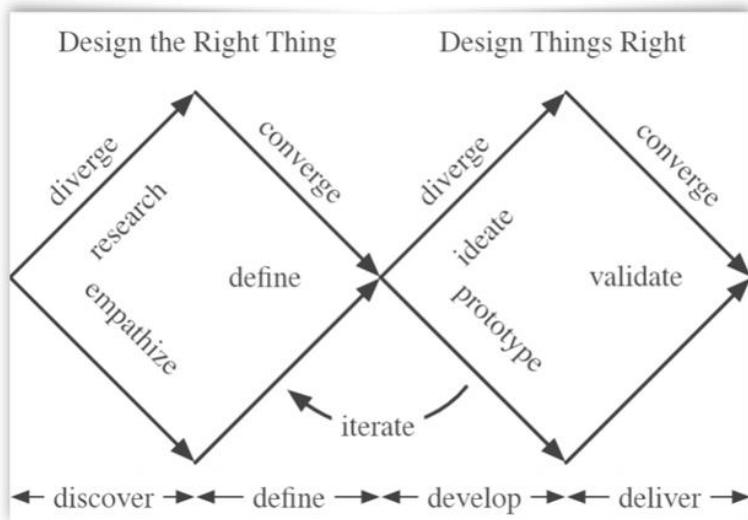


## Watch the DT video

The Design Thinking Presentation video is designed to give you a general introduction to the topic, and it can help contextualize the importance of this approach to problem-solving:

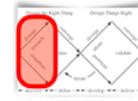
[DT VIDEO](#)

## DOUBLE DIAMOND



The Double Diamond is a map of the process. This slide is here to remind you to not forget the big picture, follow the map, and not get lost. In Chapter 15, we will explain its history. For now, you need to know you need to think divergently about your problem, and narrow down into a problem statement your team understands and agrees with, then you do it again, thinking divergently about solutions, making multiple prototypes and, based on what you learn, converging to a working solution.

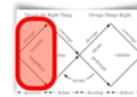
# Possible Project Areas



1. The development of Parkinson's disease (PD) and early symptoms.
2. Managing exercise and PD.
3. Social awareness and access to educational resources regarding PD.

Parkinson's disease (PD) was chosen to be the focus of this project, but this topic is multi-faceted. Here, three sub-areas of the topic were identified, which fall in the categories of disease development, management, and awareness. Note that this is not a list of problem statements, as reflected by the lack of an explicit target user and specific problem, but it will guide your journey of creating a problem statement.

# Option 1 Research



1. What could we find out about option 1? *Early detection and monitoring are important for managing PD in the growing population of the elderly*
2. Is it something people find hard to learn? *Measurement of PD severity is highly tedious and not a lot of information on the monitoring of early symptoms are made widely accessible to everyone*
3. Why do people need to know it? (Will they be motivated by applications?) *People should be able to monitor the early symptoms of PD and/or be able to measure the severity of PD*
4. Is there a good visualization component? *There are multiple ways to use smartphones to create a good visualization component (game app)*
5. How will we find people to interview? *We can find people to interview by finding people who are in the age range of being affected by PD and ask them how familiar they are with PD and how accessible it is to them.*

This is where you get started with your research. It is important to be thorough and answer the 5 questions. This will help you learn more about your chosen topic. This will then allow you to formulate better questions, and later will help you write a clear and focussed problem statement.

- Question 1 should include all that you found about option 1 (motive behind choosing that specific area).
- Question 2 should answer how difficult or easy it is for people to learn about that specific topic. Add in why or why not it's hard to learn.
- Question 3 should explain why it is important for people to know about the chosen project area. It should also encapsulate whether people would be motivated by applications.
- Question 4 should show whether there is scope for a good visualization component.
- Question 5 should elaborate on how you should find people to interview and what connection they have to the specific project area.

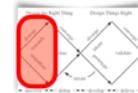
## Option 2 Research



1. What could we find out about option 1? *Exercise can improve gait, balance, tremor, flexibility, grip strength and motor coordination. Exercise such as treadmill training, biking, Tai Chi and yoga have been shown to benefit Parkinson's symptoms. [1]*
2. Is it something people find hard to learn? *Learning how to exercise is something one could easily learn to do. Either from watching videos and self learning or taking classes.*
3. Why do people need to know it? (Will they be motivated by applications?) *People need to know it so that they can use the knowledge to improve the symptoms of PD.*
4. Is there a good visualization component? *There could be several ways to visualize it. We could have a video which people follow or have a physical guide/mentor helping them or an app which is like a personal mentor for exercise with specific goals.*
5. How will we find people to interview? *We can interview people who are affected by PD or are in the age range of experiencing the symptoms of PD and see what helps them.*

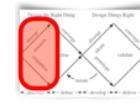
[1] <https://www.parkinson.org/living-with-parkinsons/treatment/exercise#:~:text=What%20Parkinson%27s%20symptoms%20can%20improve%20with%20Tai%20Chi%20and%20yoga>.

## Option 3 Research



1. What could we find out about option 1? *Not a lot of people know about most of the symptoms of PD and educating people would be a great way to create awareness and let people know about PD and its symptoms.*
2. Is it something people find hard to learn? *Depending on the age range, workshops could be difficult to access for some or could be difficult to pay attention to for others. In general, people might not have access to it.*
3. Why do people need to know it? (Will they be motivated by applications?) *People need to know it so they are aware of PD and how the symptoms can show.*
4. Is there a good visualization component? *Visualization can be added through engaging presentations or interactive videos*
5. How will we find people to interview? *We can find people to interview by finding people who are in the age range of being affected by PD and ask them how familiar they are with PD and how accessible it is to them.*

# Our Focus



1. Target users: Adults that may be at risk of developing PD. Specifically, adults that are of age 50 years or more [1]. Grandparents are likely to satisfy this criteria and be accessible for interviews.
2. General problem area: There is insufficient data collected regarding the progression of PD symptoms, which prevents the development of techniques and technology that can detect the earliest presenting PD symptoms, necessary for the best possible prognosis.

[1] <https://www.hopkinsmedicine.org/health/conditions-and-diseases/parkinsons-disease/young-onset-parkinsons-disease>

After exploring possible project options, your research should be used to justify which area is likely to be the most successful if pursued. In this case, the research says that centering this project around the idea of inadequate collection of data for early symptoms of PD is most likely to be a successful and impactful project focus. Explicitly stating your target users and general problem area at this point in the project will help with traceability and ensuring that your group is on the same page in the critical first steps that will shape the project. Defining the target users as adults that are of age 50 years or more will ensure that all group members are interviewing the same target audience when they venture off to perform interviews either individually, in partners, or in small groups.

You may need to do research into who your target audience should be, if it's not obvious. In this case, since we are focusing on the development of PD, we had to do some research into who is at risk of developing PD. Remember that the general problem area is not a substitute for the problem statement. In contrast to this general problem area, the problem statement should be more specific. For example, the problem statement may concern a specific symptom of PD.

# Intro Script



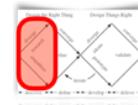
What would you say to a potential interviewee to explain why you're excited to work in this problem area and why you need their help?

Did you know that 100,000 Canadians live with PD and 25 Canadians are diagnosed with the disease every day [1]? The quality of life of those living with the condition can be drastically improved if early detection and intervention are involved. For these reasons, we want to make it easier for people to get screened and diagnosed. Further, symptom tracking would allow doctors to monitor symptoms of PD throughout treatment.

[1] <https://www.parkinson.ca/about-parkinsons/>

You should think about how you will give your interviewees context prior to conducting the interview. As seen in the slide, the background you give them should be concise, but should facilitate the interviewee's excitement or interest about the project. You can think of this like an elevator pitch – what can you say that will convince the interviewee to want to participate in the interview?

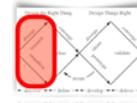
# Questions to Ask



1. How are you?
2. What are some of your favourite hobbies? How often do you do these hobbies?
3. **Do you like gardening?**
4. What do you enjoy most about spending time with your grandchildren? (If interviewing a grandparent.)
5. Do you use technology regularly? How often and which device?
6. What is the main device you use (e.g., laptop, TV, smartphone, etc.) and why (e.g., recreational, shopping, communication, etc.)?
7. When did you first learn to use a technological device? Which device was it?
8. What do you not like about technology?
9. What do you like about technology?
10. Do you have a favourite app? If yes, why is it your favourite?
11. Do you play any phone games? Why do you play the ones you play?
12. **Would you play games on your phone if it meant you could help people with PD?**

As discussed in the Design Thinking Template chapter, you should aim to make your interviewee comfortable and the interview conversational. Remember that you will rely on the same interviewee to gather feedback for later stages in your project, making a positive relationship with them crucial. You should ask them how they are, as reflected in question 1, and you should be sure to thank them at the end of the interview. In this initial brainstorm of interview questions, write down all your ideas for possible interview questions. These questions are there to ensure adequate topic coverage and fill in gaps in conversation, but if the opportunity to ask impromptu follow-up questions arises, you should always take it. This will help the interview flow more naturally, and it will ensure that you truly understand what is important to the person you are interviewing. If possible, you should do the interviews in person or through a video call. This way, you can pick up on non-verbal clues like facial expressions and gestures.

# Revised Questions



1. How are you?
2. What are some of your favourite hobbies? How often do you do these hobbies?
3. What are you passionate about?
4. What do you enjoy most about spending time with your grandchildren? (If interviewing a grandparent.)
5. Do you use technology regularly? How often and which device?
6. What is the main device you use (e.g., laptop, TV, smartphone, etc.) and why (e.g., recreational, shopping, communication, etc.)?
7. When did you first learn to use a technological device? Which device was it?
8. What do you not like about technology?
9. What do you like about technology?
10. Do you have a favourite app? If yes, why is it your favourite?
11. Do you play any phone games? Why do you play the ones you play?
12. Can you describe your typical day? What do you do?

In this slide, you should think critically about the questions you generated in your initial brainstorm. Ensure that your questions are specific, but not closed-ended. Additionally, try to stay away from questions that necessitate a simple ‘yes or no’ answer. Question 3 in the initial Questions to Ask (Do you like gardening?) is a considerably closed-ended question. An interviewee might answer yes to this question, and that might lead you to believe that this person really cares about gardening. However, they might have just said yes because you asked them a specific question and they don’t hate gardening. If everyone said yes to this question and your solution ended up relying on this fact to garner interest of your target users, you might find that your target users are indifferent to your solution because they were never really that interested in gardening.

Instead, you might ask revised question 3: what are you passionate about? Question 12 in the initial Questions to Ask brainstorm (Would you play games on your phone if it meant you could help people with PD?) should also be revised. In this case, the reason to revise this question is 2-fold. First, it is closed-ended and most people will be inclined to say yes due to the nature of the question. Second, this question reveals that this stage in the project has prematurely become solution oriented. Remember that these initial interviews are here to help you familiarize yourself with the target user. You will get to know how they feel about

your solution later (in your prototype feedback interviews).

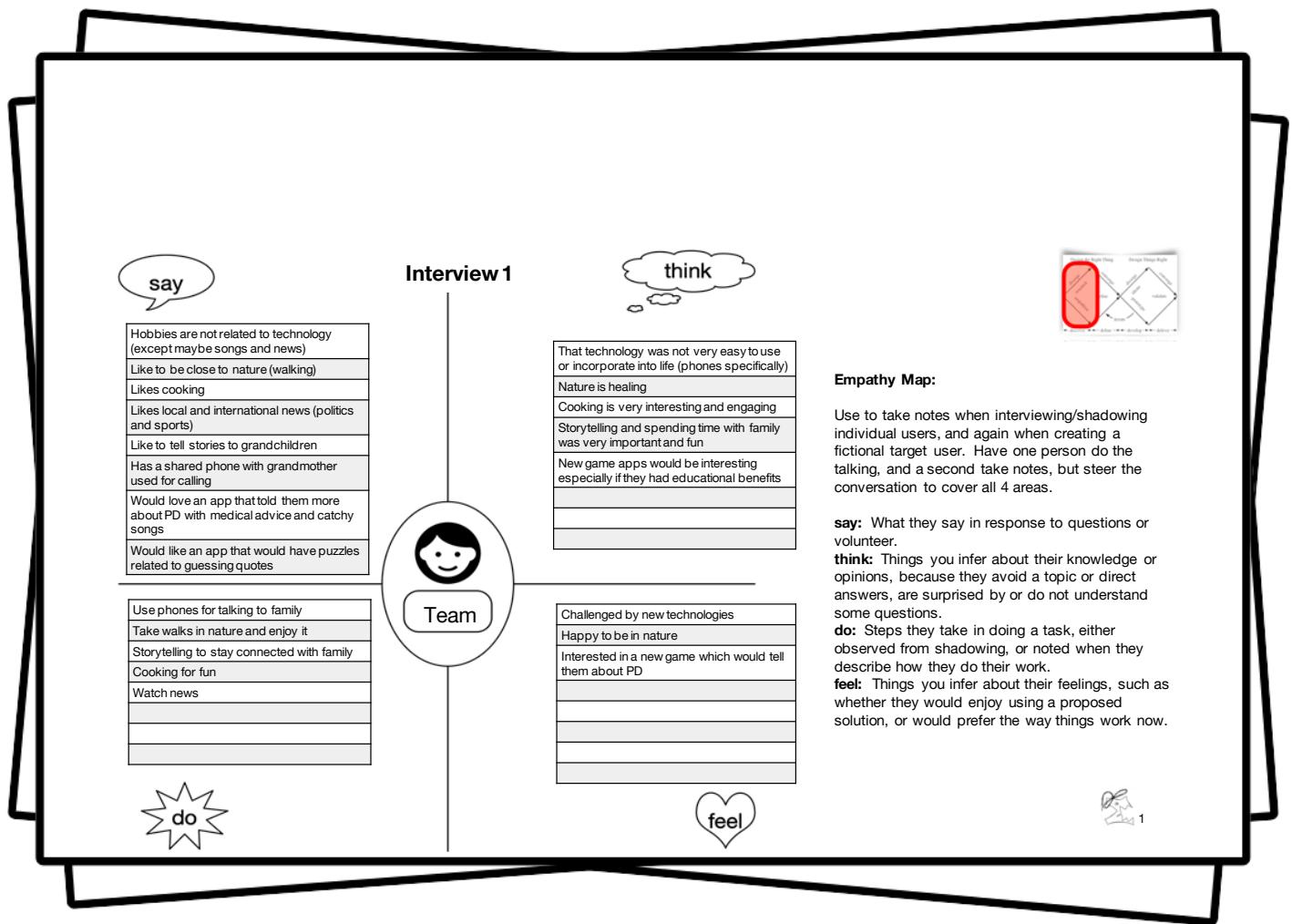
Further, by staying away from solution-oriented questions, you are more likely to produce a solution that the target users care about, as opposed to something that you care about or something you guessed they care about. You should consider asking your interviewee questions that make them describe something step-by-step, as done in the revised question 12. They might reveal something that has a big impact on their life that you never thought to ask about. Finally, make sure to consult the Empathy Map before you begin conducting interviews to ensure that you are asking questions that will elicit responses for all 4 quadrants (say, think, feel, do).

## Interview 1 Raw Notes – Team Member 1's Grandfather

- I am good, have been enjoying the summer and the beautiful nature
- Has hobbies that aren't related to technology or phones like those involving being close to nature – likes to take walks in the park or walk to the local grocery store
- Do nature related activities almost everyday
- Likes cooking and watching local and international news, especially about politics and sports
- The most enjoyable time with grandchildren is telling them stories about their childhood
- No, they don't use technology often. Has a phone which he shares with the grandmother, only to answer or make phone calls
- Only technological device used is phone to make answer calls so can't say they have learnt how to use it expertly
- They like how you can use technology to stay connected with people
- Sometimes use game apps that include puzzles or quizzes related to guessing words or famous quotes. Think that it is engaging
- Typical day involves waking up, cooking if they can, watching news, calling family, spend time with grandchildren during the summer and very rarely play some crossword games
- Said that he would love an app that tells him fun filled things about how to reduce and manage PD symptoms, which medicines to take and when, and some lovely melodious songs in them (e.g., Pakistani-Punjabi folk songs) to hold their attention and interest (when asked what app they would like)

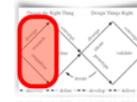
Ideally, you will have three group members present during your interview: one to take raw notes, one to take notes in the Empathy Map, and one to ask questions. People taking notes may also ask follow-up questions if they see the opportunity to do so, but a conscious effort should be made to not overwhelm the interviewee. For example, if the person taking notes in the Empathy Map notices that one quadrant is lacking data, they may decide to chime in with some questions. If a small group or pair approach is taken to perform inter-

views, don't forget to rotate roles; some people may be naturally inclined to perform better in certain roles, but everyone should get practice in every role.



The empathy map is like a heart monitor for the interview process. An empty quadrant is an alarm going off: your interview is not capturing the whole person. Some tips:

- Ensure that there are enough people to ask questions and note down points from the interview.
- Every part of the interview should be noted down; from what they say to small cues in expression or body language.
- Always ask follow-up questions and try to understand what the underlying feelings and thoughts behind what they are saying are.
- Don't make the interviewee feel uncomfortable by pushing when they don't want to answer a certain question
- Take note of the cues of discomfort as well. Nothing is insignificant!



**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.

Did you...			
set up interview, giving context?	Somewhat	Yes	Definitely!
make them comfortable?	Somewhat	Yes	Definitely!
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never	Sometimes	Always

Ask peers to give feedback and rate how you did. Peer review is great for improvement and self-reflection. Use the feedback and what you got from the reflection to rate how well you did.

## Interview 2 Raw Notes

- I am doing alright
- My hobby is to cook and learn about new food recipes. I do these very often
- I am passionate about cooking and knowing about health
- Yes, use phone regularly for communication and finding recipes
- Learnt a few years ago to use a phone to stay connected with people
- She does not like apps with so many adverts or commercials,
- She likes that technology can keep me connected with people and let her find so many new recipes and know about several things
- She likes Whatsapp and thinks it to be a reliable source of communications which does not distort the overall aura of talking to relatives, friends, etc.
- Don't really play games
- Typical day involves seeing new recipes, cooking and using phone to stay communicated with others. Also use whatsapp for getting new information
- She would use healthcare apps since they are reliable and helpful, pointing out some illnesses or symptoms of that particular illness

**say**

Use technology to learn food recipes and the like  
Would use healthcare apps since they are reliable and helpful in pointing out symptoms and illnesses  
Don't like apps with a lot of ads  
Whatsapp is a reliable source of the communications which does not distort the overall aura of talking to relatives, friends, etc.

Used technology for cooking and learning new stuff  
Does not like to use apps with lots of ads  
Uses whatsapp as a reliable source of communication with loved ones

**do**

## Interview 2

**think**

Technology is helpful to learn new things  
Healthcare apps would be good to use since they are reliable and helpful to give medical advice  
Apps with ads are not amiable  
Whatsapp is reliable source of communication

Glad to be able to use technology to learn new things  
Healthcare apps are reliable and helpful  
Ads are a little annoying in apps

**feel**



### Empathy Map:

Use to take notes when interviewing/shadowing individual users, and again when creating a fictional target user. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

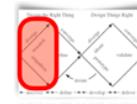
**say:** What they say in response to questions or volunteer.

**think:** Things you infer about their knowledge or opinions, because they avoid a topic or direct answers, are surprised by or do not understand some questions.

**do:** Steps they take in doing a task, either observed from shadowing, or noted when they describe how they do their work.

**feel:** Things you infer about their feelings, such as whether they would enjoy using a proposed solution, or would prefer the way things work now.

1



**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.

Did you...			
set up interview, giving context?	Somewhat	Yes	Definitely!
make them comfortable?	Somewhat	Yes	Definitely!
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never	Sometimes	Always

# Interview 3 Raw Notes

1. Very good – Had picnics, long walks, malls
2. Hobbies: playing badminton, long walks, yoga, meditation, cooking, playing board games – Uno, Carrom board
3. Most passionate about: Music Least: Gossiping during free time (using for wrong reasons)
4. Use phone regularly because it's handy – Always with you and can be used without internet for information, connecting, games, deliveries, weather, air tickets, google
5. Mainly use laptop because a lot of radiation hurts your eyes.
6. Have learnt to use technology a while ago
7. Don't like small screens and exposure to radiation
8. Favourite apps: Youtube – Shows old things from 1960s etc Whatsapp – Information is faster
9. Likes games if it can connect my grandchildren and children together (has Alerts, Use of brain, Good logic, Leads to thinking)
10. Typical day involves listening to music, playing board games, going for a walk, watching youtube, check the news, talk with family and surf the internet
11. Wants to stay fit: Can see heart rate, Blood sugar, Health parameters, Don't need to go doctor

**say**

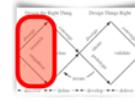
Summer was fun and that they had long picnics, went on long walks and to malls  
 They enjoy playing badminton, long walks, yoga, meditation, cooking, playing uno, carrom board  
 Phone is handy and always with you and can be used without internet  
 Use phone for information, connecting, games, deliveries, weather, tickets and google  
 Laptop is used since a lot of radiation hurts eyes  
 Use youtube for watching shows and whatsapp for connecting with family  
 Want to stay fit

Go on picnics, walks, to malls  
 Play badminton, uno and carrom board  
 Do yoga, meditation and cooking  
 Use phone for getting information, use google, weather, deliveries, etc.

**Interview 3****Team****think**

Laptop is better since a lot of radiation hurts eyes  
 Playing games, meditating, cooking and being in nature is fun  
 Phone is handy tool and helps for multipurpose tasks  
 Gossiping during free time is not good since it is used for wrong reasons  
 Music is favourite pass time  
 Staying fit is important

Whatsapp is faster since it can connect family  
 Apps with alerts, requiring brain usage, good logic and which leads to thinking will be good  
 An app which helps see heart rate, blood sugar, health parameters and avoid Dr visits would be great  
 Music is great  
 Phone is very useful

**Empathy Map:**

Use to take notes when interviewing/shadowing individual users, and again when creating a fictional target user. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

**say:** What they say in response to questions or volunteer.

**think:** Things you infer about their knowledge or opinions, because they avoid a topic or direct answers, are surprised by or do not understand some questions.

**do:** Steps they take in doing a task, either observed from shadowing, or noted when they describe how they do their work.

**feel:** Things you infer about their feelings, such as whether they would enjoy using a proposed solution, or would prefer the way things work now.



1



**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.

Did you...	Somewhat	Yes	Definitely!
set up interview, giving context?	Somewhat	Yes	Definitely!
make them comfortable?	Somewhat	Yes	Definitely!
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never	Sometimes	Always

## Interview 4 Raw Notes

- Question 1 – I am doing well!
- Question 2 – puzzles and riddles, sudoku everyday 1 time, morning walks, reading books and yoga.
- Question 3 – Passionate about yoga and staying healthy along with sudoku
- Question 5 – uses phone and Ipad, plays games like candy crush, bubble shooter, scrabble and tile matching (games that don't require much thinking and just for passing time)
- Question 6 – main device is Phone for contacting and entertainment
- Question 7 – started using technology when grandchildren starting growing. They help them learn how to use technology
- Question 8 – very difficult, knows phone by contacting and messaging and trial-error. New techniques on phones and tablets are bigger problems. When problems occur, it becomes difficult to solve it alone
- Question 9 – Likes having a phone, like a companion, feel connected to others and relaxing
- Question 10 – whatsapp and youtube for contacting Staying connected and being entertained
- Question 11 – don't play phone games. If the app is too flashy then the information in the game might disturb you while you are trying to play it. It's good to do one thing at a time.
- Question 12 – solve sudoku puzzle everyday, do yoga, play games on Ipad, use phone for whatsapp and youtube entertainment

**say**

Likes doing puzzles, riddles, play sudoku, go on morning walks, read books and do yoga  
 Uses phone and ipad to play games like candy crush, bubble shooter and other games that don't require much thinking  
 Phone is like a companion, used to stay connected and relaxing  
 Grandchildren help by making them learn about technology  
 Difficult to learn to use technology, Had to learn by trial and error  
 Health conscious; exercises and eats right amounts of food  
 Grandson uses health monitoring on smart watch and sometimes I use  
 People don't need apps to maintain their  
 Does puzzles, riddles, sudoku, reading books and exercises like walking and yoga  
 Plays games on ipad which don't require much effort to think  
 Uses phone to stay connected and relaxing  
 Does trial and error method to solve issues with technology and figure out new updates with tech  
 Stays fit by exercising regularly and eating healthy

**do**

### Interview 4



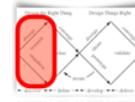
Team

**think**

Games which don't require much thinking are good for passing time  
 Phone is like a companion and helps to stay connected with people and to relax  
 Grandchildren are helpful to learn technology  
 New techniques on phone and tablets are big problems and are difficult to solve alone  
 Being health conscious is lifestyle; exercise and eat healthy  
 People don't need app to maintain their body

Glad to have games on ipad to pass time without putting too much effort  
 Happy and relieved to have phone to stay connected to people and to relax  
 Phone is like a companion  
 Feel that technology and new updates are challenging to work through alone  
 Being health conscious is important  
 If a game is too flashy then it would be disturbing to play  
 Good to do one thing at a time

**feel**



#### Empathy Map:

Use to take notes when interviewing/shadowing individual users, and again when creating a fictional target user. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

**say:** What they say in response to questions or volunteer.

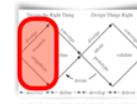
**think:** Things you infer about their knowledge or opinions, because they avoid a topic or direct answers, are surprised by or do not understand some questions.

**do:** Steps they take in doing a task, either observed from shadowing, or noted when they describe how they do their work.

**feel:** Things you infer about their feelings, such as whether they would enjoy using a proposed solution, or would prefer the way things work now.



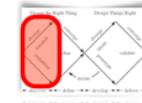
1



**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.

Did you...			
set up interview, giving context?	Somewhat	Yes	Definitely!
make them comfortable?	Somewhat	Yes	Definitely!
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never	Sometimes	Always

# Pivot or Not?



## 1. Was it easy to find potential users?

Yes, all group members have access to adults that are of age 50 years or more. We will be able to return to our original interviewees upon prototype feedback interviews.

## 2. Is there commonality among the user problems/themes/user interests?

Yes. For example, all interviewees expressed interest in fun, physically engaging tasks including cooking, gardening, and painting.

## 3. Were *interviewees* excited about it?

Yes, they seemed engaged throughout the interview and answered all questions thoroughly. Their body language (eye contact, nodding, posture) indicated their interest in the topic and project.

## 4. Were we excited about it?

Yes! We learned a lot from the interviews and were able to extract commonalities among responses, which is promising in terms of developing a solution that appeals to our target audience.

If you pivot to another idea, copy your Raw Notes and Empathy map slides and start catching up. ☺

The interview data and answers to the Pivot or Not questions justify the decision to stick with the chosen project option. Remember that the answers to these questions should follow a group discussion. The purpose of recording all-encompassing but concise answers to these questions is to ensure traceability, so make sure your group doesn't skip that part!

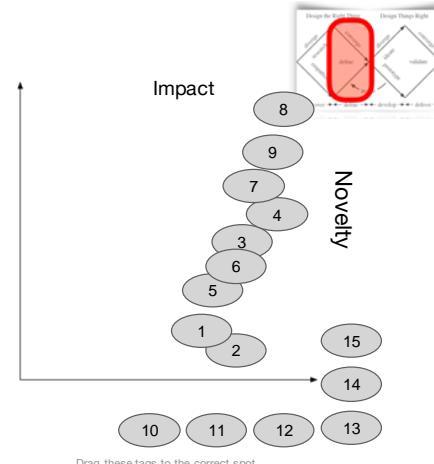
### Problem Definition

	There are not enough PD researchers that are good at their jobs.
2	There are no low time commitment ways to participate in PD symptom data collection studies.
3	There is no method/technique/technology to detect PD symptoms as soon as they develop.
4	Many PD symptoms (e.g., inability to differentiate similar colours) go unnoticed when they are first developed.
5	Older adults do not have enough energy to participate in PD studies
6	The predictability of PD symptoms and progression/severity does not have a standard and easy process
7	Patients visit clinics for PD severity monitoring during their off time (immobility), making the understanding of on time (good motor function) complicated
8	A tool for status pre-evaluation/continuous monitoring of PD symptoms is needed
9	Clinical visits have reduced to almost zero after covid, making it difficult to provide a way to help people tackle PD
10	
11	
12	
13	
14	
15	

#### How Might We?

- As individuals in the group, write five to ten different ideas on a piece of paper, for how we could make our users' lives better by reducing a problem or giving them a new opportunity.
- As a group, take turns reading your ideas, and if they are very similar to other ideas, merge them together and write down one version of the idea in the table.
- Again as a group, discuss the ideas in reverse order, and assign them potential Impact and Novelty scores by plotting the number on the scatter plot.
- Pick the best overall statement or statements, and combine them into your goal, in the box below.

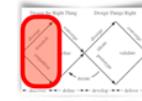
We want to find a way to help  
with  
adults at risk of developing PD, specifically those of age 50 years or more,  
being screened early, accurately, and continuously for symptoms of PD.  
need



In many cases, the problem statement is not as structured as simply as the template might make it seem. In this example template, we have 2 stakeholders, older adults, and PD researchers to consider in our process, and sometimes you might have even more. Remember, you should brainstorm as many problems as you can before you construct your final How Might We statement.

Discuss amongst yourselves and rate the impact of the problem against its novelty. For example, some problems for example can be impactful but have solutions that exist for it already; try to avoid putting effort into those kinds of problems. After your discussion and rating them, choose the problem which has the highest ranking.

## Q&A



1. Write down questions raised by the HMW process. For example, if you identified fraction arithmetic as an area your users find confusing, what grade is it covered in in Ontario's math curriculum? If PDEs cause your users trouble, what are they?

Can we further narrow down the group of people at risk of developing PD?

Roughly 10-15% of people with PD have a family history of the condition, but the genetic factors are poorly understood; hereditary cases are rare [1,2]. In addition, it has been found that men are slightly more likely to develop PD than women [3]

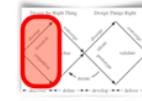
[1] "The Genetic Link to Parkinson's Disease," Apr. 10, 2022. <https://www.hopkinsmedicine.org/health/conditions-and-diseases/parkinsons-disease/the-genetic-link-to-parkinsons-disease> (accessed Jan. 04, 2023).

[2] "Is Parkinson's Hereditary?," *Healthline*, Aug. 02, 2021. <https://www.healthline.com/health/parkinsons/is-parkinsons-hereditary> (accessed Jan. 04, 2023).

[3] "Parkinson's Disease Risk Factors and Causes," Apr. 10, 2022. <https://www.hopkinsmedicine.org/health/conditions-and-diseases/parkinsons-disease/parkinsons-disease-risk-factors-and-causes> (accessed Jan. 04, 2023).

Although some research was done in the possible project areas exploration, now that there is a specific problem statement, more research should be done beyond the surface. Here, we ask if we can further narrow down the group of people at risk of developing PD. Unfortunately, it does not seem that there is a certain group in which PD is much more prevalent than another group, so we cannot narrow down our target users in this respect. Fortunately, we know that we are not leaving anything on the table in defining our target users the way we did in the HMW statement.

## Q&A



1. Write down questions raised by the HMW process. For example, if you identified fraction arithmetic as an area your users find confusing, what grade is it covered in Ontario's math curriculum? If PDEs cause your users trouble, what are they?

What are the most common, earliest presenting symptoms of PD?

The most common early symptoms are tremor, slowed movement, rigid muscles, impaired posture and balance, loss of automatic movements, speech changes, and writing changes [1]. These are all motor dysfunctions, and many of the aforementioned symptoms are associated with other diseases. Recent literature presents the idea of using colour discrimination, another symptom of PD, to differentiate between PD and other conditions [2].

[1] "Parkinson's disease - Symptoms and causes," *Mayo Clinic*. <https://www.mayoclinic.org/diseases-conditions/parkinsons-disease/symptoms-causes/syc-20376055> (accessed Jan. 04, 2023).

[2] L. Gray, "Color Discrimination May Be Sign of Parkinsons Disease," *Best Life*, Dec. 29, 2021. <https://bestlifeonline.com/vision-parkinsons-news/> (accessed Jan. 04, 2023).

We might also ask what the most common, earliest presenting symptoms of PD are. Again, we go beyond the surface level research that we did in the possible project areas exploration.

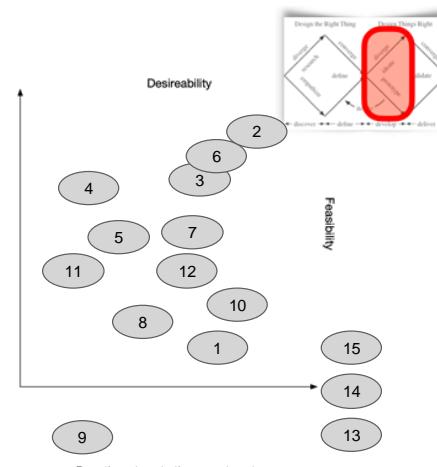
## Solutions

1	PD symptom detecting robot that follows people around.
2	Mobile colouring app that helps detect the early PD symptom of not being able to differentiate colours.
3	Watch that records movement and detects early onset tremors associated with PD.
4	Reaction game (Whack a mole)
5	Tracing musical notes
6	Cooking game app
7	Walking game (similar to just dance)
8	Puzzle game
9	Maze game
10	Dance emulsion
11	Memory card game
12	Crossword game where users pronounce instead of writing words
13	
14	
15	

### Ideation:

- As individuals in the group, write five to ten different solutions on a piece of paper.
- Include an idea costing less than \$1000 and one costing a million, include one using an app, and one without an app.
- As a group, take turns reading your ideas, and if they are very similar to other ideas, merge them together and write down one version of the idea in the table.
- Again as a group, discuss the ideas in reverse order, and assign then Desirability and Feasibility scores by plotting the number on the scatter plot.
- Pick the best overall idea, and combine them into your goal, in the box below.

Create a colouring game app that helps detect the early symptoms of PD of not being able to differentiate colours.



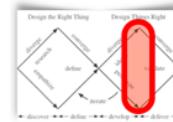
Similar to the problem definition slide, we should aim to brainstorm as many solutions as possible. Notice that the solutions are diverse and are not considered for desirability and feasibility until they are plotted; avoid premature selection and rejection of solutions by brainstorming solutions and then assessing (plotting) them.

Once you have come up with solutions, as a team rate the solutions with desirability against feasibility. For example, a PD symptom-detecting robot would be very desirable but not very feasible. There are also many studies in this area that require participants who must expend resources, notably time, to contribute to data collection. Take out solutions that don't satisfy the criteria of being easy to use. Once you have positioned everything, choose the one with the highest rank.

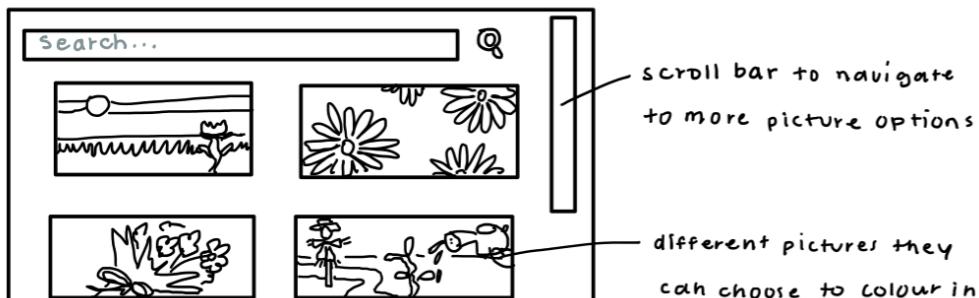
### Prototype

#### 1 (Idea 1)

Use any drawing tool you want to create a "paper prototype", including taking screen shots of Elm programs or other apps. You will need to duplicate this slide for different pages/actions in your app.



What the user sees when they open the Garden Colouring game app:

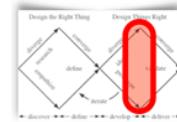


Following the finalization of the statement on the previous slide, you should develop a few prototypes that realize your solution statement. At minimum, you should develop two paper prototypes at this stage. As discussed in the Design Thinking Template chapter, these prototypes should require minimal effort while effectively conveying your solution idea, since you will perform solution feedback interviews with your prototypes. In this example, there are two paper prototypes; the first is drawn with pencil and paper and the second is made using PowerPoint. Remember that you can duplicate the slides for each of the prototypes to show different screens and app transitions. A paper prototype need not be functioning. It is your first basic idea which will be either improved further or redone based on feedback and demand. Choose something simple yet pleasing! In prototype 1, this slide shows what the user sees when they first open the app.

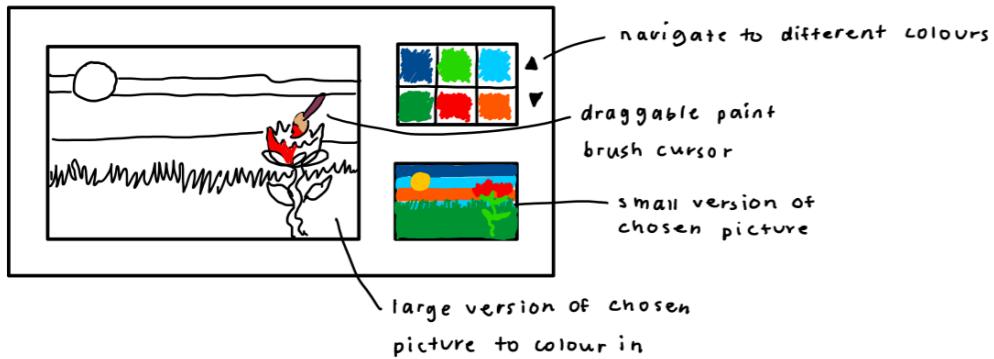
## Prototype

### 1 (Idea 1)

Use any drawing tool you want to create a "paper prototype", including taking screen shots of Elm programs or other apps. You will need to duplicate this slide for different pages/actions in your app.



What the user sees when they select one of the images on the previous slide:

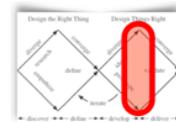


This slide of the paper prototype shows what the user sees in prototype 1 once they make a picture selection. Notice the use of labels to help depict how the app works.

**Prototype****2**

(Idea 2)

Use any drawing tool you want to create a "paper prototype", including taking screen shots of Elm programs or other apps. You will need to duplicate this slide for different pages/actions in your app.



Prototype2 made in PowerPoint, contains a lot of buttons and hovering options to see previews of each button.

**Prototype  
2 (Idea 2)**

**Instructions**

- You will see a set of colours and a set of names
- Select the colour you want to identify and then click on the name of the colour (or vice-versa)
- Example click on yellow and then
- You will receive points at the end along with a status report when you complete the level
- If you wish to learn more about Parkinson's disease, click on this icon

All data will be automatically sent to researchers to better improve Parkinson's Disease

If you don't wish to share your results, you can change the option in the settings

The instructions set down the rules and provide additional privacy information

Prototype

2

(Idea 2)

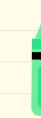
Match time!



Orange



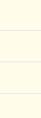
Green



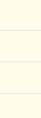
Blue



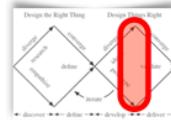
Yellow



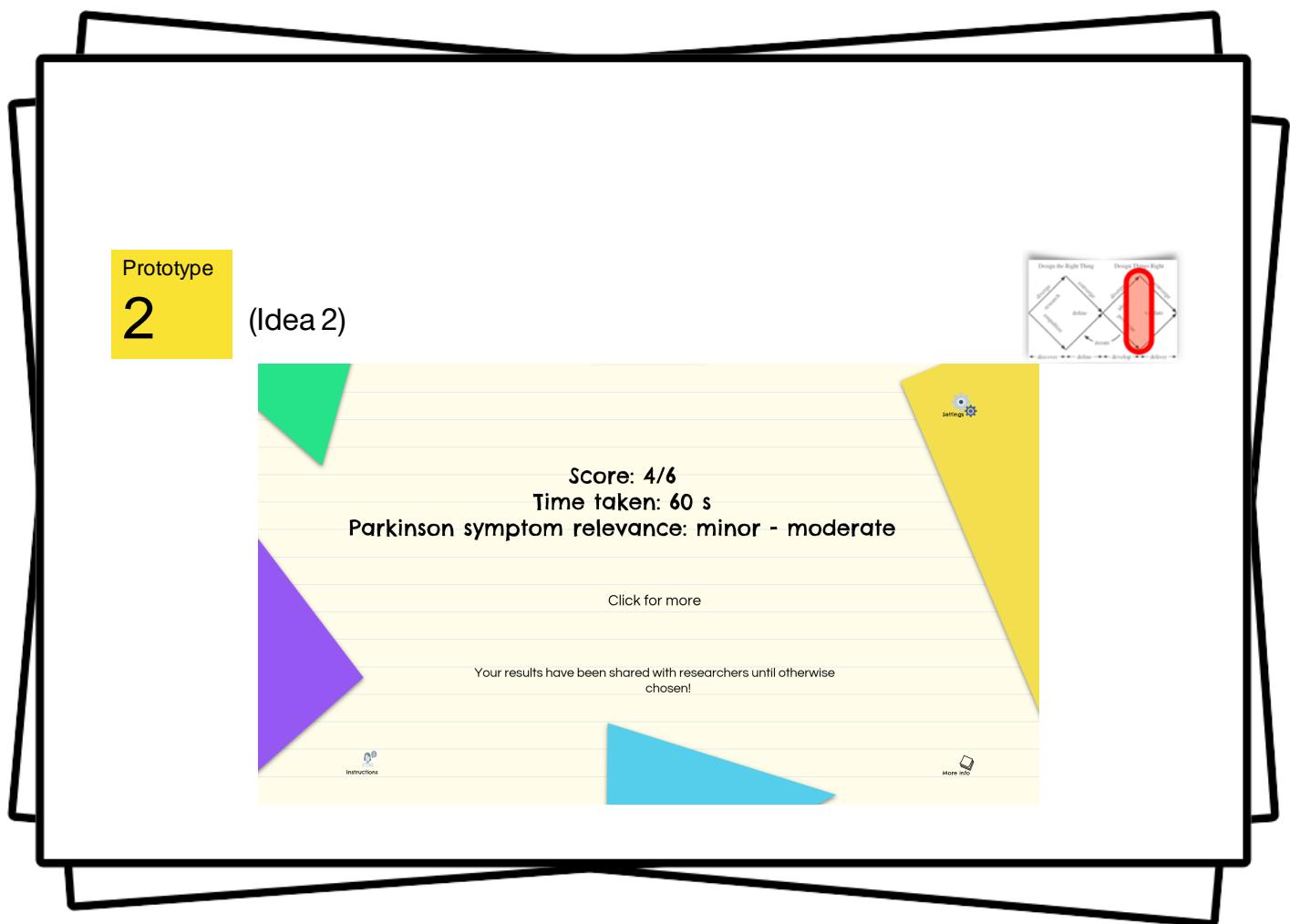
Purple



Red

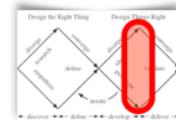


Once the game is complete, the results will show up for prototype2.



The results contain all the "results" but do not expand on how they were reached and how to improve said symptoms

Prototype  
**2**  
(Idea 2)



Settings include important ways to protect sensitive information and should be clearly highlighted

**+**

The fact that they can choose a picture that interests them
The game seems like it would be relaxing to play
Like that there are "instructions" for what colours to paint the picture with (the small version of the picture), since they know the final product will look good

**△**

Add a start page to give context to the game
Add instructions so people know how to play

**?**

How do you actually paint the picture in? When they drag the cursor does it paint everything in its path?
Can we share the completed painting in an easy way? Like emailing?

**Prototype 1**

**Feedback Grid:**

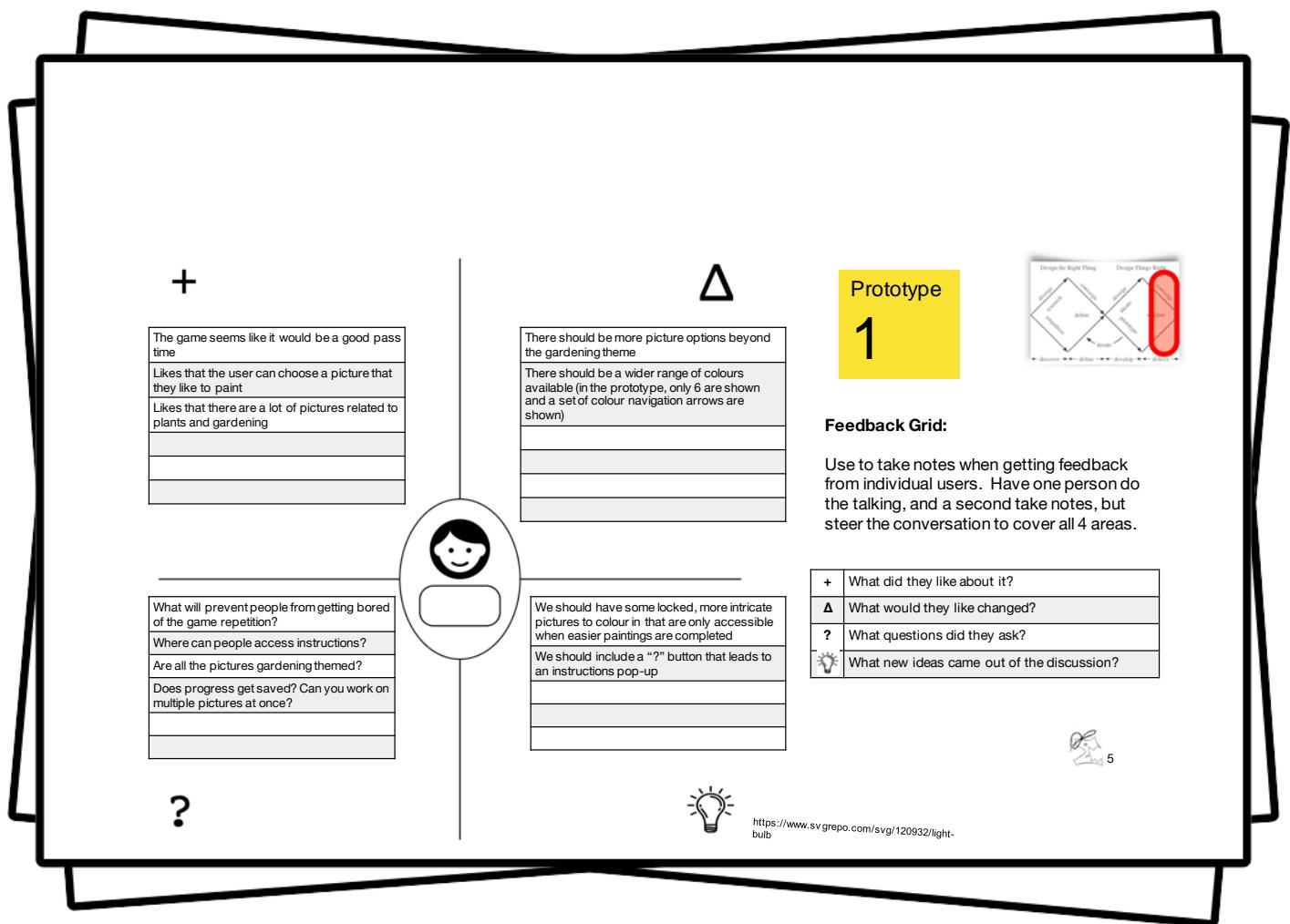
Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

+ What did they like about it?
△ What would they like changed?
? What questions did they ask?
💡 What new ideas came out of the discussion?

<https://www.svgrepo.com/svg/120932/light-bulb>

5

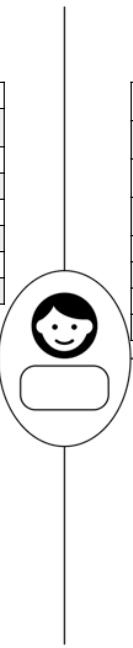
You should aim to interview the same people that you interviewed initially. You should observe the way the interviewee interacts with your prototype. You should ask them to treat the prototype as if it is an actual app; how they walk-through the app may reveal weaknesses like lack of instructions or missing transitions. You can ask your interviewee questions to prompt discussion, but just like before, be careful to not ask closed-ended questions. Also, ensure you cover all quadrants of the feedback grid prior to completion of your interview.



As you conduct interviews, similar themes may arise. For example, in the two interviews for prototype 1, the interviewees note that there is a lack of instruction on how to play the game. The re-occurrence of these themes becomes obvious in part because of the inherent organization of the feedback grid. As a final note, you may find it useful to do practice interviews and take raw notes, as was done in the initial interviews.

+

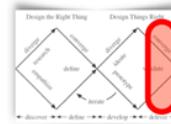
It seemed easy to use
The colour scheme wasn't too bad and not too distracting
Ways to contact the team and researchers
Privacy settings



△

The privacy option is only known if the instructions are read (make mandatory maybe)
Setting the main setting option should be the first thing one should get to choose
More levels to get more accurate and dependable results
More graphics could be added to make the game seem less monotonous and repetitive

## Prototype 2



### Feedback Grid:

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

+	What did they like about it?
△	What would they like changed?
?	What questions did they ask?
💡	What new ideas came out of the discussion?



<https://www.svgrep.com/svg/120932/light-bulb>

+

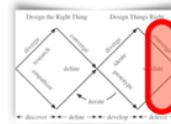
Overall game design was interesting and engaging
Options to contact researchers
Ways to choose own settings (sound, theme, etc)
Access to information about Parkinsons disease



△

More can be added to make the game more informative and helpful to deal with the symptoms
An emergency button could be added if something critical happens when playing
Could have group tournaments and challenges
The privacy option could be shown first

## Prototype 2



### Feedback Grid:

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

+	What did they like about it?
△	What would they like changed?
?	What questions did they ask?
💡	What new ideas came out of the discussion?



<https://www.svgrep.com/svg/120932/light-bulb>

# Pros and Cons

Prototype 1	Prototype 2
Pro: Users can choose a picture that interests them, widening the target user audience who may play this game.	Pro: The game seems easy to use and engaging.
Pro: According to feedback interviews, the game seems like it would be low-stress and relaxing to play.	Pro: The game uses the data of users and automatically sends it to researchers. Users have the privacy setting and control over changing that and protecting their data.
Con: There is no easy way to share completed work with others.	Con: The game is limited by common colour names and repetitive making it monotonous
Con: There is a lack of explicit instructions.	Con: Data which is collected is likely not reliable and not conclusive. It is irresponsible to lead users to believe that the app is a diagnostic tool

What did you learn from the two prototypes? Do you refine one or the other, synthesize them, or go in a new direction?

This table will help you explicitly compare the pros and cons of the two prototypes and determine whether one should be chosen over the other, or it may tell you what aspects of each prototype should be conserved in the development of a new, combined prototype. If both prototypes have significant cons compared to pros, you may opt to create a new prototype, shaped by the feedback and questions that arose during the feedback interviews. Based on this table, prototype 1 seems to be the more promising solution.

	Prototype 1	Prototype 2	Prototype 3
Pros	<p>Users can choose a picture that interests them, widening the target user audience who may play this game.</p> <p>According to feedback interviews, the game seems like it would be low-stress and relaxing to play.</p>	<p>The game uses the data of users and users have control over protecting their data</p> <p>According to feedback interviews, the game seems easy to use and engaging</p>	<p>We only started with 2 prototypes, so we can ignore this column.</p>
Cons	<p>There is no easy way to share completed work with others.</p> <p>There is a lack of explicit instructions.</p>	<p>The game is limited by common colour names and repetitive making it monotonous</p> <p>Data which is collected is likely not reliable and not conclusive. It is irresponsible to lead users to believe that the app is a diagnostic tool</p>	

What did you learn about the prototypes? Do you refine any, synthesize them, or go in a new direction?

You may choose to organize your table like a grid, as seen in this slide. This view may be superior to the table view when the number of prototypes being compared is larger. Either way you choose to visualize your feedback, the pros and cons should be traceable to the feedback grids (and interviews).

# Action Plan

What they said:	How we will improve:
Prototype 1 lacked clear game instructions.	We will add a start page with a game introduction and instructions, and a "?" button that will allow users to access the instructions at any time.
Prototype 1 has no way to easily share completed work with other people.	We will add a share button so that users can send their work (possibly to contacts on their device, their Facebook page, via email, etc.).
The picture topics are limited.	We will add more picture topics, each with a subset of related picture options.

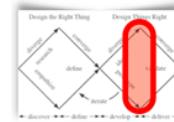
Duplicate the Feedback and Action Plan slides for as many iterations as you can fit in.

Since the pros and cons comparison of the two prototypes led to the decision to pursue the design of prototype 1 further, this action plan pertains to prototype 1 only. This action plan should be specific, and the "How we will improve" column should guide the development of the next prototype.

### Prototype

**3**

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



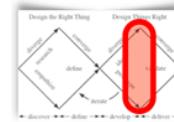
What the user sees when they open the Colour Creation game app:



In the present slide and next three slides, we see the implementation of the action plan seen in the previous slide. For example, the first item in the action plan (“Prototype 1 lacked clear game instructions”) is addressed in this slide; there is now a simple, 3-step instruction included on the first page of the app. All upcoming action plans should be as specific as this, such that the changes made in the prototype can be linked directly to the plan.

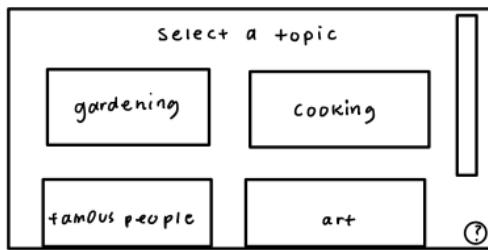
## Prototype

# 3



You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.

What the user sees after they hit the 'START' button: What the user sees when they hit the '?' button:

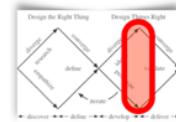


Remember to include some sort of indication of when the different app pages appear for the user. In this slide, we have page labels to indicate which button was clicked to result in that page.

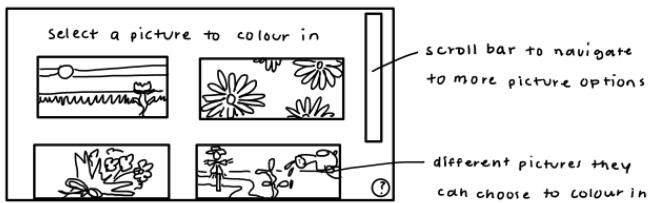
## Prototype

# 3

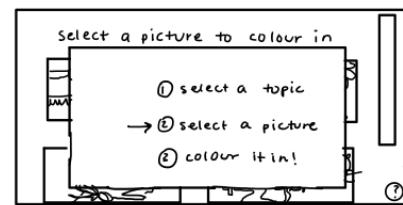
You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees after they select a topic:

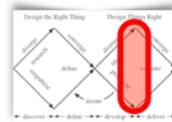


What the user sees when they hit the '?' button:



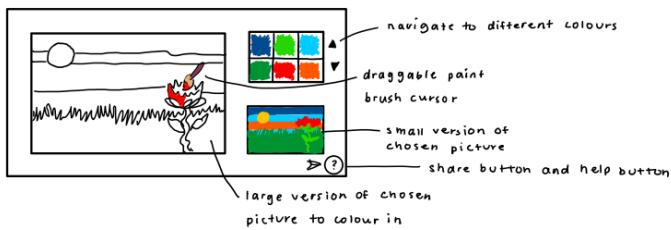
## Prototype

3

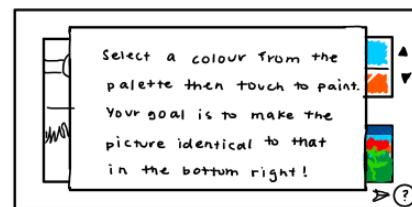


You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.

What the user sees after they select a picture:



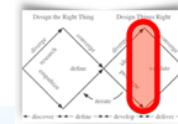
What the user sees when they hit the '?' button:



# Technical challenges

Challenge	What you will do: search resources, ask TAs for code example, build test program
Unsure of the mechanics required to share the completed work. Also don't know which platforms we can share to (e.g., are there privacy issues related to sharing work with device contacts?).	We will ask the TAs if they know who built the sharing feature on the IDE, and we will consult them if possible. We will do an online search and assign one member to building a test program.
Unsure of where we can obtain copyright free images to colour and reference.	We will do an online search to see if we can find a bank of copyright-free colouring pages related to our topics. If not, we will brainstorm an efficient way to create our own.
Unsure of touch-to-paint mechanics.	Touch-to-paint might be similar to the Among Us lab mechanics, so we should attempt implementation and consult the Among Us lab solution, related textbook chapters, and TA coach if needed.

When addressing technical challenges, try to be as exhaustive as you can. That way, you will limit the surprise road-blocks that you face. Further, ensure that your plan for addressing the challenge is logical and clear. Having a specific plan is conducive to the challenges being addressed efficiently. When formulating a plan, keep various resources in mind, including but not limited to teaching assistants, past assignments/projects, and the course textbook.



## DT Slides Presentation 1

When you present your DT Slides to another group, you will mostly use the slides you have already made for your own purposes, but you should add in a slide summarizing what you have learned so far, and what your challenges will be for the rest of the project.

*All throughout the process, we used the double diamond to assist us. We started off with research on the specific project areas and then determined our project area. We found that research was not just limited to the beginning of the project but complimentary with the design process. We realized how important interviewing of users was for understanding what users wanted, getting feedback on functionality, improving a product by discarding unnecessary things and adding desirable properties. We learnt that teamwork was essential since interviews, discussions, choosing our problems and solutions and improving our prototype relied on those. We learnt that we had to continuously improve our product and ourselves. The challenges we will face for the rest of the project will be the logistics behind sharing and constraints in using materials found online. Continuously improving our product and coming up with new prototypes will be challenging as well.*

We always teach Design Thinking to multiple teams, so there is always another team going through similar issues who you can talk to.

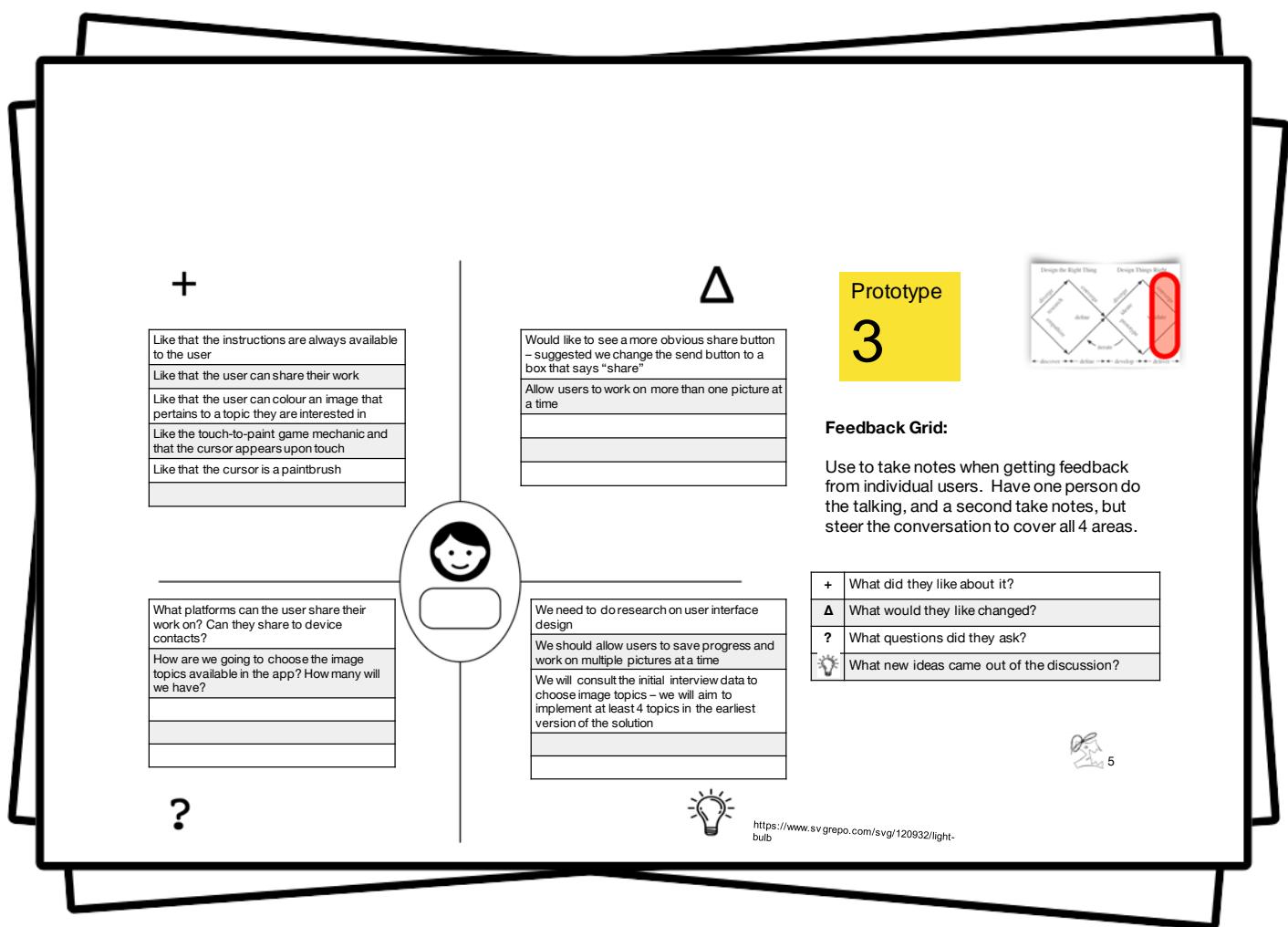
We also include check points, when every team prepares a presentation of their project so far, in order to get peer feedback—whether they think they are “ready” or not!

Add a summary of everything that you have learned so far and include challenges you anticipate. Don’t hesitate to add all that comes to mind since it helps you learn and be ready for future challenges. Include specifics about *your* project.

# Peer Feedback

What did audience question, where did explanations come out wrong, etc.	What will we do about it?
The audience questioned the use of the word "topic" and suggested that we prompt the user by asking something like "What would you like to colour in?" rather than "choose a topic".	We will change the "choose a topic" prompt to "What would you like to colour in?".
The audience questioned the user's motivation to keep playing the game (i.e., motivation to keep colouring picture after picture).	We will brainstorm ways to incorporate some sort of level system like standard video games have. This came up in our initial interviews and we had the idea to organize images in terms of complexity and lock more complex images until less complex images are complete.
The audience asked if users would be able to store their completed work to look back at. They suggested that it would be a good keepsake for their time spent on the app.	We will add a user gallery so that users can look back at completed work.

Taking notes during a presentation is not always possible. If this is the case, you should make notes immediately following your presentation, when the feedback from peers is still fresh in your mind. Remember to be specific in the feedback you record; this is important for project development and traceability.



This set of feedback interviews is just like the last set of interviews. Review optimal interview strategies and approaches (outlined in the Design Thinking Template chapter and previous interview slides in this example) prior to conducting your interviews in order to get the most out of them. One quick check you should do before ending your interview involves ensuring that all quadrants in the feedback grid have notes in them.

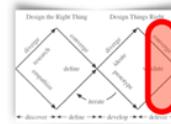
+

Like the touch-to-paint game mechanic
Like the variety of picture options available
Like the layout of the picture colouring page
Like the organization of choosing topics then pictures

△

Would like to see the colour palette look more like an actual palette upon app implementation
Would like to work on multiple projects simultaneously without losing progress on previous works

## Prototype 3



### Feedback Grid:

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

+	What did they like about it?
△	What would they like changed?
?	What questions did they ask?
💡	What new ideas came out of the discussion?

?



<https://www.svgrappo.com/svg/120932/light-bulb>



5

# Action Plan

What they said:	How we will improve:
The game seems repetitive and there is nothing to make people keep playing.	We will organize the pictures by level of complexity, and more complex pictures will be locked until easier pictures are painted. In all, this gives a reason for players to want to come back to the game.
The user might get bored of painting one picture at a time.	We will allow users to save progress and store a gallery of their completed and semi-completed pictures.
The send button is not obvious and users may overlook it.	We will include a pop-up that points to the share button at the start of colouring and when the picture is 90% complete.

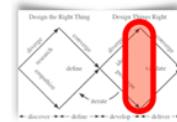
Duplicate the Feedback and Action Plan slides for as many iterations as you can fit in.

The process of iterating the prototype, getting feedback (either from peers or interviews), making an action plan, then developing a new prototype is central to the project development. Although the process is repetitive, try to reflect on each iteration to ensure that the next one does not have any of the same downfalls as the previous cycle. For example, if you noticed that your previous prototype cycle involved difficult-to-construct prototype because of an un-specific action plan, mostly due to the vague recording of interview data, then you should discuss strategies with your group to ensure that this does not happen in the next prototype cycle.

## Prototype

# 4

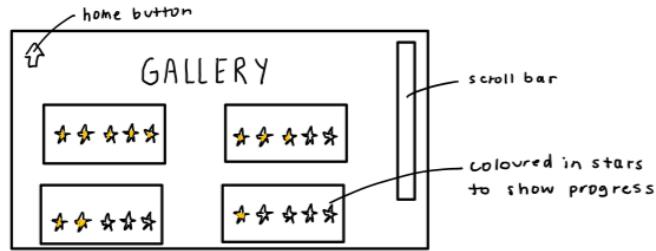
You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees when they open the app:



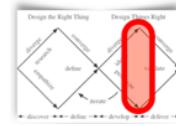
What the user sees when they click 'gallery':



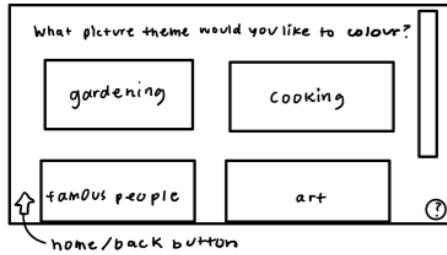
## Prototype

### 4

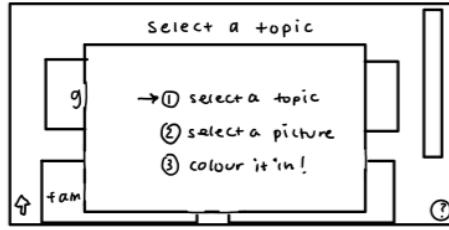
You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees when they click the 'START' button on the homepage:



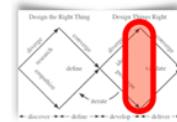
What the user sees when they hit the '?' button:



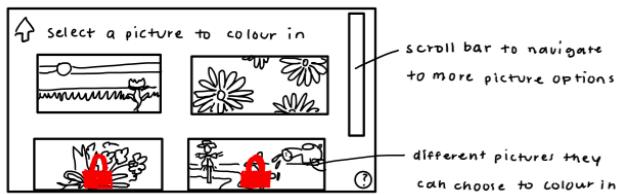
## Prototype

# 4

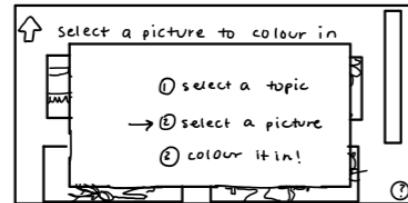
You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees after they select a topic:



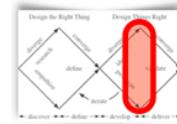
What the user sees when they hit the '?' button:



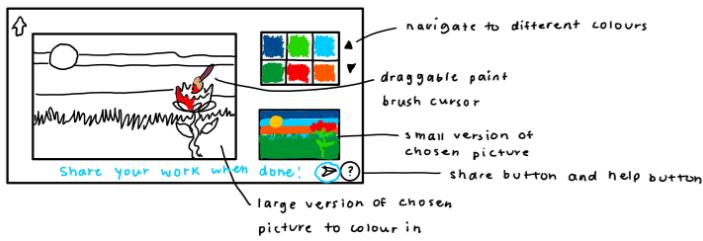
## Prototype

# 4

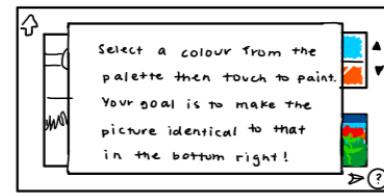
You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees after they select a picture:



What the user sees when they hit the '?' button:



# Risks

Main things which could go wrong and prevent you from completing a working app.	What you can do about it? (assign a second person, find more users, etc.)
Picture sharing mechanics are a new topic to all group members, including the person assigned to work on this feature.	After consulting a TA and learning more about how the share feature was implemented in the IDE, one group member realized that they had worked on something similar in the past. This group member will take over the implementation of this feature.
We have yet to implement the feature that will check if users have coloured in the picture correctly. Further, we have to investigate what tolerance level should be allowed for correct colour as people may colour slightly out of the lines.	The group member who was previously working on the share feature will now devote time to working on this feature. This group member will make a test program, which will be used to determine a colouring in the lines tolerance.
There are not that many copyright-free colouring pages online.	We will consult local artists to ask if they are willing to create simple colouring pages to support the game and cause. If time permits, we may build a program that creates line drawings out of copyright-free images (of which there is an abundant supply online).

At this point in the project, the solution is considerably developed. This slide is all about taking pre-emptive action. If you realize that there are ambiguities in your app, you should have a group discussion to resolve these obscurities, create an updated prototype, then access the risks associated with the project.

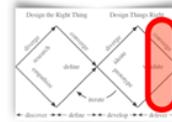
+

They liked that there is a gallery to view completed and current work  
 They liked that you could have multiple projects on the go  
 They liked that some images were locked, since this gives incentive for the users to keep playing  
 They liked that the instructions are accessible on every page

△

The users seemed to look around for too long for the home button – the home buttons should always be in the same corner  
 Would like an explicit explanation for the stars in the gallery so that users don't have to guess what they mean  
 Add app context: mention that the app helps with building a database for medical research

Prototype  
4



#### Feedback Grid:

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

What do the stars in the gallery mean? How is progress defined?  
 Where can users learn more about why the app was made?

?

We could make the home button's position uniform throughout all pages – the consistency will likely minimize user confusion  
 We should include a pop-up or some equivalent to explain what the stars in the gallery mean

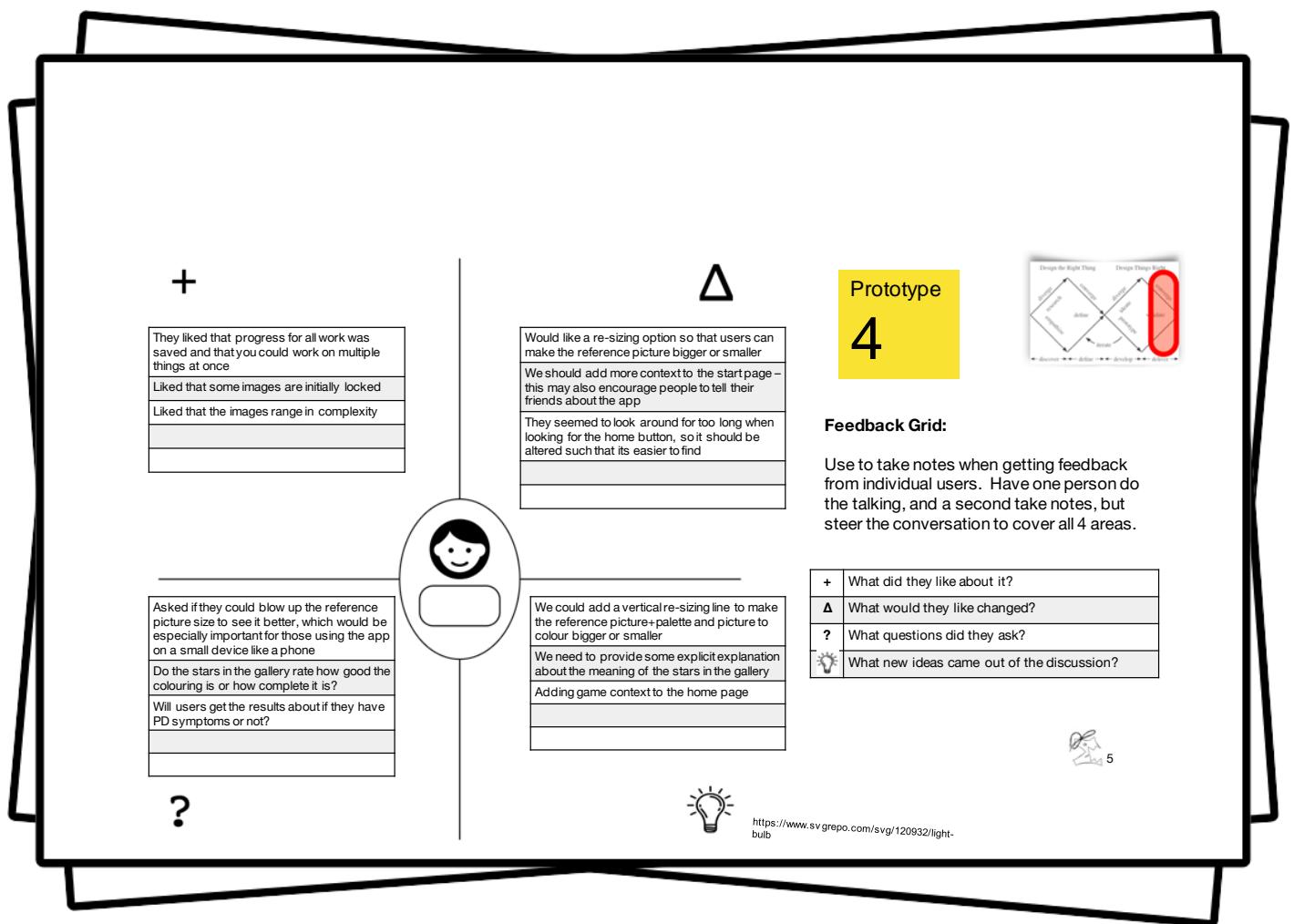
+	What did they like about it?
△	What would they like changed?
?	What questions did they ask?
💡	What new ideas came out of the discussion?



<https://www.svgrappo.com/svg/120932/light-bulb>



5



Note that not all user feedback will lead to changes. In this interview, the user questions whether or not they will receive feedback on if they present with PD symptoms or not. This issue arose in prototype 2 – we can't tell someone they have PD based on one symptom as it would be an ill-informed diagnosis.

# Action Plan

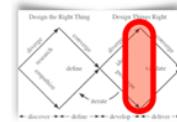
What they said:	How we will improve:
The app lacks visual appeal.	We will do some research into user interface design. One quick search reveals how rounded corners appear friendlier and more welcoming. We may also make the palette colours look like paint.
The home button was not easily found on every page.	We will make the home button in the same corner on every page.
It was unclear what the stars in the gallery meant.	We will explain the stars progress tracker in a pop-up/instruction page on the gallery page.
Would like a resizing option to control reference picture and picture to colour size.	We will add re-sizing bars to the colouring screen layout.
Would like more context for the game available to users.	We will add some background information for the game to the homepage.

Duplicate the Feedback and Action Plan slides for as many iterations as you can fit in.

## Prototype

**5**

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees when they open the app:



What the user sees when they click 'gallery':

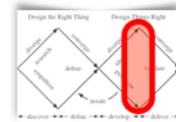


The gallery shows stars based on how much is coloured in; not how much is correct

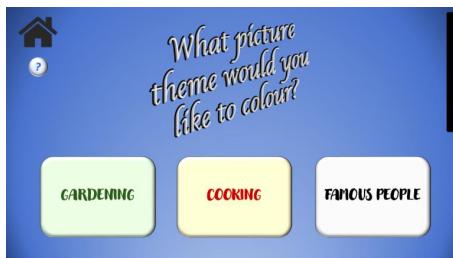
## Prototype

**5**

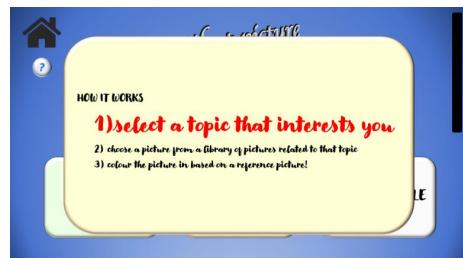
You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees when they click the 'START' button on the homepage:



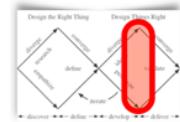
What the user sees when they hit the '?' button:



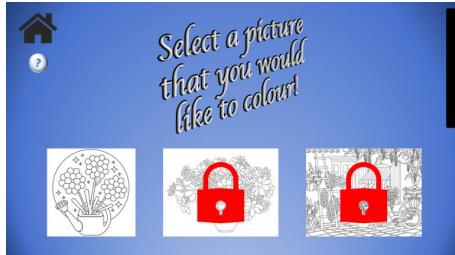
## Prototype

**5**

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees after they select a topic:



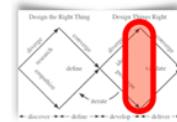
What the user sees when they hit the '?' button:



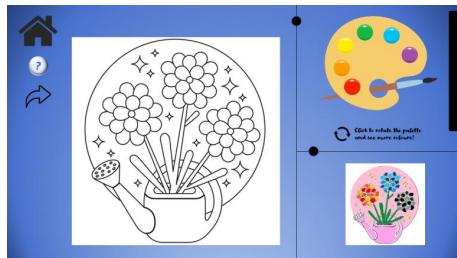
## Prototype

**5**

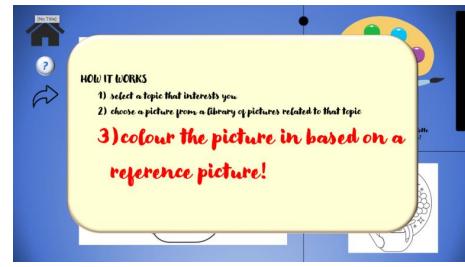
You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees after they select a picture:

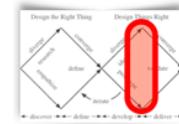


What the user sees when they hit the '?' button:

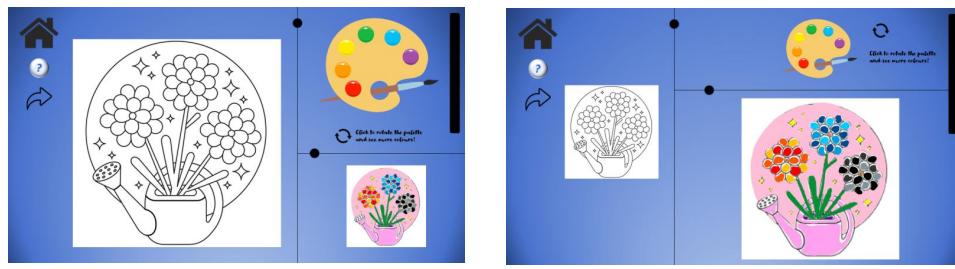


**Prototype****5**

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



The windows on the colouring screen can be re-sized by dragging the circles:



## DT Slides Presentation 1



When you present your DT Slides to another group, you will mostly use the slides you have already made for your own purposes, but you should add in a slide summarizing what you have learned so far, and what your challenges will be for the rest of the project.

*We learnt that we not only had to keep users in mind but also calculate the risks that came with the project. We learnt to weigh the pros and cons, assess the risks, feedback and use that to decide on our action plan. We were challenged with improving our product again and again but learnt to see things from a new angle and consider everything. We anticipate more challenges as we come closer to finalizing our product. How can we minimize risks and meet all requirements of our users? We also to make sure that our prototype is as detailed as possible and contains more working components than just paper prototypes.*

Once again assess what you have learned and the challenges you anticipate.

# Peer Feedback

What did audience question, where did explanations come out wrong, etc.	What will we do about it?
Unclear how the users can unlock more complex pictures to colour.	We will add text to explain that users must complete at least 1/3 of an image before the next image is unlocked. The final image will only be unlocked when all other images are complete.
Unclear how the windows can be re-sized when the user is painting a picture.	We will add some arrows on either side of the dragging circles to indicate that they move.
Right now progress is based on how much of the picture is painted. What if the user thinks they are done though? How does the user "submit" their work to signify that it is done?	We need to add a "complete" button to accommodate for this case. Although the work is not complete, the user thinking the work is complete still tells us something in terms of their colour acuity. We will also add a progress bar on the painting page to users can self-monitor progress.

+

Like the appearance of the app
Like that the windows in the colouring page can be re-sized
Like the instruction layout as it describes which step of the overall process the user is at
Like the context given on the homepage

△

Interviewee mentioned there was some ambiguity in the colour selection process; the app should be changed to include explicit instructions for this feature

Prototype  
5

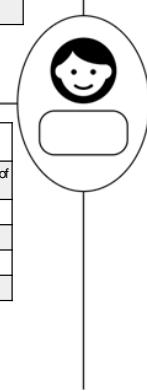


#### Feedback Grid:

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

+	What did they like about it?
△	What would they like changed?
?	What questions did they ask?
💡	What new ideas came out of the discussion?

?



<https://www.svgrepo.com/svg/120932/light-bulb>



5

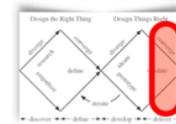
+

Like the palette for colours
Like the visual appeal of the app pages
Like the explanation about the game on the homepage as it will promote use of the app
Like the star system used in the gallery page

△

Would like to see some explicit instructions to support the re-sizing window function on the image painting page

Prototype  
5



#### Feedback Grid:

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

+	What did they like about it?
△	What would they like changed?
?	What questions did they ask?
💡	What new ideas came out of the discussion?

?



<https://www.svgrepo.com/svg/120932/light-bulb>



5

# Action Plan

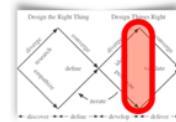
What they said:	How we will improve:
The way the user can go about re-sizing windows in the painting screen is not clear.	We will add arrows to either side of the dragging circles, so that the user understands that the circles can be dragged.
How the users can unlock pictures is unclear.	We will add instructions to the picture choosing page so that users understand why some pictures are locked, and how to unlock them.
Some users might believe they are done even if they technically aren't (i.e., if the picture isn't entirely coloured in).	We will add a progress bar at the top of the painting screen so that users can monitor progress. We will also add a "done" button for in case the user believes the progress tracker is wrong and they are actually done. This may be important in data analysis. Further, it will allow people to keep playing the game despite decrease in colour acuity.
Initial interaction with the app may be confusing for users. How do they start painting?	We will add instructions to the painting screen so that first time users know how to start (i.e., how to select a colour and how to paint).

Duplicate the Feedback and Action Plan slides for as many iterations as you can fit in.

## Prototype

# 6

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees when they open the app:



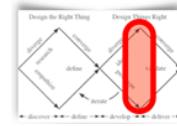
What the user sees when they click 'gallery':



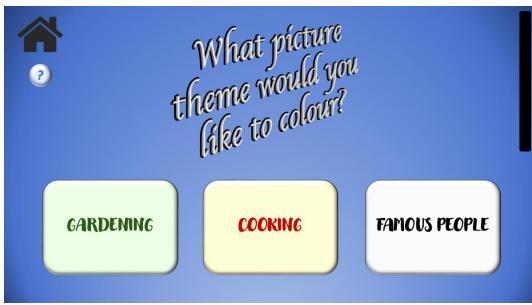
## Prototype

# 6

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees when they click the 'START' button on the homepage:



What the user sees when they hit the '?' button:

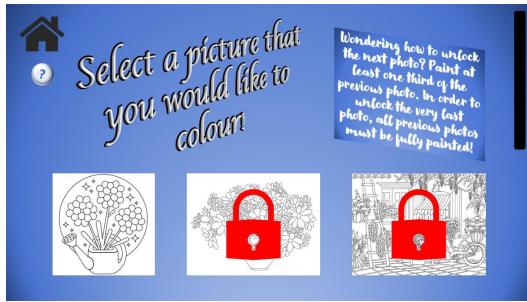


## Prototype

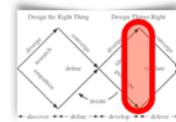
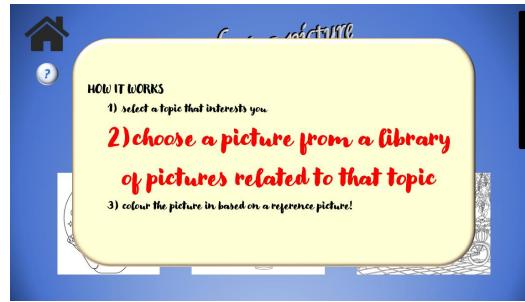
# 6

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.

What the user sees after they select a topic:



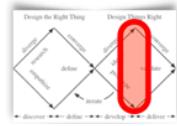
What the user sees when they hit the '?' button:



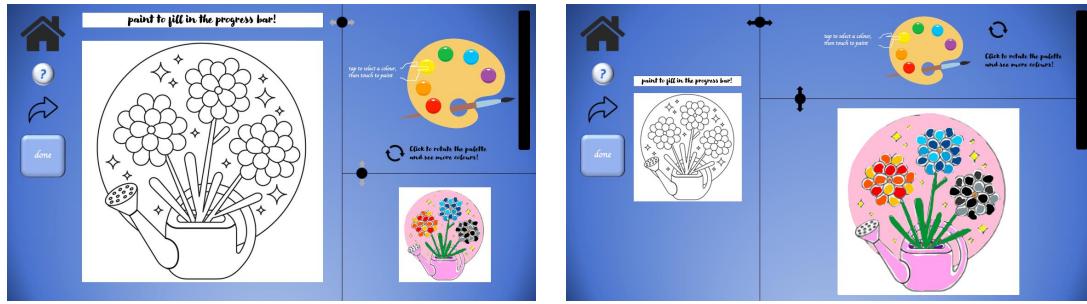
## Prototype

# 6

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



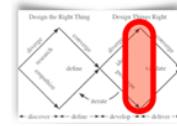
The windows on the colouring screen can be re-sized by dragging the circles:



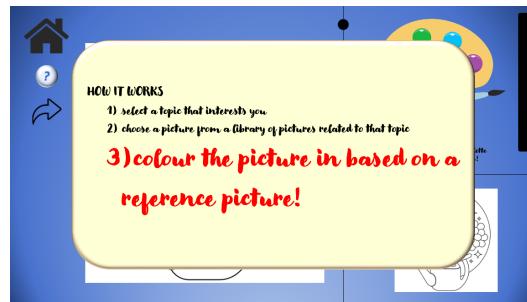
## Prototype

# 6

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



What the user sees when they hit the '?' button on the image colouring page:



**+**

Like the integration of instructions for re-sizing the windows on the image painting page
Like the integration of instructions for the paint palette and painting on the image painting page
Like the integration of the instructions regarding unlocking more pictures

**△**

For the prototype, they would like to see how the shared work will look when sent to others
Would like to see at least 6 picture topics available in version 1 of the app

**?**

How do you know your image themes will appeal to everyone in your target audience (given the size of the target audience?)

**?**

We should do more interviews with different people to observe which interests re-occur
We should do research to find topics that are interesting to most of the target population; our chosen topics are a reflection of the four people we interviewed initially

**Prototype 6**

**Feedback Grid:**

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

<b>+</b> What did they like about it?
<b>△</b> What would they like changed?
<b>?</b> What questions did they ask?
What new ideas came out of the discussion?

<https://www.svgrappo.com/svg/120932/light-bulb>

**+**

Like the progress tracking bar at the top of the image painting page
Like the user interface – the design is friendly and has a game-like appearance
Like the unlocking pictures premise and associated details and presentation of instructions

**Δ**

Would like to see more than 3 topics in the picture topic section; recommended based on appealing to a wider audience upon initial app launch

**?**

Are there more details regarding the share function?

**?**

We should focus on finalizing our share feature implementation since the target users seem to care a lot about it – the first version of the app should include this feature

**Prototype 6**

**Feedback Grid:**

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

<b>+</b> What did they like about it?
<b>Δ</b> What would they like changed?
<b>?</b> What questions did they ask?
<b>💡</b> What new ideas came out of the discussion?

<https://www.svgrepo.com/svg/120932/light-bulb>

5

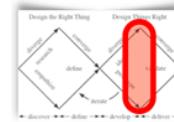
# Action Plan

What they said:	How we will improve:
The first version of the game should include at least 6 picture topics.	We will consult our original interview data to find at least 6 topics which appeal to the target audience. If we cannot find 6 re-occurring themes, we will consult online sources.
The share feature is important to target users	We will focus efforts on completing the share feature. We will follow the plan outlined previously (in the Risks slide).

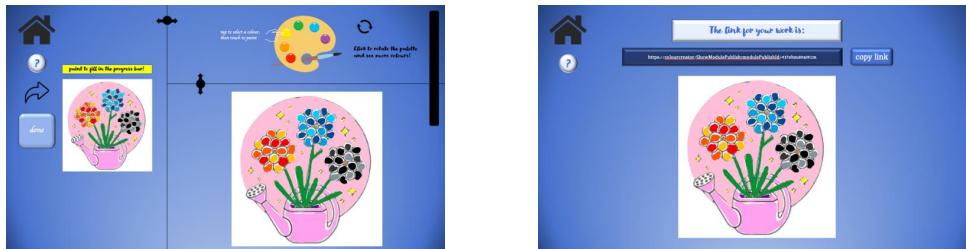
Duplicate the Feedback and Action Plan slides for as many iterations as you can fit in.

## Prototype 7

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



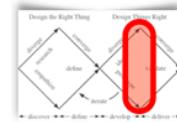
The windows on the colouring screen can be re-sized by dragging the circles:



In this prototype, only the pages with changes are shown. However, when performing interviews, you should include all pages in your app to ensure that the viewer (and potential user) is getting the full experience of your app.

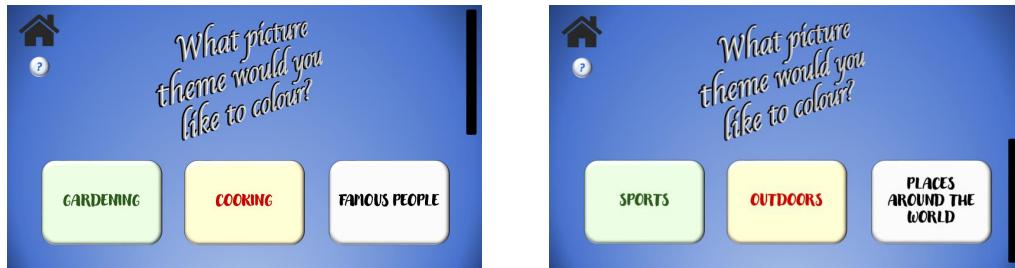
### Prototype

**7**

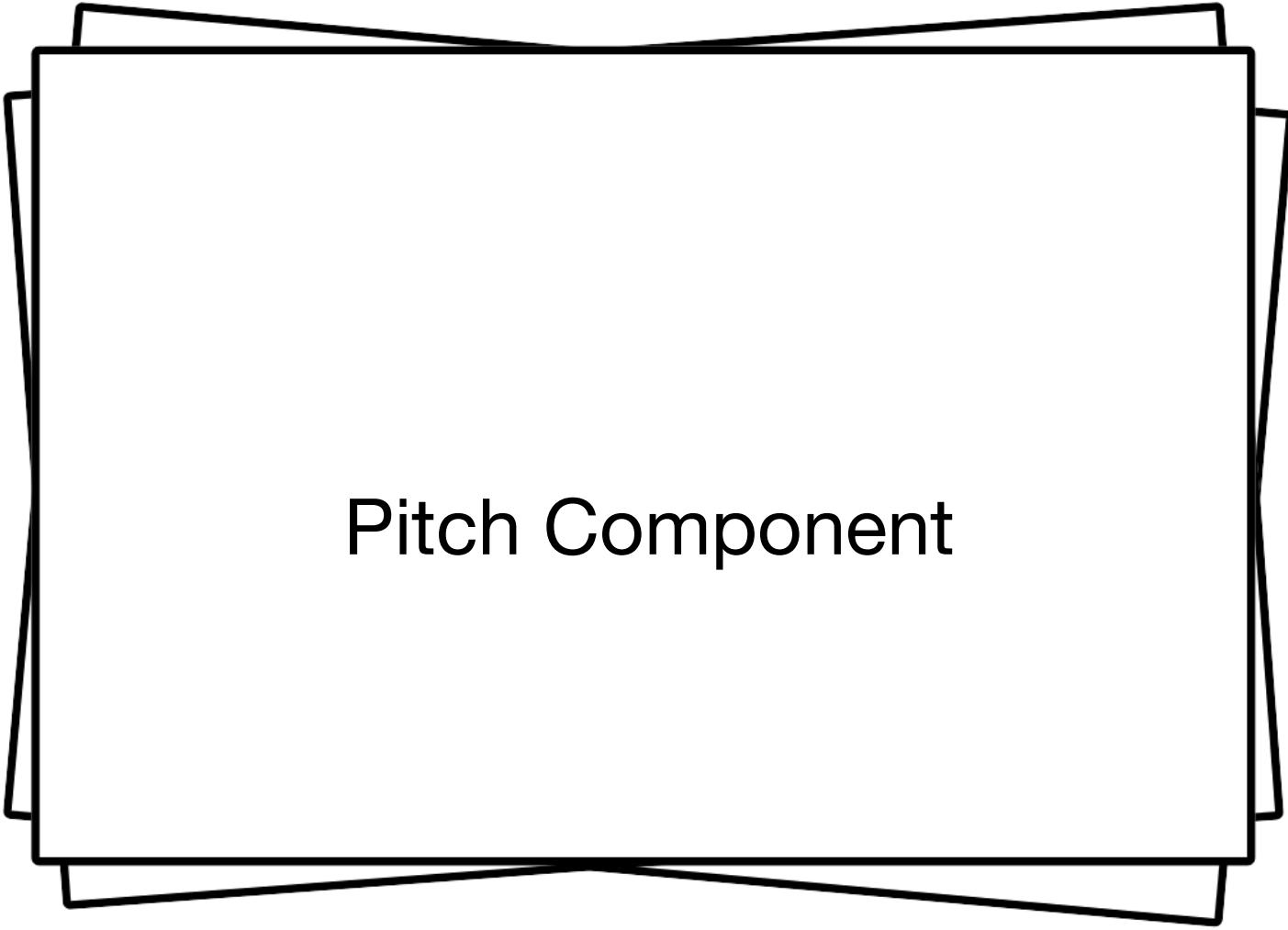


You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.

What the user sees when they click the 'START' button on the homepage (note the scroll bar):



In this example, we stopped prototyping at 7 iterations. Note that you are not limited to the number of iterations the template outlines; you will probably have more iterations than this example shows in your Math Visualizer project journey.



# Pitch Component

The pitch component slides that follow prompt you to address certain topics in your presentation, but you are not limited to only these topics. You should include whatever you deem important in your presentation. You might consider creating a separate PowerPoint for your presentation, with neater graphics and more visual appeal. Your pitch should convince the audience that the solution developed will be successful in the market – this means that the audience should be convinced that your solution directly and effectively solves the identified problem.

## Meet Bob

Tell us about your prototypical user.

**Our prototypical users are people that are most at risk of developing Parkinson's disease. According to our research, this includes adults that are of age 50 years or more.**

## Bob's Problem

Tell us their problem

There is insufficient data collected regarding the progression of PD symptoms, which prevents the development of techniques and technology that can detect the earliest presenting PD symptoms, necessary for the best possible prognosis.

## We are...

Who are you and why do you think you care about solving Bob's problem

We are first year computer science students at McMaster University. We have employed design thinking to understand the problem noted on the previous slide, from the points of view of potential app users and researchers. We are passionate about this issue and believe that a solution is very desirable to all stakeholders.

# Solution

Show how you solve the problem, with a voiceover of the app being used, but don't make it a walk-through of the app, or a tutorial, talk about how it solves the problem.

The app allows users to play a fun colouring game, while recording data relating to colour acuity. This data will contribute to a medical database relating to the onset and progression of PD symptoms in those who are at most risk of developing it.

As the slide mentions, it is important to explain how your app addresses the problem, and not just how the app works. One effective method includes doing a screen recording of your app, which would allow you to convey interactive features of your app, while doing a voiceover.

# What we learned from DT

Duplicate your actual DT slides which help you explain your points, and talk over them.

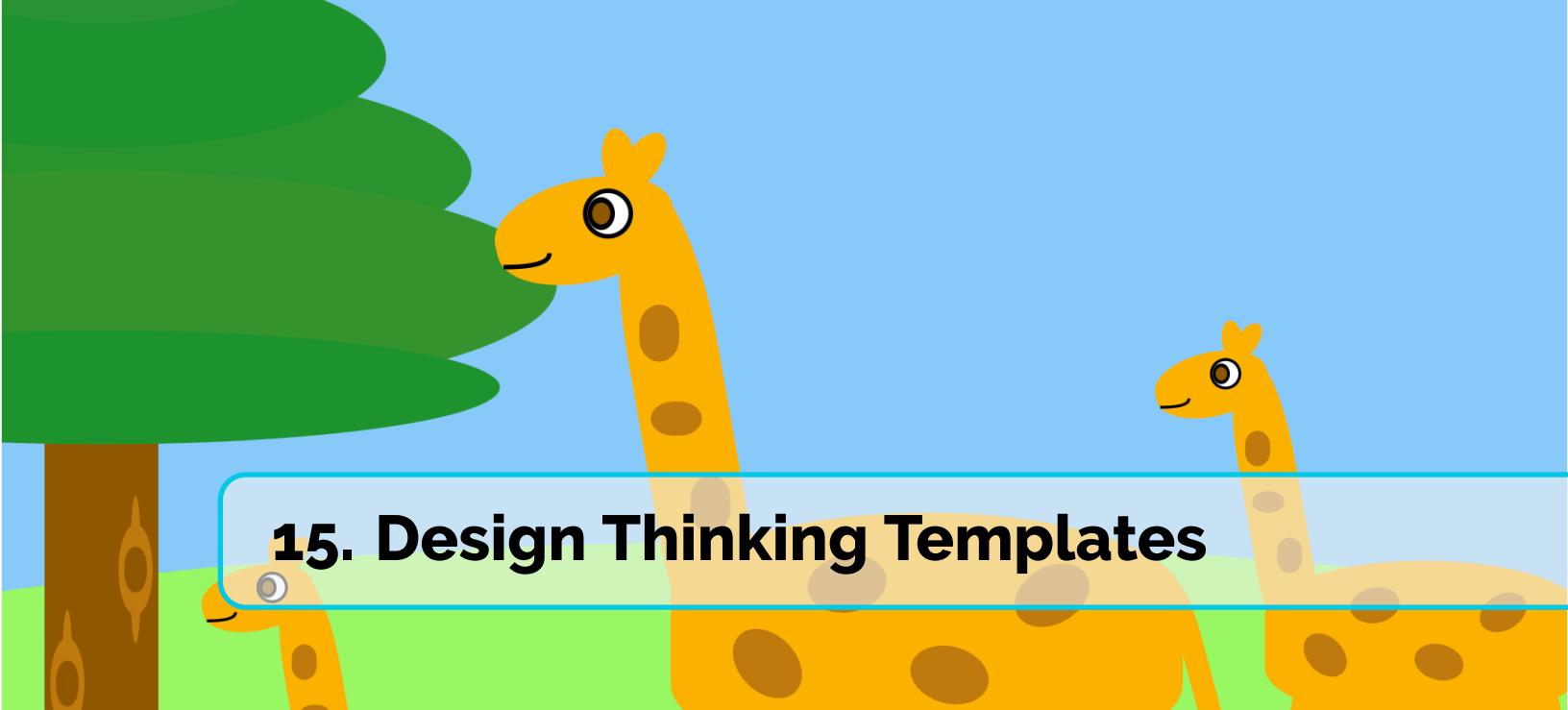
To save some space in the textbook, we won't duplicate the DT slides. You should ensure that you address the data collected in the initial interview. The audience is likely to lose interest if you talk through every quadrant of all initial interviews, so consider presenting a summary of the collected data, including re-occurring themes. You should address your problem generation process and associated How Might We process, including justification for how you decided on the final problem to focus on. You should speak to the solution process and statement in a similar fashion. Finally, talk about your prototypes, the feedback you received, and how that feedback resulted in prototype changes. The audience should understand why changes were made, and they should agree that the changes you made are justified.

# Next steps

## What we plan to do to improve the app we have

To begin, the plan to finish the implementation of our app; currently our minimum viable prototype has working game mechanics, but they are not integrated with one another. We plan to do an early release of the Elm app to the interviewees that we relied on to develop this app and to at least 4 other people that align with our defined target users. Having people use version 1 of the app, as they would use any other app, will provide us with valuable insight as to how the app is perceived. We will allow users to have a few days with the app, and will request that they complete colouring at least 2 pictures. Following this, we will perform 8 interviews (one with each of the pilot users). We will examine the interview data to extract themes, much like we did throughout this process so far.

Try to be as specific as possible as you outline the next steps for your product. Ask yourself what stands in the way of your current app and market success.



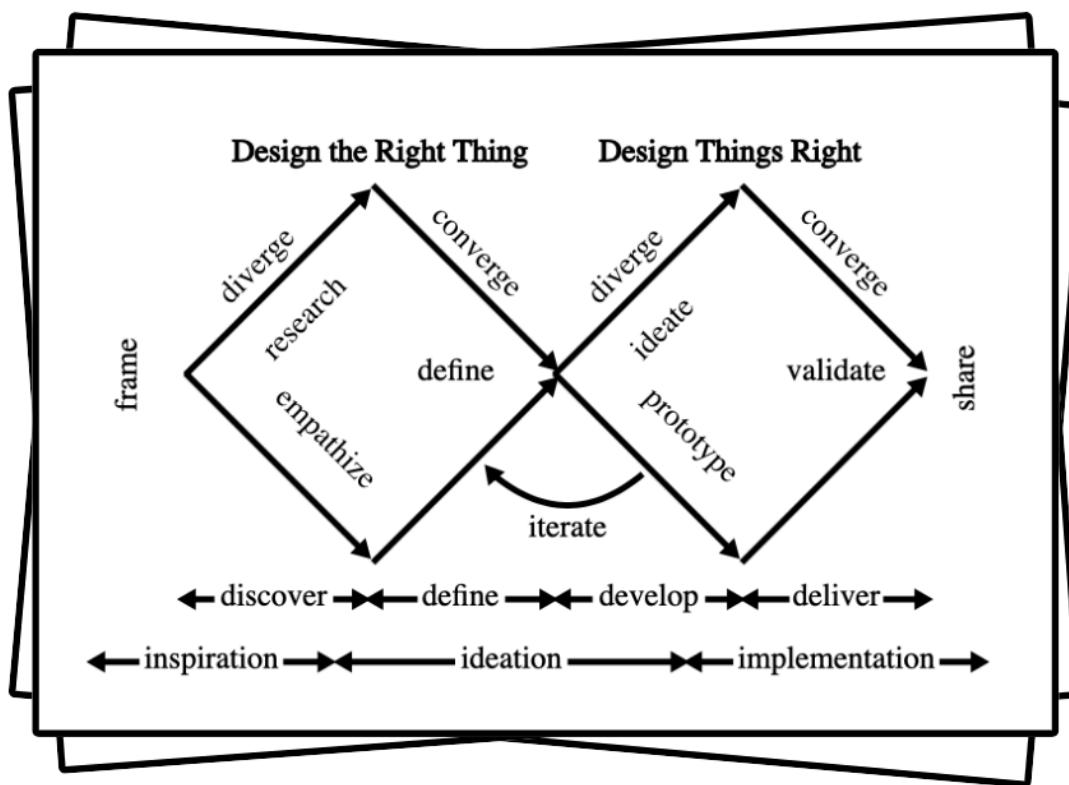
## 15. Design Thinking Templates

You have learned a lot about decision-making, and the invention of Design Thinking. One of the things you have learned is that human cognition has inherent limits, and processes need to be designed to work within those limits. We have put this principle into practice by creating slides with fillable tables and other editable elements to serve as templates for following the DT process—at least our version of it.

As much as possible, we have collected all the information you need for one step of the process on a single slide, so that you can focus on taking one step at a time. Through experience, we have made many improvements to the templates, and we continue to make modifications for particular camps, in-class workshops, and our own development. You can copy (and modify) the slides<sup>1</sup> in either Google or Microsoft format.

---

<sup>1</sup>See Avenue/Week3/Lab.



- The Double Diamond is a visual representation of the design thinking process
- It features two phases of divergence and two phases of convergence
- The first diamond focuses on problem definition (designing the **right thing**) and the second focuses on the solution (designing the **thing right**)

Created by the British Design Council, the Double Diamond is a visual representation of the design thinking process<sup>2</sup>. Although it oversimplifies the design process, it is a good starting point for those who are new to design thinking. The Double Diamond features two regions of divergence, and two regions of convergence. The divergence phases involve research and exploration while the convergence phases involve narrowing down ideas and determining which ideas are most important to the problem being addressed. The divergence and convergence phases are not a matter of climbing up and down a figurative mountain, respectively, but rather players in a game going off in different directions then agreeing to meet each other at some middle point.

In the divergence phase, different schools of ideas may be explored via things like an online search, interviews, or reading books. Information being collected is not consistent with a pre-defined goal. However, the broad question “How can I help people?” can be used to guide your research. Although this phase involves exploring different avenues, the researcher should remember that they won’t be able to solve every single problem encountered with one project. Attempting this would result in a solution that would likely not help anyone, as the solution would be too complicated to address specific problems. There

<sup>2</sup>See “The Double Diamond: A universally accepted depiction of the design process.” <https://www.designcouncil.org.uk/our-work/news-opinion/double-diamond-universally-accepted-depiction-design-process/> (accessed Jul. 1, 2022).

is no checklist that can indicate whether the researcher has diverged enough. In a course, a deadline may influence how much time is spent diverging and converging. In most cases, the researcher will have to decide when enough information has been collected to define and solve a problem. The slides presented in this chapter may help you navigate the design thinking process.

## 15.1 Starting the First Phase of Divergence

[to contents](#)

# Possible Project Areas



1. What do they need to know? Who are they? Who will you find them to interview?
2. Option 2
3. Option 3

- Narrow down the audience which you are trying to help
- Choose a group that you will have access to throughout the design thinking process, as you will require their engagement to collect interview data and solution feedback
- This step starts the initial divergence phase of the Double Diamond, where the designer brainstorms potential groups, and associated problems

Although open-mindedness is the heart of design thinking, it is helpful to narrow down the audience which you are trying to help. Examples of groups are third-year Computer Science students at your university and Grade 3 teachers in your city or a region with the same curriculum and students with the same background preparation. The template prompts you with guiding questions. It is important to choose a group that you will have access to throughout the Design Thinking process, as you will require their engagement to collect interview data and solution feedback (which may involve multiple meetings since you will have many solution iterations). This step in the design thinking process aligns with the

initial divergence phase of the Double Diamond, where you brainstorm potential groups, and associated problems. Examples of problems could be “not understanding fractions” or “Nash Equilibrium game theory”. Ensure that the problems you brainstorm are distinct; rewording the same idea would be counterproductive to the design thinking process. Although you will not pursue all brainstormed options, it is good to have a log for traceability. In addition, having multiple ideas can facilitate meaningful discussion and findings.

## 15.2 Researching Possible Project Areas

[to contents](#)

## Option 1 Research

1. What could we find out about option 1?
2. Is it something people find hard to learn?
3. Why do people need to know it? (Will they be motivated by applications?)
4. Is there a good visualization component?
5. How will we find people to interview?

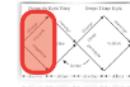
- Continue diverging by doing internet and library research
- Think about problems the identified groups may face

After identifying possible groups to empathize with, the initial divergence phase can be continued by doing preliminary research. Before going off in many directions, and interviewing many groups of people, you should think more about possible problems that the identified groups may face. For each possible project area, answer the questions in the slide above and then make an informed decision about which project area to pursue.

## 15.3 Choosing a Focus

[to contents](#)

# Our Focus



1. Target users:
2. General problem area:

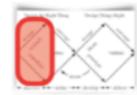
- Choose a target group and a problem area
- Keep your team on the same page: explicitly state your focus

Following reflection of the possible project areas and preliminary research, you should choose a group of target users and a general problem area. Although the previous slides are highly related to finding a group to focus on, it is important to be explicit in stating your group's focus as it will ensure that you and your team members are on the same page, both now and later in the project. It is very frustrating and hurts team morale when some work must be discarded because it does not fit into the whole, because not everyone had the same understanding of your focus.

## 15.4 Interview Preparation

[to contents](#)

### Questions to Ask



1. Q1
2. Put questions here to guide your interview.
3. Q3
4. Q4
5. Q5
6. Q6
7. Q7

- Prepare for the interview, so it doesn't seem rehearsed!
- Brainstorm questions to use as icebreakers, and pauses
- Cover different angles and possible problem areas
- Include open and closed questions
- Include process questions

Now that a target user and general problem area have been identified, research in this divergence phase should be furthered by conducting interviews with the target group. To make the most of the interviews, you should come up with open-ended questions to ask your interviewees. While you should put a considerable amount of thought into the questions you ask, the question list should not be treated as a checklist to get through in the interview. This list of questions may help your interview get started or fill any awkward gaps in the conversation. During the interview, you should aim to make the interviewee comfortable by using a conversational tone. An advantage of a synchronous, person-to-person interview is that you can extract information from the interviewee that something like a Google form could not; take advantage of the fact that you can ask follow-up questions to clarify or learn more about certain answers, or when you think the interviewee has more to say, but isn't sure you want to hear it. In all, the interview should go in the direction that the interviewee takes it.

Also, note that data extracted from the interview should not be limited to what the interviewee says. Body language, gaps in conversation, and tone of voice can be used to augment

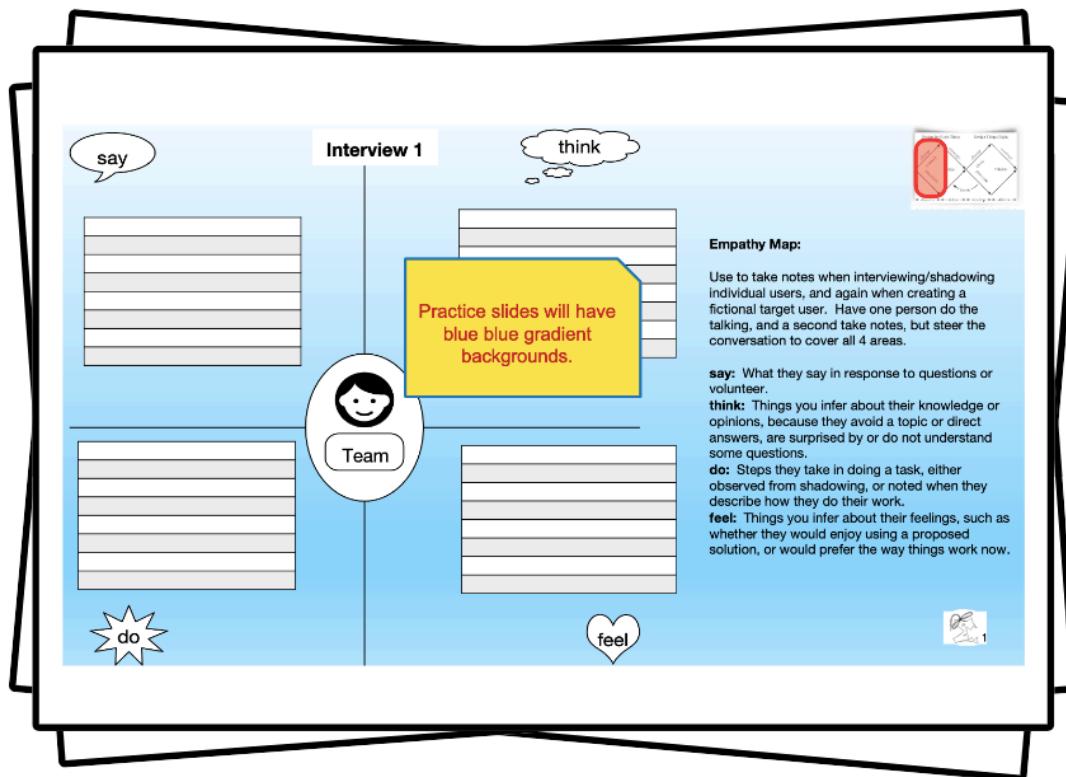
the spoken word. Implicit data can be as important as, or more important than, what is said out loud, and the way you interview should account for that. If the interviewee pauses before answering a question, it might be because they are uncomfortable about something, or maybe they think the question is too complicated to answer. They might tell you something that you did not expect to hear, and that might lead your interview in a new direction, and you might not get back to your question list, but that's okay. In fact, this is good since it means that you are learning about what the user is passionate about and what is important to them.

*Open and closed questions:* Normal conversations have open and closed questions. Closed questions have “yes” or “no” answers, while open questions are invitations for the interviewee to elaborate and add their own ideas.

*Process questions:* If you cannot observe your interviewer perform a task or resolve a problem, ask them to “walk you through the steps”. Asking questions in this way will cause the interviewee to include details which explain the context of the problem.

## 15.5 Practice Interview

[to contents](#)



- Practice to be a better interviewer
- Record only with permission
- Individual notetakers need raw notes or recordings
- Teams can fill in this Empathy Map live, guiding the interview to cover each quadrant

As you learned in the previous slide, interviewing isn't as easy as asking the list of questions you brainstormed before the interview. Being a conversational and engaging interviewer may come naturally to some, but it is important for all interviewers to practice interviewing. A good interviewer is essential to a good interview. You may have permission to record your interview and take notes after, or you might opt to take notes during the interview. If the latter, it would be helpful to have one group member interview and the other take notes. Above is one option that can be used for note-taking; it is called an Empathy Map and it helps you empathize with the person you are interviewing. The empathy map is split into quadrants which help you identify the interviewee's pain points. The reason for having four quadrants, "think", "say", "feel", and "do", is to prevent having an interview in which you only extract technical details, neglecting the user's feelings towards the topic being discussed. The danger of only recording technical details from the interview is that you are at risk of creating a solution that people do not actually care about, even if a technical problem is being solved for them.

The quadrants of the empathy map can also guide the style of the interview. In the "say" quadrant, you put what the interviewee explicitly says. Sometimes people don't say things explicitly, which is where the "think" quadrant comes into play. Typically (and unfortunately), the "do" quadrant is easily skipped. This quadrant can give insight into pain points that the interviewee cannot vocalize themselves, perhaps because they did not realize that it was a pain point or do not know how to describe the issue they are facing. To ensure that this quadrant is accounted for, you may ask the interviewee to walk you through a process. For example, you might ask them to walk you through how they approach a factoring problem. From their answer, you should aim to understand what they think they are supposed to do and what they think is hard about it. You should note what they explain in detail, what they skim over, their conversational tone, and, if applicable, where they make a mistake or struggle, and why.

Although the Empathy Map is useful for obtaining a holistic understanding of the interviewee, you may opt for a different method of note-taking. One such method involves taking raw notes, where points are not sorted into quadrants as they are in the Empathy Map. This method may be easier if you are alone in conducting the interview because you must both ask questions and take notes. If this is the case, it would be useful for you to transfer notes to an Empathy Map post-interview, to ensure that you are as close as possible to having a complete understanding of the interviewee. The best of all cases would be to have one group member ask questions during the interview, another take notes in the Empathy Map, and another take raw notes.

## 15.6 Evaluation of the Practice Interview

[to contents](#)

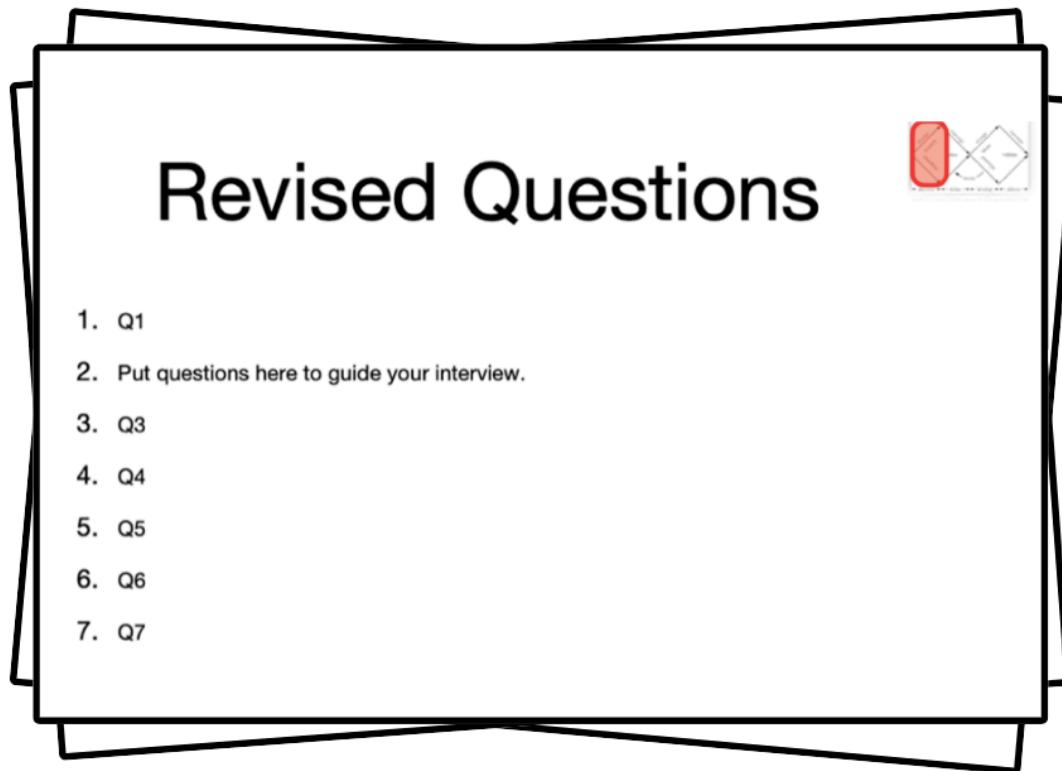
**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.

Did you...	Somewhat	Yes	Definitely!
set up interview, giving context?			
make them comfortable?			
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never	Sometimes	Always

- Reflection accelerates learning
- Reflect on your team's interviewing skills intentionally and constructively
- If group size permits, have one person focus solely on how the interview is being conducted and how to improve

Evaluation of the practice interview allows you to reflect on an interviewer's interviewing skills intentionally and constructively. The evaluation grid above should be used to critique and resolve any issues or shortcomings of the planned interview before real interviews are conducted. The evaluation could be performed as a self-evaluation or a peer-evaluation. If group size permits, having one person focus solely on how the interview is being conducted would be ideal as they might notice things that someone who is multi-tasking may not (i.e., you are more likely to miss conversational cues if you are both observing the interview and taking notes). Commonly noted on practice interview evaluations is the absence of a conversational tone in the interview. For example, perhaps there were multiple opportunities for follow-up questions which were missed because the interviewer was too focused on asking the questions they prepared, in order.

## 15.7 Revising Interview Questions

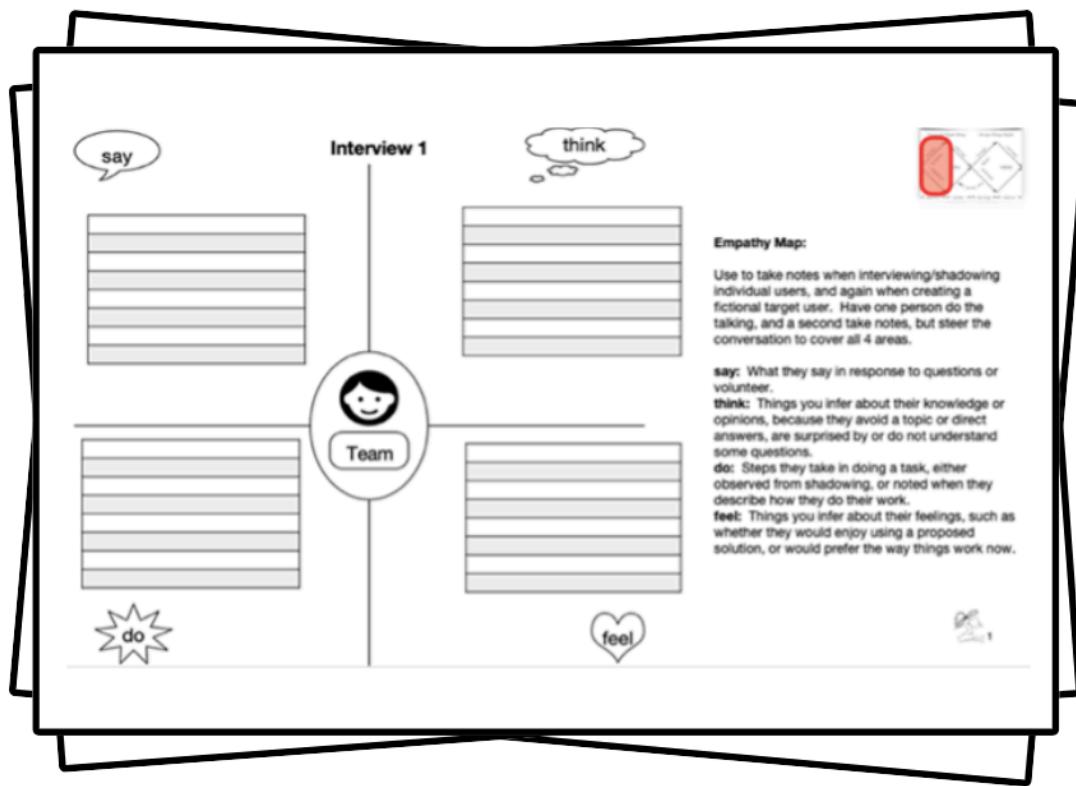
[to contents](#)

1. Q1
2. Put questions here to guide your interview.
3. Q3
4. Q4
5. Q5
6. Q6
7. Q7

- Revise the interview questions after the practice interview

Upon reflecting on the practice interview, you might have noticed that some questions were poorly worded or sounded awkward in actual conversation. You may also be missing key points. Time should be allotted to revising the interview questions before conducting real interviews.

## 15.8 Interviews and Interview Evaluations

[to contents](#)


**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.

Did you...	Somewhat	Yes	Definitely!
set up interview, giving context?			
make them comfortable?			
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never	Sometimes	Always

- Interview confidently, you are prepared!
- Make the interviewee feel at ease
- Roles: raw note taker, empathy mapper, observer, interviewer
- Keep growing through reflection

Based on your successes and shortcomings from your practice interviews, perform your first real interviews with members of your target audience. Ideally, you will have one person to ask interview questions, one person to take notes in the Empathy Map, one person to take raw notes, and another to observe the interview. Although these are not practice interviews, you should do an interview evaluation after each interview, and improve anything you can for every subsequent interview. In all, these interviews will serve as vehicles of data in the initial divergence phase of the Double Diamond. You should aim to interview as many people as you can, but this can be context dependant. In a course setting, you should aim to have at least a one-to-one interviewee-to-group member ratio.

## 15.9 Pivot or Not?

[to contents](#)

# Pivot or Not?



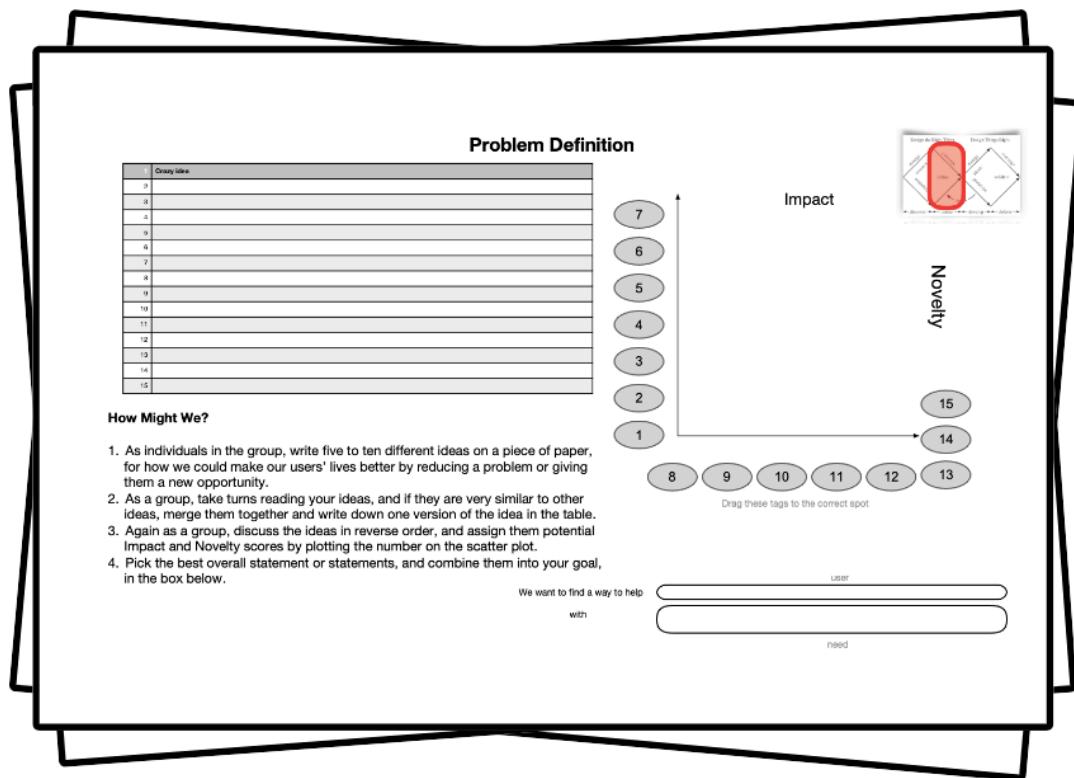
1. Was it easy to find potential users?
2. Is there commonality among the user problems?
3. Were interviewees excited about it?
4. Were we excited about it?
  
5. If you pivot to another idea, copy your Q and Empathy map slides and start catching up. ☺

- First contact with reality
- Did the interviewees care?
- Or do you need to pivot?

Following interviews, you might discover that no one in the target audience is invested in the problem area that they were interviewed about. If they do not care for the issue,

they are not likely to care about and thoughtfully test solution prototypes, culminating in a poorly designed solution and subsequently poor solution success. Pivoting may involve slightly reformulating to completely reinventing the project option that the design thinking journey is centered about. The questions in the slide above can help you and your group decide whether a pivot is necessary.

## 15.10 Problem Definition

[to contents](#)


- Brainstorm problems without filtering
- All problems come from a pain point or need to be identified in the interviews
- The How Might We (HMW) statement is the point of convergence

Once a problem area has been chosen, and shows promise based on the Pivot or Not analysis, specific problems should be extracted from the interview data. The importance of a good notetaking system is emphasized in this stage; all brainstormed problems should be linkable to a pain point or a need identified in the interviews, since we want to target a problem that we know people care about, not something we think people might care about. All problems should be written down, no matter what you believe the novelty or impact would be if a solution was produced. It is important to not discount any brainstormed problems as they may lead to discussions and the generation of new problem ideas. After the brainstorm, we move into the second part of the first diamond and begin converging to define one specific problem to tackle. We are explicit in this convergence by writing a How Might We (HMW) statement, which takes on the form “we want to find a way to help [user]

with [need]”. To assist in choosing one of your brainstormed problems, or choosing a few to combine, you may adopt a variety of methods. One such method involves plotting the various problems on a graph, with impact and novelty on the axes. Novelty refers to how new the problem is (and if the problem has been solved already) and impact refers to how the target user is affected by the problem. There is no scale on the graph. As such all plotting is relative, necessitating group discussion for plotting that reflects the thoughts of all group members and interviewees. It is normal that new problems are generated at this stage, and they should be added to the list and plotted like the others.

## 15.11 Symptom or Disease?

[to contents](#)

### Q&A



1. Write down questions raised by the HMW process. For example, if you identified fraction arithmetic as an area your users find confusing, what grade is it covered in Ontario's math curriculum? If PDEs cause your users trouble, what are they?

- You have a problem
- Before you try to solve it, find out
  - What other people know about it
  - What science can tell us about it
  - What other solutions may exist
- Are you addressing a root problem or a symptom

Before you continue with the design thinking process, you should take some time to reflect on the HMW statement generated by your group. Specifically, you should think about any questions that arose during the process that led to the HMW statement. Use these questions to guide research so that you can learn more about the chosen problem. This research contrasts the research you did when generating ideas for possible project areas, because it

is specific to the chosen problem. For example, you may have chosen ‘math’ as a possible project area, and subsequently ‘Ontario elementary school children struggling with fractions’ as the specific problem. In this case, you might consider consulting the Ontario curriculum to see what is said about fractions, and the grades that align with certain learning outcomes regarding fractions. In all, your research should help you become well-informed about the problem before a solution is developed. It is important to consider your problem from different perspectives; you should think analytically, creatively, and comparatively, and consult your group, target users, and experts in the area.

Keep a good log of your research. It is important to know where certain ideas came from; being explicit can help make sure that you do not lose sight of why you make certain choices, which is essential to designing a successful solution.

This research should help you make sure that the problem that you are tackling is actually the problem that users are struggling with or if there is an underlying problem that is causing the pain point. This is analogous to thinking about if you are tackling the symptom or disease. Sometimes you cannot solve the underlying problem. However, being informed about it is essential to solution development.

To help in finding out if you are tackling the right problem, you could consider asking yourself “Why?” five times: Why does your chosen problem exist? What causes that underlying problem? Is there a problem that causes the problem that causes your underlying problem?

An example of targeting the symptom rather than disease can be illustrated using a story about a car manufacturing line. Say that a car manufacturing company is producing cars that have mufflers that keep falling off. They could solve this problem by using tape to secure the muffler, but it is likely that there is a deeper issue (disease) in the production line causing this issue (symptom). Perhaps there is a bent piece of equipment which causes a machine downstream to wiggle, and the wiggling causes misaligned screws, and those misaligned screws cause the mufflers to be inadequately attached to the car. In this case, it would be more efficient to fix the bent piece of equipment, since the intermediate problems could cause other problems later which would necessitate the development of another solution, requiring time, money, and other resources.

This reflection may result in a pivot, entailing a minor rephrasing of the HMW statement or alteration of the entire statement. This might be frustrating, but it is important to make these changes relatively early in the design thinking process since you do not want to create a solution that no one cares about.

## 15.12 Solution Ideation

[to contents](#)

**Solutions**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The scatter plot shows Desirability on the vertical axis and Feasibility on the horizontal axis. A grid is overlaid on the plot area. A red oval highlights a specific point in the upper-right quadrant, indicating a high-desirability and high-feasibility solution.

Desirability

Feasibility

Ideation:

- As individuals in the group, write five to ten different solutions on a piece of paper.
- Include an idea that costs less than \$10 and one costlier. You can include one using an app, and one without an app.
- As a group take turns reading our ideas, and if they are very similar - merge ideas, and if they are very similar - merge them together and write down one version of the idea in the table.
- Again as a group, discuss the ideas in reverse order, and assign the Desirability and Feasibility scores by plotting the number on the scatter plot.
- Pick the best overall idea, and combine them into your goal, in the box below.

Drag these tags to the correct spot

**Game Generator**

	Color & Contrast Problems (1)	Emotional Effects (2)	Coordination (3)	Memory (4)	Balance (5)
Simple and Fun (A)	Mixing colors to determine what color will you get	Every time you open the app, there is a question which say's "How r u feeling today?" Or more than 1 question	Draw lines or a pic (if not tremors)	Track how much time it takes for puzzle to complete	Activity tracing # of Steps
Nature (B)	Color by # Pic what image to color	Nature meditation check-in, nature themed game idea	Plant plants in certain areas and then water and grow it	Memory game with various plants mix and match cards	How much walking in nature - Self check-in
Educational (C)	Telling the user various color patterns (complementary, tertiary) and then getting the user to pick different color patterns	Slideshow related to SEL. Then after you read the slides, there is a mini question-are about it to see if you have learned stuff.	Have the user trace a certain image/plant and then have audio overlay that would be able to give a description/explanation of the	Slideshow related to nature. Then after you read the slides, there is a mini question-are about it to see if you have learned stuff.	Teaches you how to do balance simple exercises like walking in a straight line.

The scatter plot shows Desirability on the vertical axis and Feasibility on the horizontal axis. A grid is overlaid on the plot area. A red oval highlights a specific point in the upper-right quadrant, indicating a high-desirability and high-feasibility game idea.

- Solutions answer the HMW statement
- Write down all solutions no matter how 'way out' they seem
- Plot solutions on the Desirability/Feasibility graph
- Spirited discussion must lead to an agreed common problem statement

Solution ideation marks the start of the second diamond. The solutions brainstormed should center around the HMW statement (i.e., all solutions should address the same problem). Different methods may be employed to support solution ideation. One such method involves traditional brainstorming, in a list (see top slide above) or mind map form, or you might opt for a more structured approach, which we call a solution generator (for example, the "Game Generator" below it). The generator may be laid out like ours, where interests and symptoms are aligned along the left and top of a table addressing data collection of Parkinson's disease using phone apps. In this generator, there is a solution for each interest, symptom pair. Regardless of which method is used, you should be sure to come up with as many solutions as you can. Write down all solutions no matter how 'way out' they might seem. Through discussion, you might find that the idea is not as 'way out' as you thought, or maybe there is a component of the idea which you could take and use in your solution. Even if you cannot use it, looking at the problem through the lens of the 'way out' idea may help you understand the problem better.

Once solutions have been brainstormed, one specific and concise solution statement should be written down. Like the HMW statement, being explicit about this statement will help make sure that the group does not lose sight of what they aim to produce as a final deliverable of the design thinking process. The solution statement can involve one brainstormed solution or a combination of a few brainstormed solutions.

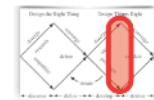
To aid in producing the solution statement, a grid similar to that used in the problem definition can be used. The axes are denoted by desirability, referring to how much the target user would enjoy the solution, and feasibility, referring to how likely it is that the solution can be developed and implemented.

## 15.13 Prototyping

[to contents](#)**Prototype****1**

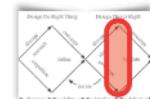
(Idea 1)

Use any drawing tool you want to create a "paper prototype", including taking screen shots of Elm programs or other apps. You will need to duplicate this slide for different pages/actions in your app.

**Prototype****2**

(Idea 2)

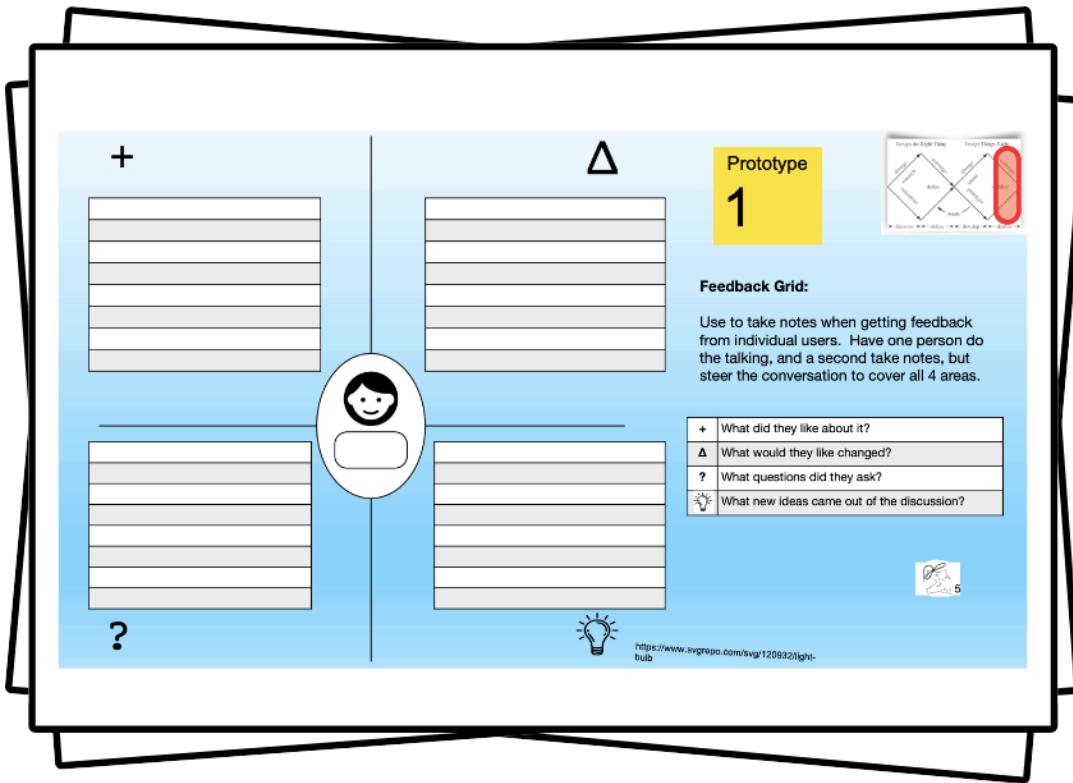
Use any drawing tool you want to create a "paper prototype", including taking screen shots of Elm programs or other apps. You will need to duplicate this slide for different pages/actions in your app.



- Prototypes are experiments to learn about the user
- Paper prototypes are quickly constructed, tested, and improved
- The paper prototype should convey solution functionality, not be a solution
- Do not start coding until you have stopped learning from paper prototypes

Although a solution statement has been constructed, there will be many possible implementations of the solution. Paper prototypes should be developed to assess these possible implementations. These may take on the form of drawings or sketches, or you might opt to use PowerPoint slides with textboxes and shapes. The paper prototype should convey solution functionality. For example, a software-based solution paper prototype might involve many slides which show what happens when different buttons are clicked. Although the paper prototype should convey the solution implementation idea, it should not require copious amounts of time and effort to develop. The paper prototype exists as an alternative to prototypes that are time and resource intensive for development. For example, coding an entire app as a prototype might take months to develop, for a potentially small return if major changes are required. In addition, the prototype developer might feel attached to the prototype if they spent a lot of time and effort developing it, putting them in danger of producing a final solution which does not solve user needs.

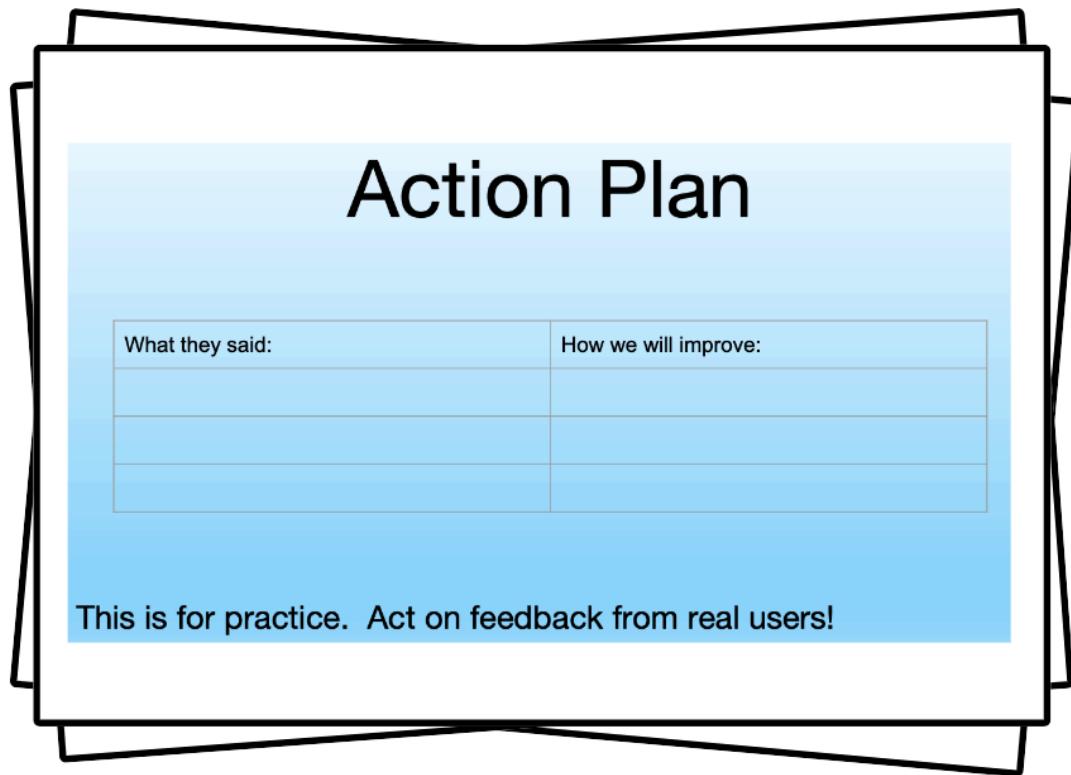
## 15.14 Feedback

[to contents](#)


- Prototypes are about learning
- Ask the user to pretend they are using your app
- Have them "think out loud"
- Use the grid to capture
  - what they like
  - what they want changed
  - what confused them
  - what new ideas were generated

You should seek feedback on your paper prototypes from your target users, ideally the same ones that were interviewed initially. Here, the importance of having a prototype that conveys functionality is emphasized. If possible, you should have the user interact with the prototype while you observe. This can reveal things about the prototype which might not be vocalized by the interviewee. For example, if they struggle to find where certain operations are in an app, that might indicate that the app buttons have poor visibility. Feedback should be collected from many angles, like how multiple angles were considered in the initial interviews. To help with this you might use a feedback grid, similar to the Empathy Map, as shown in the slide. There are four quadrants: what the user liked, what they would like changed, what questions they asked, and what new ideas came out of the discussion. Feedback should be collected for all developed papers and functional prototypes.

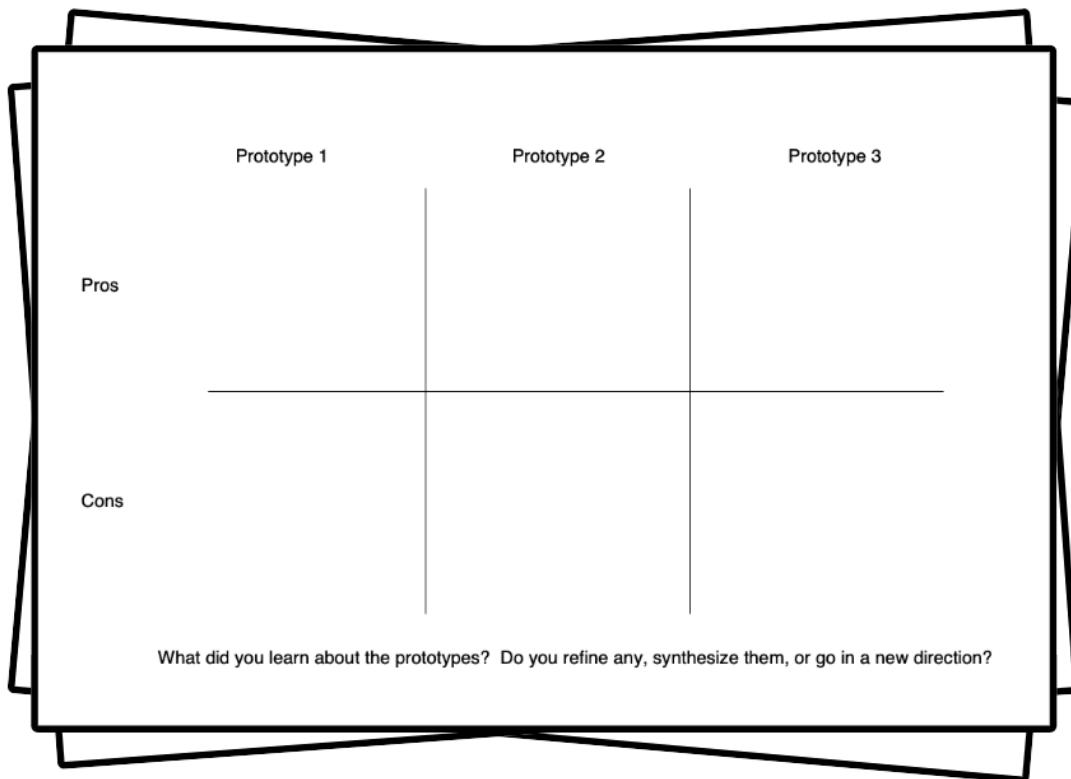
## 15.15 Creating an Action Plan

[to contents](#)

- Make an Action Plan for each prototype
- Record why you are making your next changes

An action plan should be created to address the points in the feedback grid. In software, this is called traceability, and it refers to why things are implemented. It is important to be explicit in these steps (i.e., writing down exact changes and what prompted this change) so that you remember why certain choices were made for the solution. An action plan should be made for each paper prototype; this will give you insight into what different implementations have in store in terms of full development.

## 15.16 Comparing Prototypes

[to contents](#)

- Reflect on what prototypes taught us
- Using a table helps guide group reflection
- Implementation is expensive: make sure you are acting on your best ideas

Finally, you should compare the various implementations of your solution. The feedback grids and action plans of each should be analyzed, and pros and cons of each solution should be extracted to decide on a solution implementation. You might decide to use one implementation or combine two or more. To guide group discussion, you could use a table similar to that of our template.

## 15.17 Assessing Technical Challenges

[to contents](#)

# Technical challenges

Challenge	What you will do: search resources, ask TAs for code example, build test program

- Gather your tools
- Identify technical challenges that require new skills
- How will you get those skills?
- Do you need proofs of concept to test technical feasibility?

Your solution may take on a form that requires knowledge that you do not have yet. For example, for a software-based solution, you might not have the technical knowledge required to realize your solution. This does not mean that you have to change your solution! It does mean that you need to catalogue the skills that you do not have yet, and where you might obtain those skills. You might consult an online search, a teaching assistant, a professor, or your peers.

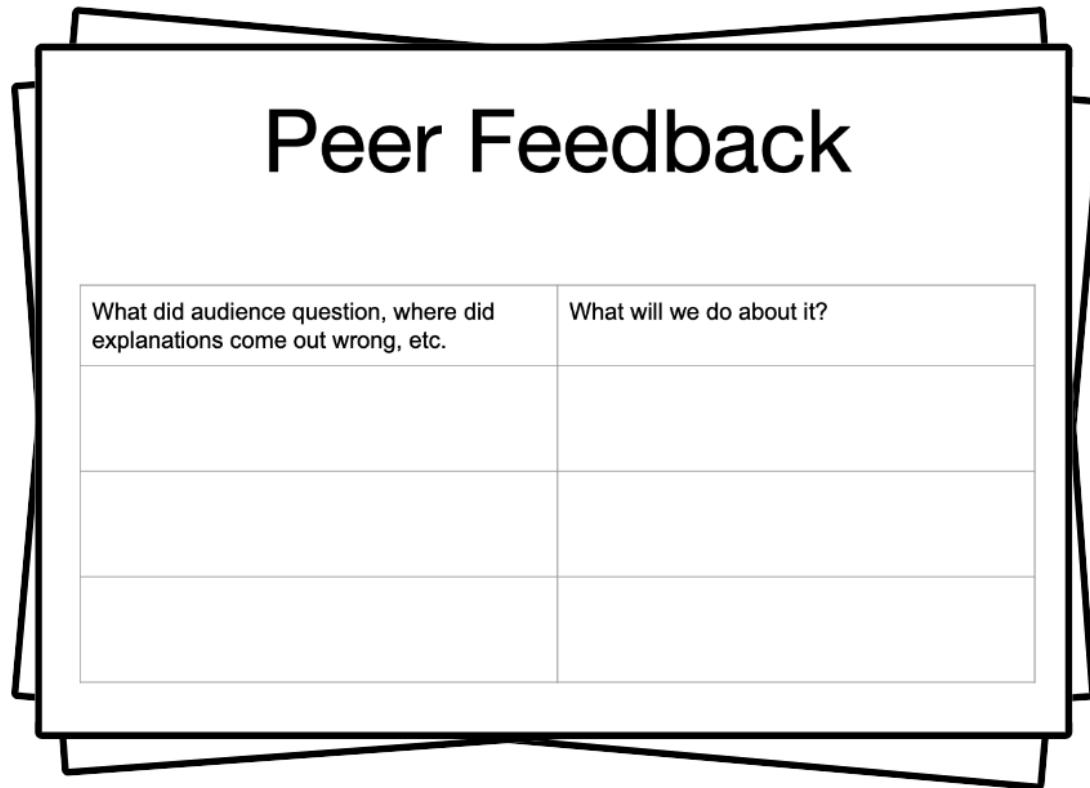
## 15.18 Assessing Risks

[to contents](#)

- Non-Technical challenges abound:
  - legal
  - regulatory
  - human resource
  - physical resource
- Another stakeholder (e.g., school boards) may block a solution your user (e.g., teachers) love

You should also reflect on project risks, which are more general than technical challenges as they involve things that are too complicated to do, no matter how much knowledge or skill you acquire. For example, your solution may be illegal! It may be patented by someone else. You may need approval of an ethics board or other regulator. You may need a part which is in short supply. You may not be able to distribute work to your team, because only one team member knows the frameworks you need. You may require a 1000 qbit quantum computer which won't exist for another ten years! The school board may refuse to buy iPads to run your solution.

## 15.19 Peer Feedback

[to contents](#)

- Peers have ears: tell them about your problems
- When you give feedback, ask
  - why they made certain decisions
  - can they trace them back to the user
  - why did they reject certain solutions
  - what surprised them in interviews and feedback sessions
- Giving constructive feedback may be the most important skill you learn in this course

In a course setting, you should take advantage of the fact that your peers are experiencing the same project process as you, and studying the same subjects, but have diverse backgrounds and interests, and therefore different perspectives. Through their own projects, they might have learned something that would be very applicable or helpful to you. You should seek their feedback throughout the design thinking process, especially during prototype development and iteration.

Conversely, you should give your feedback to your peers and guide them in any way you can. To do this, you should be an attentive listener and ask questions where things are unclear. You should ask them why they made certain decisions and aim to understand how their interview data has led them to design the solution that they have designed. Being able to give constructive and helpful feedback is both a technical and soft skill which can be applied to a variety of professional settings.

## 15.20 Pitching Your Solution

[to contents](#)

### Pitch Component

Tell us about your typical user.

Tell us about their problem.

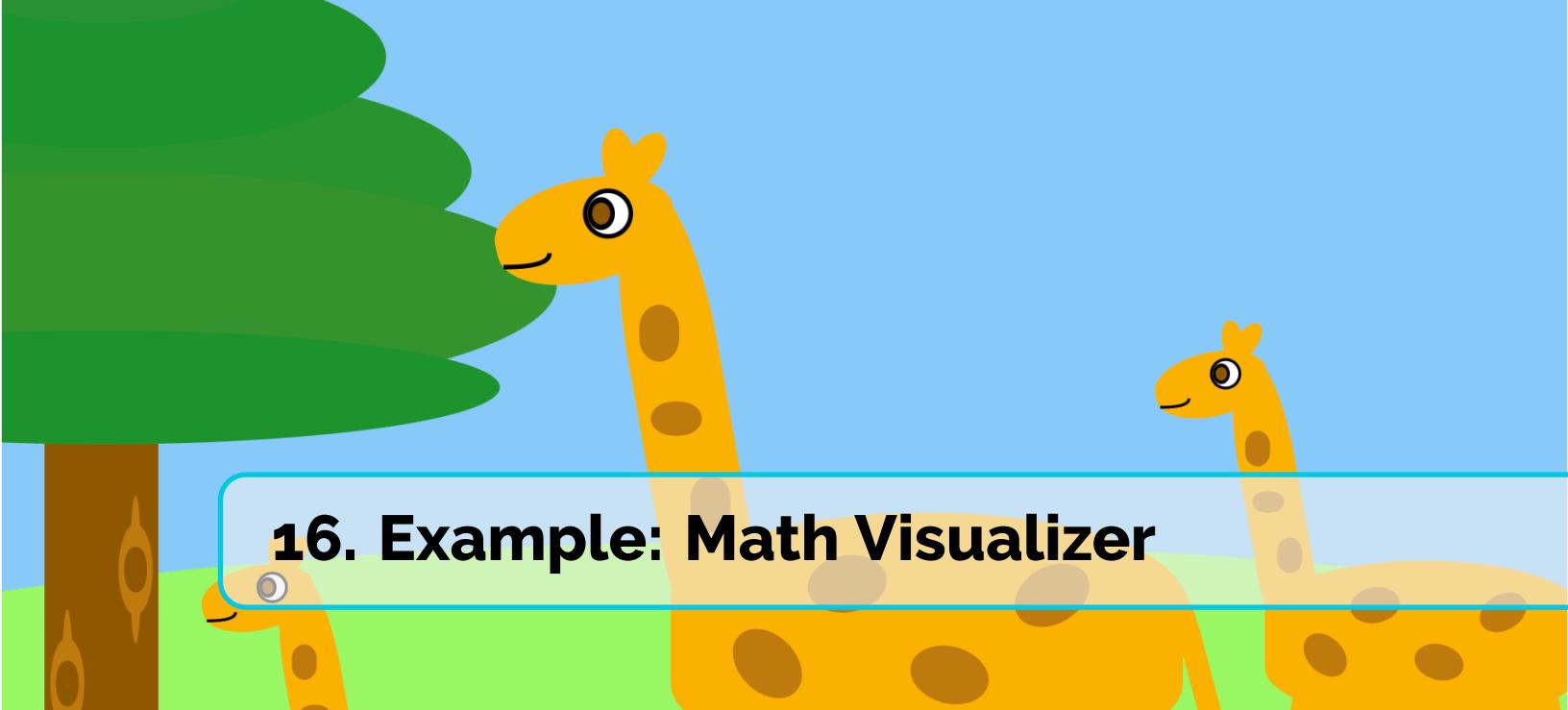
Who are you and why do you care about solving the problem?

Show how you solved the problem.

What are your next steps?

- A two-minute elevator pitch is an effective way to communicate your solution
- You should aim to be clear, concise, and interesting

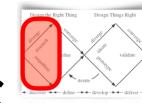
Your presentation may take on the form of an elevator pitch. Elevator pitches are clear and concise, so that the audience can understand your solution and maintain engagement throughout your presentation. Your presentation should relay who the target user of your solution is, the problem you set out to solve, who you are (i.e., what credentials you have and why you thought you could help), and how you actually solved the problem you identified. You should aim to make your entire pitch interesting - think about word choice and flow of ideas. Elevator pitches are short - about two minutes. The slide above lists the components that should be included in your pitch. Your presentation should include multiple slides so that you do not jampack all the information in one slide and overwhelm the audience. At the same time, having too many slides can be hard to follow along with.



## 16. Example: Math Visualizer

This completed example is aimed at using design thinking and Elm to help students with math. This is the starting point that was given to the students to work with and expand on. The template helps in defining a problem, the solution, and gives you the starting steps towards a functional prototype. Since this is also a part of a Computer Science course the solution should be software based.

# Possible Project Areas

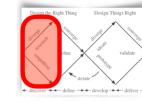


What do they need to know? Who are they? How will you find them to interview?

1. *Grade 7 Math, 7th Graders in CBSE (Central board of Secondary Education)*  
(<https://www.schoolconnectonline.com/cbse/Class-7th/Maths>), our younger siblings and their friends.
2. *Grade 11 Math, Students of Grade 11,*  
(<https://www.edu.gov.on.ca/eng/curriculum/secondary/math1112currb.pdf>), Juniors from High School

To come up with the possible project areas, we discussed amongst our teammates and came up with the two best options. We decided to go with the above options as the target audience was more reachable. We researched and concluded that working on the above curricula would both benefit the target audience and be feasible for us. It is important to make sure to have reliable access to members of the target audience here as you will need their input throughout this project.

# Option 1 Research

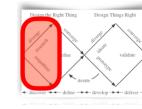


1. What could we find out about option 1? - *Grade 7 Math in CBSE has Algebra, Geometry, Probability and Statistics.*
2. Is it something people find hard to learn? – *Grade 7 Geometry is something that the students find a little hard to visualize.*
3. Why do people need to know it? (Will they be motivated by applications?) *To make math fun and simpler, it will help them in their day-to-day math activities and assignments. Will be adding animations and using interactions to keep the application interesting and easy to use.*
4. Is there a good visualization component? *Geometry, probability, and statistics*
5. How will we find people to interview? *We will be interviewing our younger siblings and their friends*

For the options research, you will need to answer the questions. With these answers, you and your teammates will be able to determine which topic is best suitable. To determine the topic that we would focus on and our target audience, we explored the sub-topics that most students found difficulty in. Tip: You don't need interviewees at this stage, you can answer these questions according to your knowledge.

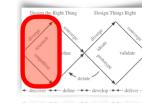
- Question 1 is talking about all the research you can produce about the topic
- Question 2 is about assessing the difficulty of the topic and determining how the target audience deal with the option
- Question 3 is finding out how important is it for people to know about it, and applications related to the option
- Question 4 is figuring out if it is possible to visualise the topic in the form of figures, charts, etc.
- Question 5 is about how easy is it to find people relevant to the target audience, you can use sources like family, friends, your peers in class, the criteria is to have reliability for the interviews to come.

# Option 2 Research



1. What could we find out about option 2? *Grade 11 Math has Calculus - derivatives, integrals, and Algebra.*
2. Is it something people find hard to learn? *Depends on the person and their adaptability to abstract math.*
3. Why do people need to know it? (Will they be motivated by applications?) *Cause it applies to the real-life situations. We will apply animations, quizzes, and provide hints to keep the application interactive.*
4. Is there a good visualization component? *Real-life questions where calculus is applied and visualized using animations.*
5. How will we find people to interview? *Communicating with our juniors from high school*

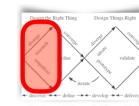
# Our Focus



1. Target users: *Grade 7 Students in CBSE (Central board of Secondary Education)*
2. General problem area: *7<sup>th</sup> Grade Geometry : Angle measurement.*

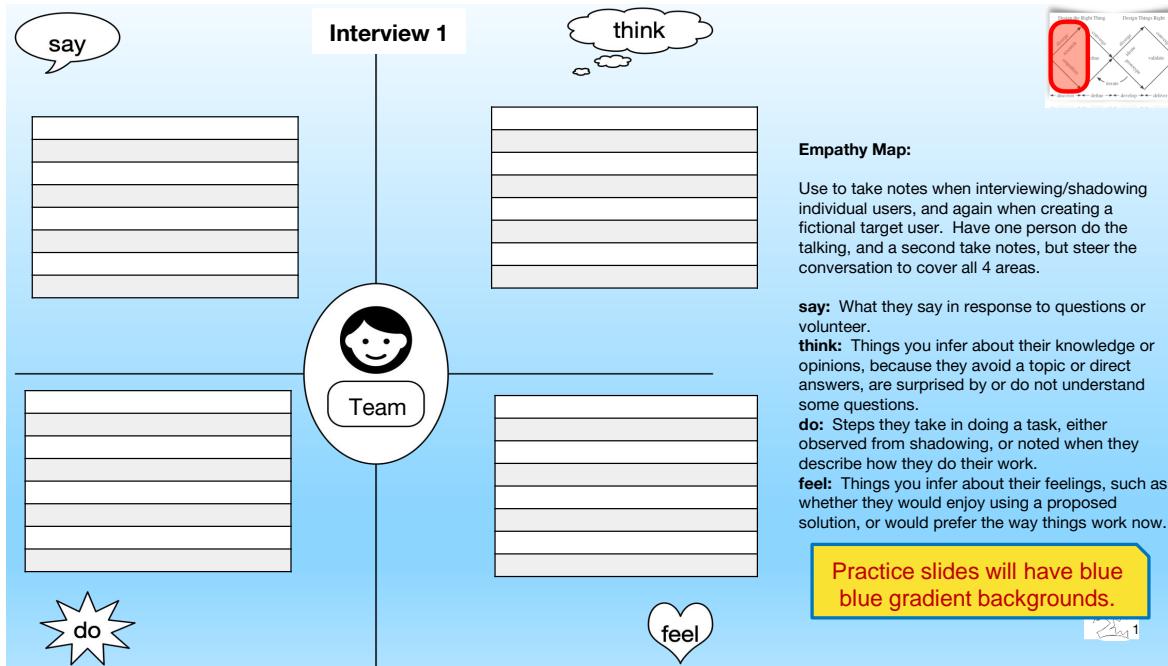
After looking at your options, you and your team should choose an area to focus on, from the project areas that you have researched. To make the journey easier, you can choose the option which has easy access to your target audience, ease of building a project with the topic, etc.

# Questions to Ask



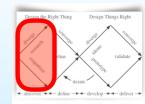
- How old are you? What do you like to do for fun? Dream job? What subject do you struggle with the most? What's your favorite subject? Why is that ?
- What do you think about Math at school ? Why do you think so? Do you like Geometry? Do you like your teacher?
- Do you know anything that your teacher's uses to help you understand Math better? The same about your textbook?
- Do you think Algebra is a useful subject in the Real-World?
- Do you practice problems in Math to learn the topic ?
- Do know anyway of teaching that you think is better, to learn about a topic?
- Could you describe your normal day after school ?
- What's your favorite game and what do you like about it?

In this slide, you need to note down the questions, you will ask your interviewees. The questions are meant to be open-ended and should not be leading to a conclusion that you have previously determined. The questions should be specific and not close-ended. For example, a question like “Do you like geometry?” is very specific and leading. An alternative is “What topics in math are you interested in?” Leading questions would not lead to the type of answers that help you to construct your project idea. As a rule of thumb, make the questions without having any conclusions in mind. Another tip is to make sure that you are starting with questions to get to know your interviewee, to make them comfortable, such as - “What are your hobbies?” “What is your favorite subject?” etc.



You can do a practice interview with your teammates! Have one of them be the target audience. For example, we had our teammate act as a grade 7 student. You should ask the questions that you and your teammates have prepared in the previous slide. You or another teammate of yours can be the note-taker to ensure that you fill out the empathy map. The Empathy map helps to understand your target audience better. These maps and interviews help you to determine the problems and solutions, and it is a practice to improve your communication and organizational skills. Tip: Create questions that can help you fill all 4 sections on the empathy map and make sure to keep the points concise.

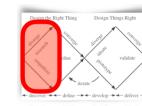
**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.



Did you...			
set up interview, giving context?	Somewhat	Yes	Definitely!
make them comfortable?	Somewhat	Yes	Definitely!
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never	Sometimes	Always

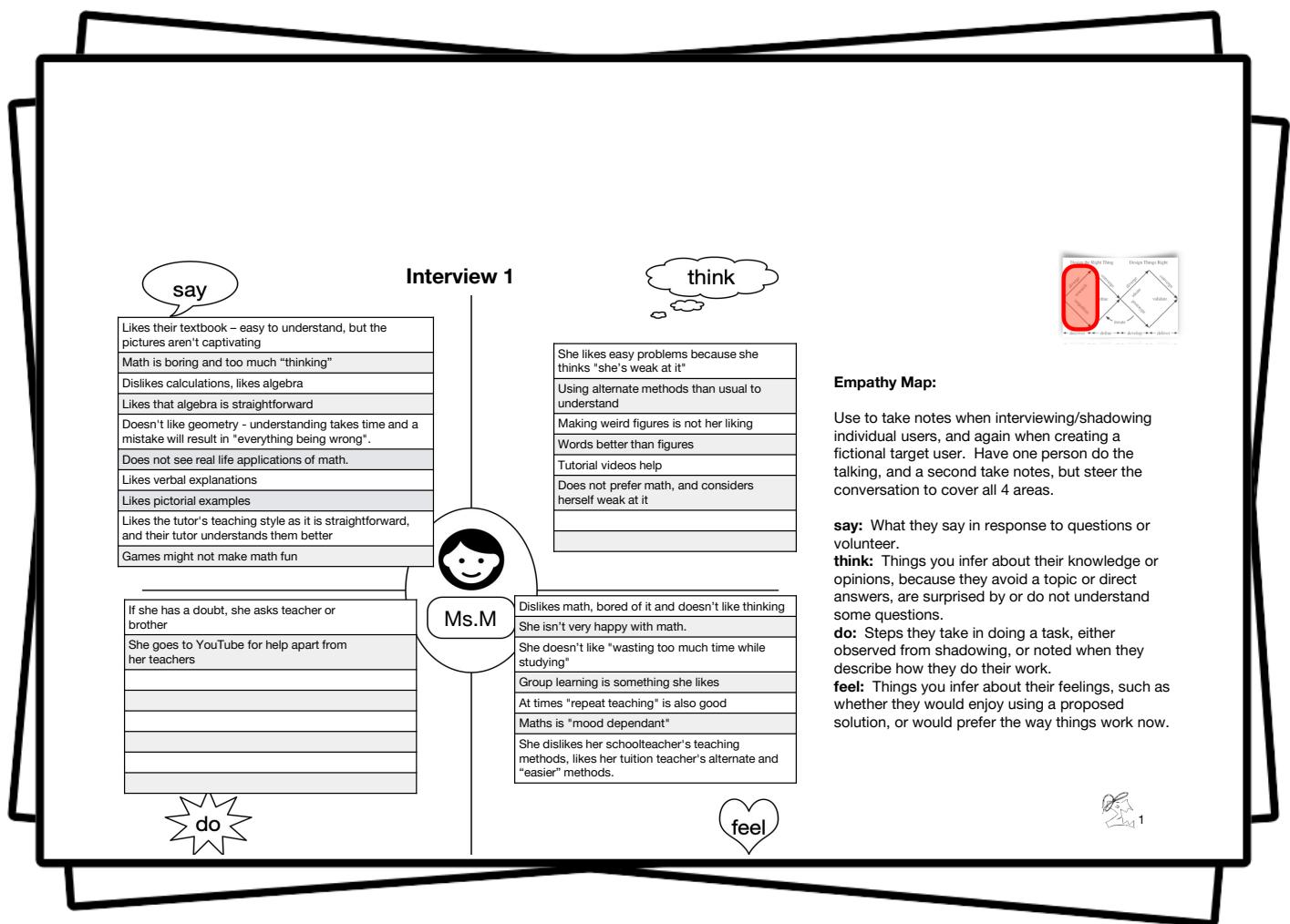
After the empathy map, mark the relevant interview skills that were portrayed in the practice interview. This will help you create better questions for the interviews, and if you miss any interview skills, you can always refer to and prepare better for the next interviews. You can repeat this practice as many times in the week till you get satisfactory questions for the interviews.

# Revised Questions

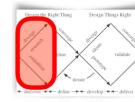


- How old are you? What do you like to do for fun? Dream job? What subject do you struggle with the most? What's your favorite subject? Why is that ?
- What do you think about Math at school ? Why do you think so? What's your most favorite and least favorite topic in Math ? Is there anything about Math Classes that you like/dislike ?
- Do you know anything that your teacher's uses to help you understand Math better? The same about your textbook?
- Do find any topics that are very rarely used in the real-world?
- How do you study Math ?
- Do know anyway of teaching that you think is better, to learn about a topic?
- Could you describe your normal day after school ?
- What's your favorite game and what do you like about it?

From the interview skills and practice interview, the questions that are modified need to be inserted here. The revised questions will help to guide your interviews with your target audience from now on. Ensure that your questions are firstly open-ended, getting to know your target users, then to understand what they would like in an application such as – the best features, what they would like the app to have, etc. For example, instead of “do you like your teacher”, you can ask ”Is there anything about Math Classes that you like/dislike?” This is a much better question to ask as it doesn’t limit the answers just about the teachers, but about her math classes in general. Do not ask “Do you think Algebra is a useful subject in the real world?” Do ask “Are there topics you find hard to imagine in real-world use?” The idea is to widen the answer range for them and give them the freedom to think, and discuss. Asking them to list topics will focus them on specifics and hopefully avoid statements too general to be useful. If they are not confident about making an overall judgement, or finding the most important topic, the formulation of the question takes the pressure off by asking for any topics.



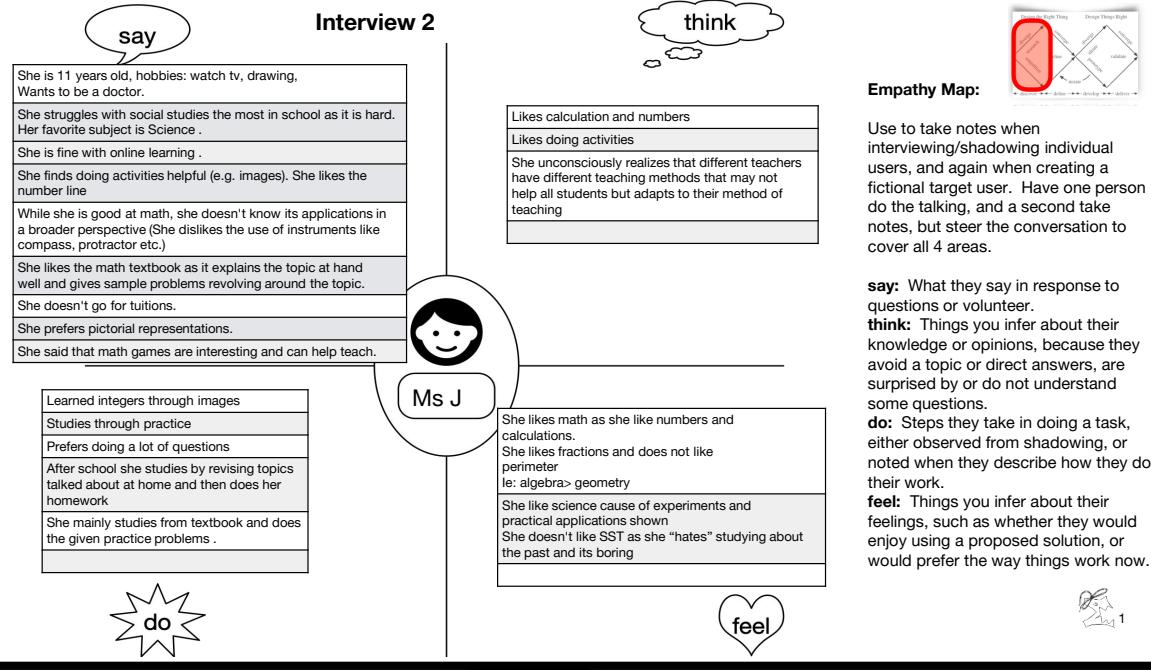
Make your interviewees feel comfortable by making the interview less intimidating, asking easygoing questions, and getting to know your interviewees. As an interviewer, it improves your social and communication skills, and as a note-taker, it helps you to be flexible and adaptable. The goal of this interview is to get as many opinions and problems as possible as it helps us generate ideas for an application. The first interview would most likely be a new experience for both sides with its fair share of minor inconveniences, but it will give you all the confidence for future interviews. Tip: 4-6 interviewees are recommended as you get different opinions and ideas.



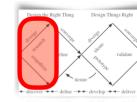
**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.

Did you...			
set up interview, giving context?	Somewhat	Yes	Definitely!
make them comfortable?	Somewhat	Yes	Definitely!
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never	Sometimes	Always

Highlight, circle, or underline the appropriate choices, according to the interview held. The note-taker or the interviewer should fill this out after the empathy map.

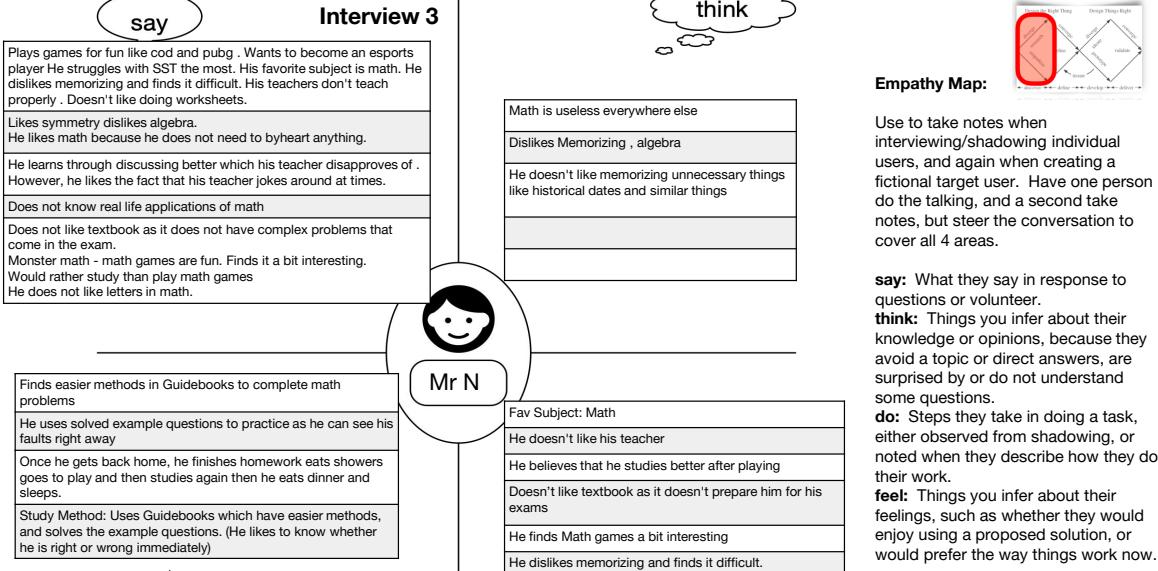


The note-taker should try to be fast, and accurate and capture as much information as possible. Even if the interview went well, if the notes are not used it won't be as effective in the later stages. As a note-taker, try to fill out the 4 areas from the interview. Tip: To fill the 4 areas, try asking specific questions that might answer each quadrant, for example – for FEEL, ask: What about math do you dislike? Or for DO, ask: If you face a difficult maths problem, what do you do to try solving it? The written responses shouldn't be too wordy, try writing them as highlights or bullet points. Tip: Create a doc and type in the conversation or replies of the interviewee, this way, you and your teammates can decide what/where replies should be placed in the empathy map after the interview, when you have time to discuss it.

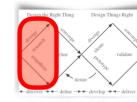


**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.

Did you...			
set up interview, giving context?	Somewhat	Yes	Definitely!
make them comfortable?	Somewhat	Yes	Definitely!
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never	Sometimes	Always



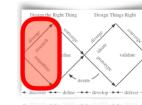
The responses from the interviewee, sometimes, may not associate with the topic but try to include them in the empathy map as it may be useful for brainstorming. Some interviews may not meet the expectation of replies, if the interviewee doesn't answer the questions asked accordingly, you can try asking more specific questions. If certain responses are hard to paraphrase, you can then quote them. For example, instead of - He learns through discussing better which his teacher disapproves of, use - He "learns through discussing better" which his teacher "disapproves of".



**Interview Skills:** Fill this out after empathy map interview for your group. Highlight your choices.

Did you...			
set up interview, giving context?	Somewhat	Yes	Definitely!
make them comfortable?	Somewhat	Yes	Definitely!
ask open-ended questions?	Never	Sometimes	Whenever appropriate
Ask follow-ups?	Never	Sometimes	Whenever appropriate
Ask how they felt about things?	Never	Sometimes	Whenever appropriate
Ask for stories, or to describe the steps of a job?	Never	Sometimes	Whenever appropriate
React to non-verbal clues they were uncomfortable, wanted to say more, etc?	Never (Online can't see interviewee)	Sometimes	Always

# Pivot or Not?



1. Was it easy to find potential users? Yes
2. Is there commonality among the user problems? Yes
3. Were interviewees excited about it? Yes
4. Were we excited about it? Yes
  
5. If you pivot to another idea, copy your Questions and Empathy map slides and start catching up. ☺

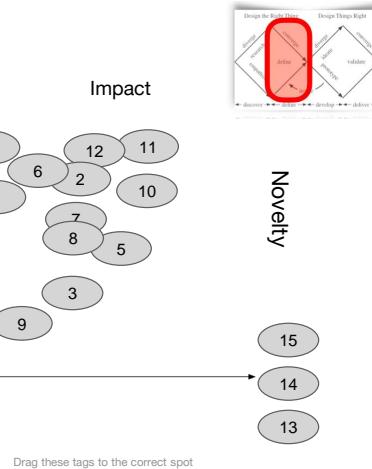
In this slide, you are re-evaluating your choices and checking if your initial idea allowed for a viable answer. If you do not have enough people to interview, the problems mentioned are too diverse or face an unforeseen obstacle, it may be better to start over. This slide helps you and your teammates to decide whether the topic you have chosen is good to go and if the target users require help with the topic chosen. If you and your team want to go with another idea, then you can do so! After this slide, you can copy the questions and empathy map slides, and do another interview round regarding the new topic.

1 Maths is boring
2 They don't find its application in real world
3 they don't see the inner meaning in the math
4 They are embarrassed to ask their teacher to re-explain a topic
5 They don't like using mathematical instruments for geometrical constructions
6 Textbooks are boring
7 Textbooks don't prepare them for exams
8 They lack the confidence a lot of times to go after a difficult or a higher-level problem.
9 Most Questions lack solutions to them to compare answers.
10 Teachers' teaching methods can be boring and stale
11 Teachers cater to a general audience and don't produce unique methods to make sure, each student has grasped the concepts
12 Difficulty in memorizing some formulas

#### How Might We?

- As individuals in the group, write five to ten different ideas on a piece of paper, for how we could make our users' lives better by reducing a problem or giving them a new opportunity.
- As a group, take turns reading your ideas, and if they are very similar to other ideas, merge them together and write down one version of the idea in the table.
- Again as a group, discuss the ideas in reverse order, and assign them potential Impact and Novelty scores by plotting the number on the scatter plot.
- Pick the best overall statement or statements, and combine them into your goal, in the box below.

We want to find a way to help  
with



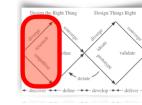
user  
Class 7 students of the CBSE curriculum.

How to use a Protractor.

need

Finding the issues that the interviewees have with the topic of concern should be evaluated and noted down here. In the above slide, the highlighted problems led to the conclusion mentioned. Try to find relations between problems and figure out a common root problem. This step will determine the project you will work on. The graph is a visual aid to represent the level of novelty and impact solving a problem will have. Highlight the problems that you and your team want to create solutions for. Tip: To fill out the problem definition slide, you and your teammates can each write down 5-10 different ideas on how it could be beneficial for your target users. If the ideas are similar then you can merge the ideas. Once you have 15 problems ready, you can write them in the table and place the numbered tags in the graph accordingly. Impact, in this context, means the influence that the idea has on the users' lives. Novelty, in this context, means the newness or freshness of the idea. Note: Try to write down at least 10 ideas in this slide.

# Q&A



1. Write down questions raised by the HMW process. For example, if you identified fraction arithmetic as an area your users find confusing, what grade is it covered in Ontario's math curriculum? If PDEs cause your users trouble, what are they?

*How might we make the application fulfill the users' needs?*

*How might we help the user discover the inner meaning behind the math topic they are learning?*

*How might we make a solution that helps every student understand the topic?*

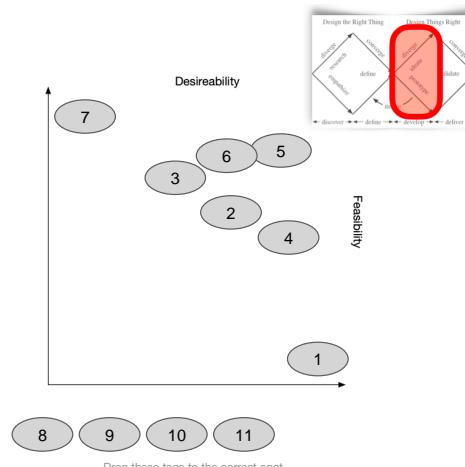
*How might we keep the attention of a student to our application?*

*We found that our users, grade 7 students, had difficulty visualizing geometry. We decide to produce tools to identify and measure angles and provide lessons to explain the concepts in geometry.*

This slide will help you figure out the right question you should ask yourselves when you will group up to come up with a solution. The “How Might We” or HMW method allows you and your team to brainstorm and be able to resolve future problems that the users may face. For example, we identified from the previous interviews that our interviewees were facing difficulty in concentrating or enjoying math, thus we need to write down questions associated with the problems such as – how might we bring real-life problems associated with math into the application?

## Solutions

1	A virtual environment to draw and measure angles with a digital protractor.
2	Digital Transparent protractor to experiment with.
3	Practice Exercises on measuring angle with a protractor with solutions .
4	Drawing polygons with specific angles using a protractor.
5	Tutorial on how to measure angles using a protractor.
6	How to measure the angle on both directions of a protractor.
7	Tutorial Videos on how to use a protractor
8	
9	



### Ideation:

- As individuals in the group, write five to ten different solutions on a piece of paper.
- Include an idea costing less than \$1000 and one costing a million, include one using an app, and one without an app.
- As a group, take turns reading your ideas, and if they are very similar to other ideas, merge them together and write down one version of the idea in the table.
- Again as a group, discuss the ideas in reverse order, and assign then Desirability and Feasibility scores by plotting the number on the scatter plot.
- Pick the best overall idea, and combine them into your goal, in the box below.

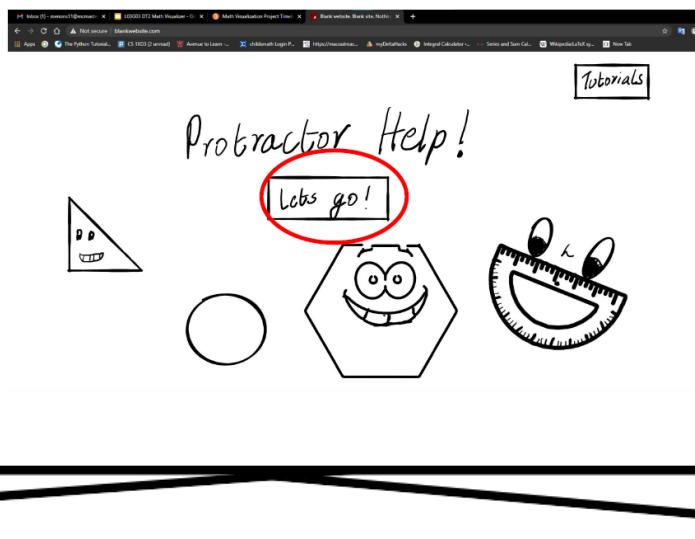
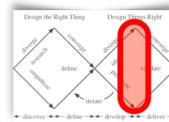
A sandbox application that allows a user to look up tutorials on using a protractor, provides a digital protractor for the user to get familiar with and provides a few exercise problems where the user drags a digital protractor to measure a given angle and select an option promoting a right/wrong response and providing the solution after.

Discuss the various solutions with you and your team and note them down. Come up with a solution imagining that you have a billion dollars, as impossible as it sounds, you want it. Then come up with a solution from the opposite end of the spectrum, one that costs minimal to nothing. After figuring out these outliers, come up with more realistic solutions and list as many as possible. Every idea counts here! Once you are out of ideas, roughly rate the solutions based on feasibility (if it is possible within the scope of this course) and desirability (how much the target audience wants this solution). Discuss and figure out your final solution that the team converges to and state it below.

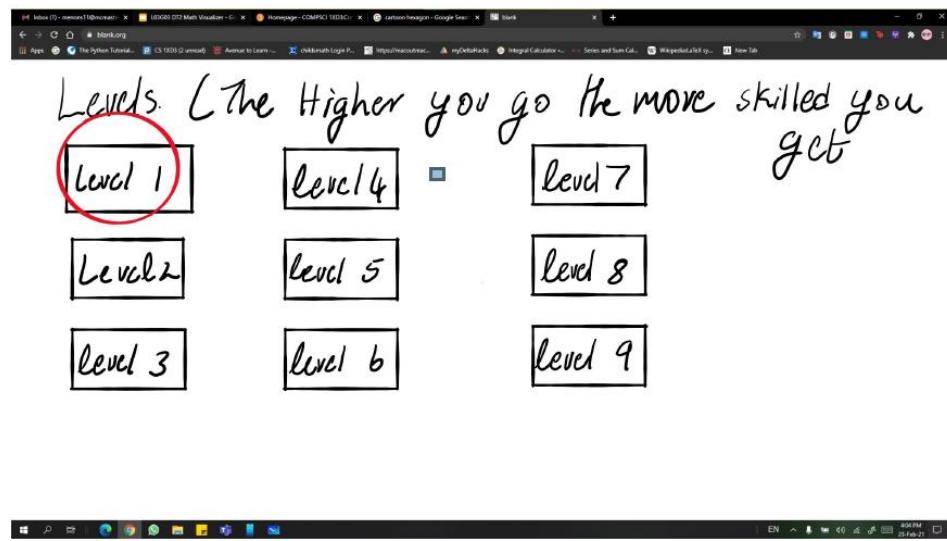
## Prototype

### 1 (Idea 1)

Use any drawing tool you want to create a "paper prototype", including taking screen shots of Elm programs or other apps. You will need to duplicate this slide for different pages/actions in your app.



Prototype 1 is the first and simplest idea of your application. Writing and drawing your ideas out helps to visualize and solidify ideas. Start with something, like a line or a curve, anything, if you are stuck with a rough idea but nothing visually intact then it becomes a barrier for the final prototype/application. Tip: You and your teammates could each come up with a paper prototype, if they are similar then you can combine the idea, else you can insert the different ideas accordingly. There are different ways to create a prototype, you could draw on a piece of paper (but that wouldn't be environmentally friendly), draw on MS Paint or other drawing apps and take screenshots, create a PowerPoint that takes you through each page, etc.

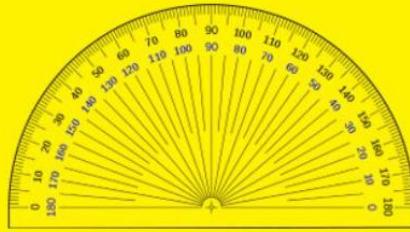
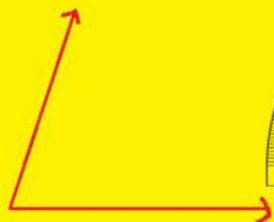


The red circle indicates the button being clicked. In the previous slide, we see the "Lets go!" button being clicked which leads us to this page. The current slide is a levels page where there are different stages of math problems. When we click on a level button, here we chose level 1, it leads us to the slide below.

Measure the angle below by dragging the protractor, and enter it in the box below:

[Back](#)

LVL 1

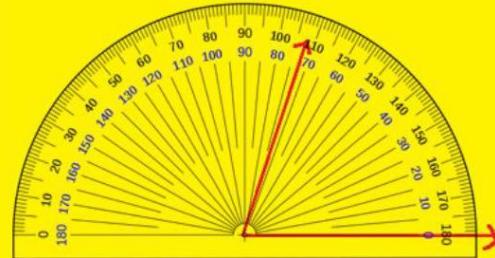


[Enter](#)

Measure the angle below by dragging the protractor, and enter it in the box below:

Back

LVL 1



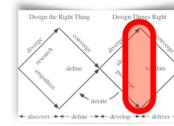
You got it right!

72

Enter

**Prototype****2** (Idea 2)

Use any drawing tool you want to create a "paper prototype", including taking screen shots of Elm programs or other apps. You will need to duplicate this slide for different pages/actions in your app.



My team and I had a combined idea thus we decided to fill out only prototype 1/idea 1. We do recommend to have more than one idea as your users will be able to pick and choose the best features and ideas in each. By having at least 2 ideas or prototypes, you and your team can come up with the better solutions/prototypes for the users.

**Prototype 1**

**Feedback Grid:**

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

+	What did they like about it?
Δ	What would they like changed?
?	What questions did they ask?
!	What new ideas came out of the discussion?

https://www.svgrepo.com/svg/120932/light-bulb

From this slide onwards, ensure to keep in contact with your interviewees as they can do the testing and provide the feedbacks that you require! If your previous interviewees aren't available at certain moments, then you can add another interviewee. Try to explain the idea/prototype to your interviewees. The best tip that we could provide is to imagine you are in 'Shark Tank' (exactly like the TV show, but this is an idea pitch, not a business deal!) and you are trying to pitch your idea! You have good reactions or not-so-great reactions but the feedback from all is very important.

# Action Plan

What they said:	How we will improve:

This is for practice. Act on feedback from real users!

In your action plans, you may receive feedback that might change the solutions or that might be almost impossible to solve. Write them down here, and note down how you can improve or solve these problems. The action plans help you to create goals for the application.

**+**

It is more entertaining.
Gives feedback and appreciates you
Custom backgrounds
Avatar and currency
Liked the drawings

**△**

Custom backgrounds,
Something to keep you motivated to keep going
Make the smiley a lot less creepy

**Prototype 1**

**Feedback Grid:**

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

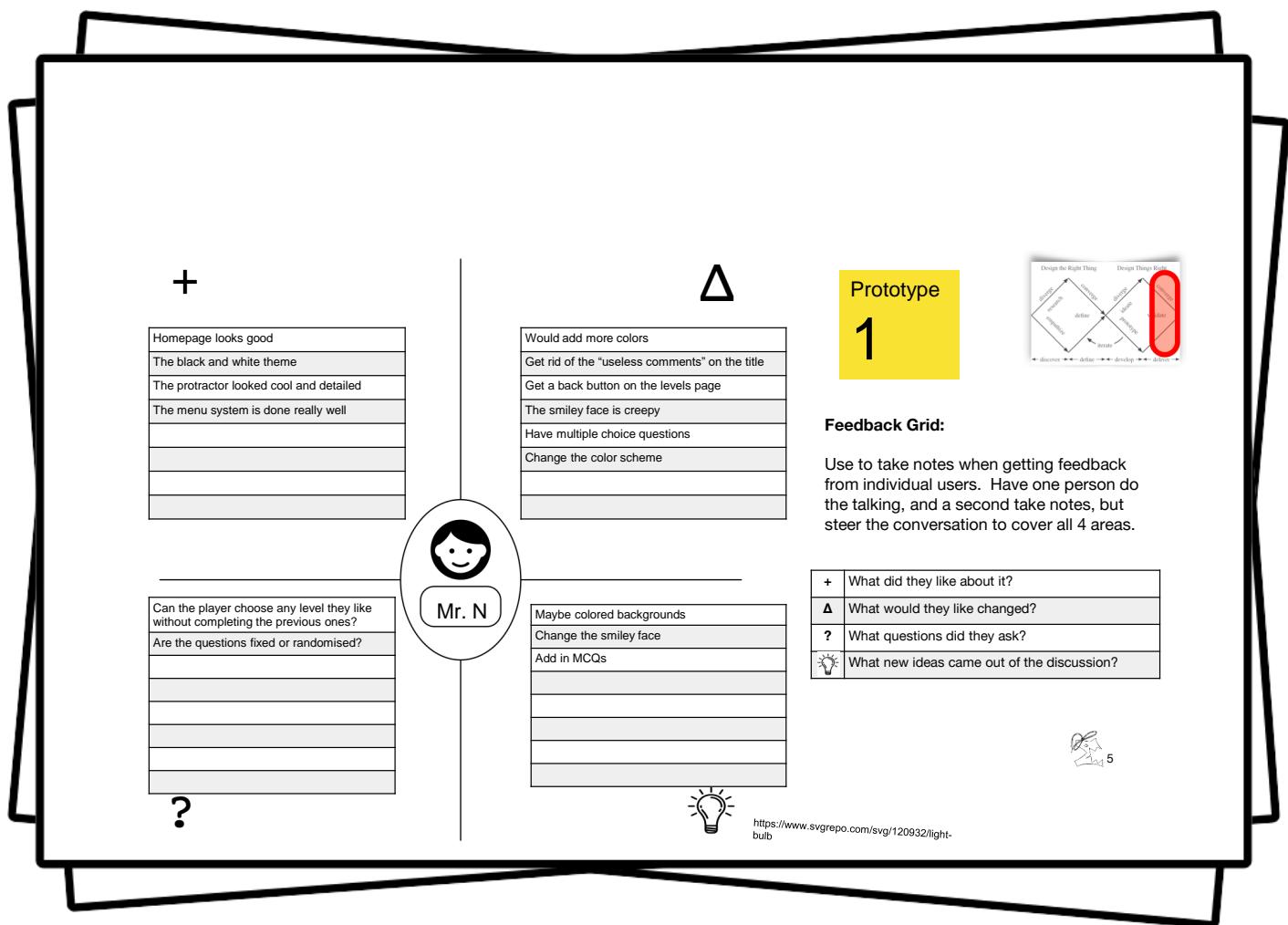
+	What did they like about it?
△	What would they like changed?
?	What questions did they ask?
💡	What new ideas came out of the discussion?

**?** The interviewee was prompted to ask questions multiple times, but still did not have any.

<https://www.svgrepo.com/svg/120932/light-bulb>

5

Explain the idea behind the prototype that you have made, and show them a mock demo. Since these are paper prototyped it needs your help to convey your idea to the interviewees. Once the prototype is clear to them, do the same as your first set of interviews, ask open-ended questions about your prototypes. Be open to changes and note them down, even if the changes might sound impossible, note them down. Feasibility is for you to figure out later when you read these notes and come up with updates.



When the same improvement is mentioned by multiple interviewees, it is a clear sign that it needs to be addressed, just like the creepy smiley face. The solutions suggested by interviewees need not be directly implemented. Maybe a different easier, and feasible solution would give the same improvement.

**+**

Buttons are obvious
Homepage drawing are attractive
Levels are well explained and obvious

**Δ**

Homepage could be more colorful
Font change on level Page
Level page could be colorful

**?**

No Questions...

**Prototype 1**

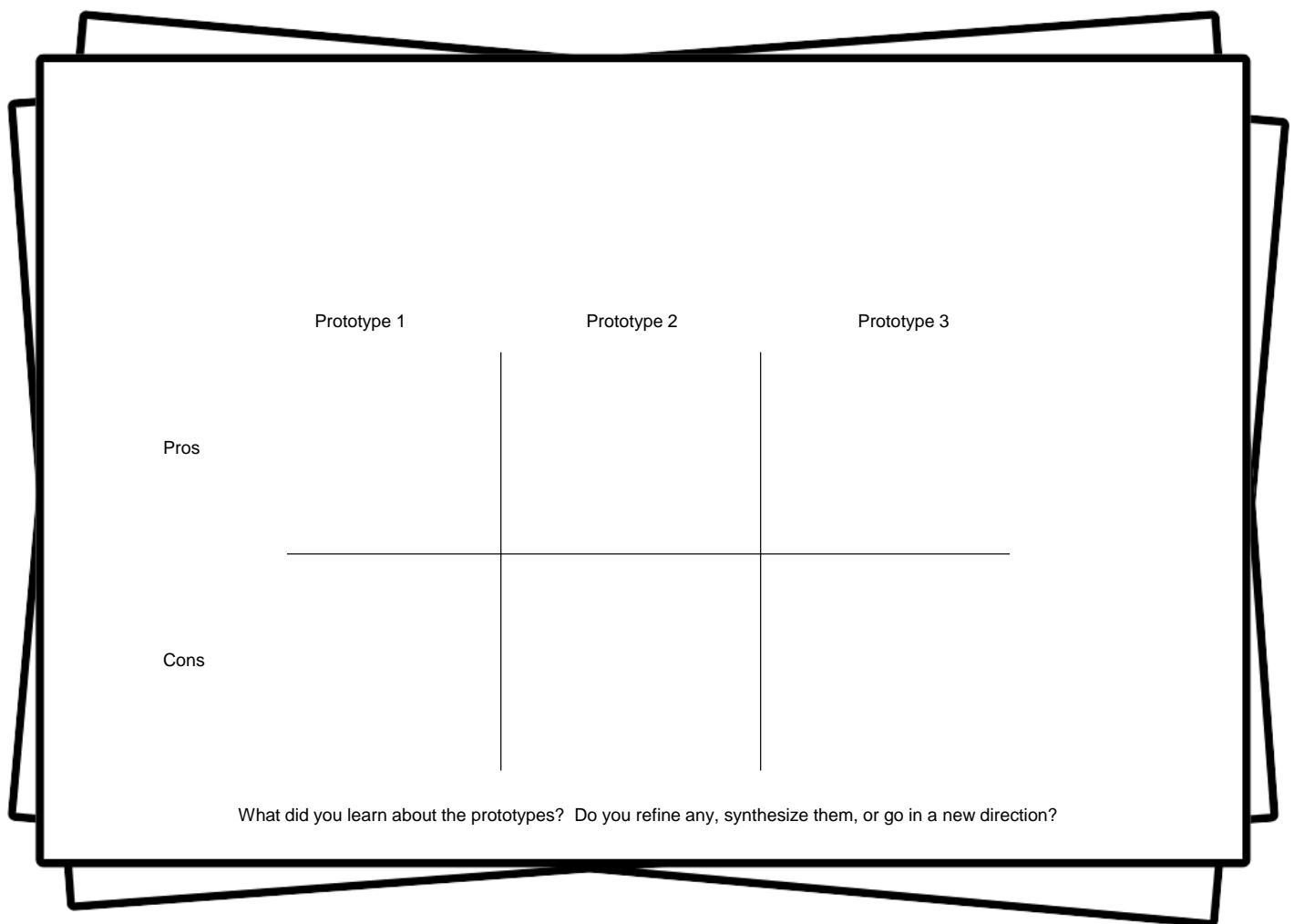
**Feedback Grid:**

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

<b>+</b>	What did they like about it?
<b>Δ</b>	What would they like changed?
<b>?</b>	What questions did they ask?
<b>💡</b>	What new ideas came out of the discussion?

<https://www.svgrepo.com/svg/120932/light-bulb>

Try to fill as many sections as possible and fill out the tables. There may be situations where the interviewee simply doesn't have any questions, thoughts, changes, or ideas and that is fine. In case of this, mention that the interviewee didn't have anything to say. If you face such interviews, try interviewing more than 2 users. The probability of getting feedback will be higher. We had 3 interviewees therefore we were able to make changes and come up with solutions and ideas. After this slide, if you have more than one prototype/idea, copy and paste the empathy maps after this slide, with the responses for the idea accordingly.



	Prototype 1	Prototype 2	Prototype 3
Pros			
Cons			

What did you learn about the prototypes? Do you refine any, synthesize them, or go in a new direction?

Here you would be comparing the prototypes that you have made before and learn from them. We had created one prototype but don't do what we did, create at least 2 in the beginning! Use these thoughts and improve upon your prototypes. Note down the pros and cons of each prototype here and collect the best features to evolve another prototype.

# Action Plan

What they said:	How we will improve:
Backgrounds are a bit bland	Better backgrounds
Need something to keep us motivated	Maybe add a score/currency system
Add in MCQs	Make questions MCQ
Did not like the smiley face	Don't add the smiley face

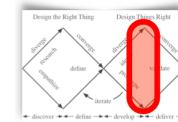
Duplicate the Feedback and Action Plan slides for as many iterations as you can fit in.

## Prototype

**3**

Elm share link: <https://macoutreach.rocks/share/9ccb0fa0>

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



You can now start to make an ELM prototype. A basic skeleton and idea would be enough. For example, if involves a level system, then a few examples of levels, a menu system and any other sections thought of would do. It should give a starting step that reflects the final products in some way.

# Technical challenges

Challenge	What you will do: search resources, ask TAs for code example, build test program
How to include the protractor	Ask TA for code and search internet
Variety of questions	Research on 7 <sup>th</sup> Geometry

Here you would discuss the technical challenges that could pop up when making your application on Elm. Are there any concepts that you need to learn before you start making the application? What kind of research is needed to make a factually accurate application? What other issues do you anticipate? Your solutions can range from self-research to asking your mentors for an extra tip regarding the programming language.

# Peer Feedback

<b>What did audience question, where did explanations come out wrong, etc.</b>	<b>What will we do about it?</b>
Do the levels have an increased difficulty?	Yes we do plan on making the levels more difficult as you go
Is there any content provided to do the questions?	We will work on a tutorials page to make sure that student have a resource to refer to whenever needed
How many external applications are provided to cover the content of angles?	We are planning on having questions that ask you to figure out what kind of triangle it is based on the angle of each corner.

Asking your peers other than your teammates would allow for a different perspective on the idea that you have, helping to polish any and all details missed out or not perfected.

# Risks

Main things which could go wrong and prevent you from completing a working app.	What you can do about it? (assign a second person, find more users, etc.)
Not being able to include a Protractor	New topic of geometry

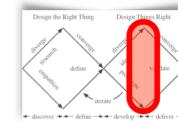
It is time for risk assessment! Here you will find out the achilles heel (the biggest weakness). This could range to anything from something that Elm can't do to something you won't be able to figure out. Even if you think you can figure something out, if you've never done it before, you should probably acknowledge the risk, but do not worry, you will be experts soon! To mitigate these risks raise them at group meetings with your mentors, online research, and our good friends at StackOverflow.

Prototype

7

Elm share link: <https://macoutreach.rocks/share/9ccb0fa0>

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



Time to start the prototype. Remember, the best prototype is the simplest prototype which gives you more information about your user.

+



**Mr. N**

△

Liked the color scheme
The homepage is nice
Texts are easy to read
Linked the protractor
The levels are pretty straight-forward and you know what to do
Nice font for levels
Liked the format of the questions
Dragging the protractor is cool

Add stuff to homepage background
Could mistake the 'angle management' title for a button
The background in tutorial page
Tutorials should have names
Add circles to the angles
Font for tutorials
Make it so you can go from one level to the next
Ask what type of angle that is
Do better to congratulate the students when they get it correct

What does the 'help!!' Button do?
Can you go back to the level page without completing the level?
Can I choose any level I want?
Is there a score system?
Do I get a prize after finishing the game?
Will you get a bonus level if you get everything right?

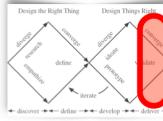
Add background to the tutorial page
Add title to tutorials
Change the color scheme for tutorial 2
More visuals on the tutorials
Change the text and background of level page
Change the distance of level buttons
Implement a score system



<https://www.svgrepo.com/svg/120932/light-bulb>

**Prototype**

**7**



**Feedback Grid:**

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

+ What did they like about it?
△ What would they like changed?
? What questions did they ask?
💡 What new ideas came out of the discussion?



5

The sharelink/prototype 7 is displayed to the interviewees and feedback is received. These interview sessions finalize your final prototype.

**+**

Liked the colors
Homepage is attractive; wants to try the app out
The question difficulty increase

**△**

The name can be better
Have an explanation on why the answer was that and a hint when you get the answer wrong
Change the tile of button to concept review
Add angle indication in Tut1

**?**

How would you know that the protractor isn't a bit off?
Can you change the font?

**Ms. N**

**Prototype 7**

**Feedback Grid:**

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

<b>+</b> What did they like about it?
<b>△</b> What would they like changed?
<b>?</b> What questions did they ask?
<b>💡</b> What new ideas came out of the discussion?

<https://www.svgrepo.com/svg/120932/light-bulb>

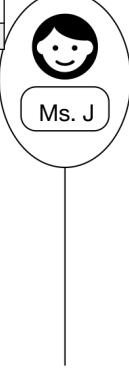
5

+

Finds the buttons perfect for their uses  
 The reflex angle catches everyone off guard.. It's a new question for them  
 The game is very informative and slightly challenging (good)  
 Believes it could replace teachers teaching the topic and might be faster as well

How do you turn the protractor?  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

?



Δ

Needs angle indicators on the tutorials  
 Update tutorial 1 as there is no specific information of the angles (between 0 and 90)  
 Wants to add rotation to protractor  
 Wants a hint button  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

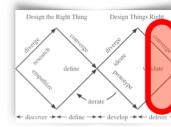
Maybe add a tutorial button for each level  
 Wants some sort of reward for completion  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_



<https://www.svgrepo.com/svg/120932/light-bulb>

Prototype

7

**Feedback Grid:**

Use to take notes when getting feedback from individual users. Have one person do the talking, and a second take notes, but steer the conversation to cover all 4 areas.

+	What did they like about it?
Δ	What would they like changed?
?	What questions did they ask?
💡	What new ideas came out of the discussion?



5

# Action Plan

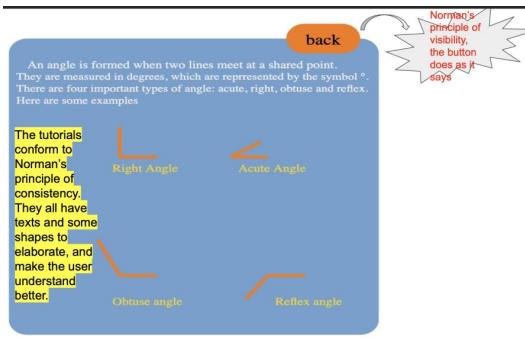
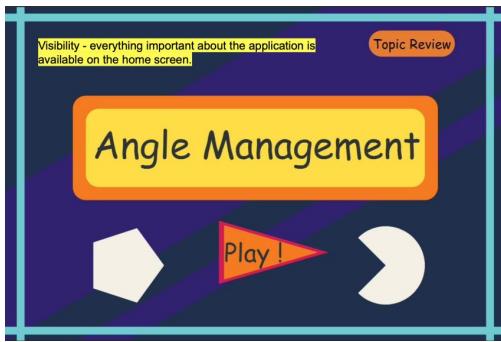
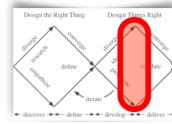
What they said:	How we will improve:
Users had issues with the reflex angle problem and The triangle problems	Add tutorial information on how to solve such questions as well as a hint button.
They didn't like backgrounds for tutorial page and level page	Make custom backgrounds
The fonts for the tutorials were not good	Change the font
Add a scoring system	Make a scoring system

Duplicate the Feedback and Action Plan slides for as many iterations as you can fit in.

## Prototype

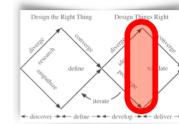
8

You should have a more detailed prototype now, but you may start including more screenshots of Elm programs and may even include sharelinks to working examples of parts of the app.



Prototype

8



### LevelPage

Back

Mapping: it signifies that the user can interact with the space and go to different levels/new spaces

- Level 1
- Level 4
- Level 7
- Level 2
- Level 5
- Level 8
- Level 3
- Level 6
- Level 9

### Tutorial Page

Back

Also implements mapping

- Tutorial 1: Angleless!!!
- Tutorial 2: What is a protractor???
- Tutorial 3: How to use a protractor
- Tutorial 4: Types of Triangles
- Tutorial 5: Angle Sum Property
- Tutorial 6: Complementary Angles
- Tutorial 7: Supplementary Angles

## Prototype

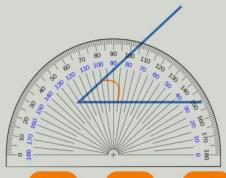
8

Measure the angle below by dragging the protractor.

Constraint : You can't rotate the protractor

HINT

Back



45° 50° 40° 35°

Measure the angle below by dragging the protractor.

Design the Right Thing  
Design Like a Right  
discover → define → iterate → develop → deliver

HINT

Back



45° 50° 40° 35°

Consistency: the levels are all consistent with how the questions are and how the protractor works

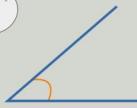
Prototype

8

Measure the angle below by dragging the protractor

Feedback: Gives the user feedback once a question is attempted

Correct!

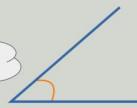


back

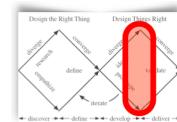
Measure the angle below by dragging the protractor

Incorrect : The answer was 40 degrees

Feedback : If you get the answer wrong , it gives a message



back



## Prototype

8

**BACK**

1: Place the center of the protractor on the red dot

Mapping: Arrows to make abstract concepts easier to understand

2: Measure from here to here and read the blue scale

**HINT**

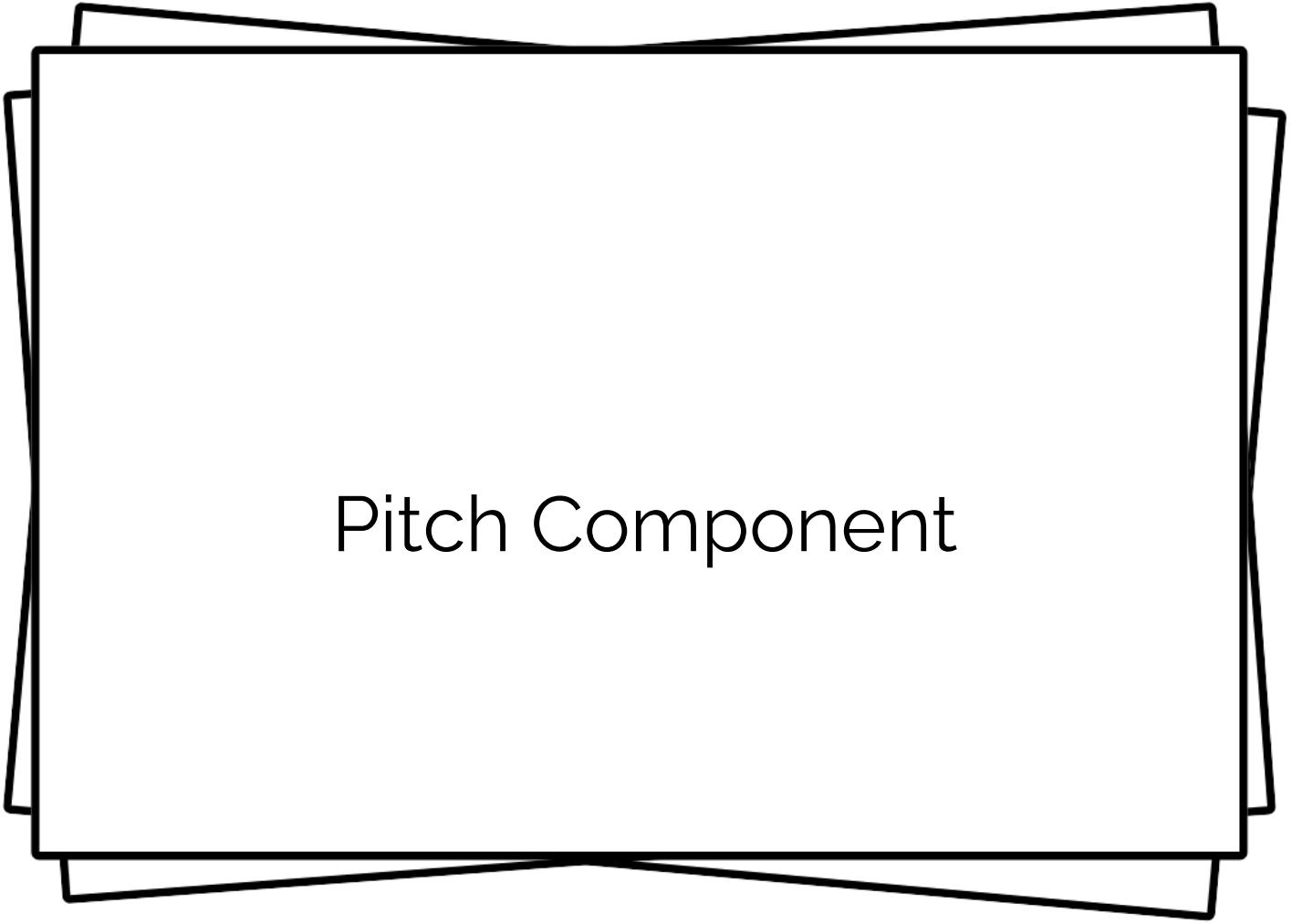
Signifier: tells the user that they have to choose one of these buttons to answer

**Back**

Measure the angle below by dragging the protractor.

Consistency : Back button on every level and tutorial

145°    130°    140°    135°



# Pitch Component

You have finally come to the conclusion and have a release version ready! You are now preparing a pitch to explain your idea. Next, you present your solution to a prototypical user.

# Meet Bob

Tell us about your prototypical user : Bob is a Grade 7 student studying the CBSE Indian Syllabus.

Your hypothetical customer, Bob, describe their background (target audience), their environment, their daily routine, anything of the sort to give a good description of who they are.

# Bob's Problem

Tell us their problem: Bob is having a tough time studying the Math taught by their teachers. Topics that need visualization is especially tough. Asking for question is difficult for Bob due to the online nature of classes. Bob also finds textbooks boring.

Now you're going to talk about their problem in detail. You could explain circumstances and any other factors that are important to the problem.

## We are...

Who are you and why do you think you care about solving Bob's problem: **We are First-year University Students that are tasked with fixing the problem that Bob and many others like Bob that study math in CBSE face with a software-oriented solution.**

Describe yourself, your reasoning and your motivations. Talk about your goal as a team, and how motivated you are to solve Bob's problem. But try to make it into a story, so it holds Bob's interest. Imagine you're writing a story for Netflix, or Disney+!

# Solution

Show how you solve the problem, with a voiceover of the app being used, but don't make it a walk-through of the app, or a tutorial, talk about how it solves the problem.

we created a fun game that students like Bob, and others in Grade 7 in the CBSE syllabus, can use to visualize and practice the use of a protractor aiding them in learning geometry, a significant portion of their Math curriculum.

Explain how it helps rather than what each button does. In reaching this step, you will have learned how to make the interface intuitive, so explaining the buttons and functionality is not your aim. You should instead explain how it helps solve the problem. For example, Bob had a problem: he found the textbook boring. The application solves this by making it in the form of a game with bright and catchy colors so the chance of being bored of the math is greatly reduced. Another example would be difficulty visualizing the problem. The application solves this by making sure the protractor is a free tool that lets you experiment visualize the protractor moving to where you need it.

# What we learned from DT

Duplicate your actual DT slides which help you explain your points and talk over them.

To avoid repetition, we have not copied the slides here, but you can talk about your journey in brief here with the help of your interviews charts and graph that you have plotted. Your explanation should convince the stakeholders that are listening to this pitch that there was a genuine problem and your solution has fixed it.

# Bibliography

## Articles

- [Blo+20] Nicholas Bloom et al. “Are ideas getting harder to find?” In: *American Economic Review* 110.4 (2020), pages 1104–44 (cited on page 194).

## Books

- [And07] John R Anderson. *How can the human mind occur in the physical universe?* Oxford University Press, 2007 (cited on page 200).
- [Ban13] Bela H Banathy. *Designing social systems in a changing world.* Springer Science & Business Media, 2013 (cited on page 201).
- [Nor13] Don Norman. *The design of everyday things: Revised and expanded edition.* Basic books, 2013 (cited on page 189).
- [SY21] Sanjay Sarma and Luke Yoquinto. *Grasp: The Science Transforming How We Learn.* Anchor, 2021 (cited on page 182).
- [Sim19] Herbert A Simon. *The sciences of the artificial.* MIT press, 2019 (cited on page 195).

# Index

containers, 121

duck typing, 121

free theorems, 120

polymorphism

ad hoc, 120

parametric, 120

subtype, 120

Simon, Herbert, 194

Stencil, 11