Planting the Seed: An Elm-Based Introductory Computer Science Curriculum for High School Students

Contents

1 Introduction **6**

2 The Elm Programming Language **7**

2.1 Description 7

2.2 Functional Reactive Programming 7

2.3 Elm in Education 7

3 Elm as an Introductory Language **8**

3.1 Simple Syntax 8

3.2 Straightforward Semantics 10

3.3 Helpful Error Messages 11

3.4 User-Friendly Programming Environment 13

3.5 Ability to Engage Students 14

4 An Elm-Based Introductory Computer Science Curriculum **16**

4.1 Topics 16

4.2 Lesson Plan Structure 18

4.3 Course Evaluations 19

5 Pilot Course **20**

5.1 Structure and Students 20

5.2 Preliminary Observations 20

6 Conclusion **22**

A Lesson Plans **27**

A.1 Drawing with Elm 27

A.2 Drawing and Functions 34
A.3 Functions and Variables 42
A.4 Types 49
A.5 Lists and Map 55
A.6 Conditionals 62
A.7 Mouse Signals 68
A.8 State and Pattern Matching 74
A.9 Paddle Game 82
A.10 Final Projects 92

B Modular Material **96**

B.1 Comments 96
B.2 Imports 96
B.3 `rgba` 97

C Homework Assignments **98**

C.1 Drawing with Elm 98
C.2 Drawing and Functions 98
C.3 Functions and Variables 99
C.4 Types 100
C.5 Lists and Map 100
C.6 Conditionals 101
C.7 Mouse Signals 101
C.8 State and Pattern Matching 102

D Pre/Post Survey **103**

Acknowledgements

Thank you to my advisor, Stephen Chong, for your invaluable guidance and support over the past year. I am so grateful to have had the chance to work with you, and this thesis certainly would not have been possible without you. Thank you to my readers, Stephen Chong, David Malan, Karen Brennan, and Paul Bamberg, for showing me what it means to be an outstanding teacher. Thank you for being a part of this project, and I look forward to receiving your feedback. Thank you to Rosemary White and my students at Boston University Academy for making the pilot program not only possible, but also fun and rewarding. Thank you to my amazing friends for an endless supply of encouragement, writing advice, and snacks. A special thank you to Tom Silver for helping to proof read the final product and for providing emotional support throughout the entire process. And finally, a huge thank you to my family. I am I so grateful for your love and support in everything I do.

1 Introduction

As of 2016, only a quarter of primary and secondary schools in the United States offer computer science courses with a programming component [31]. This proportion is far too low given the current industrial demand for computer scientists and the intrinsic value of a computer science education. Last year, fewer than 43,000 computer science students graduated into a field with over 600,000 open jobs [5]. Regardless of their ultimate career path, students benefit from the design thinking, creativity, and problem solving skills computer science develops. Research has shown that students who study computer science at the secondary level "demonstrate improved readiness for post-secondary studies" [34].

Administrators at the state and federal levels are realizing and responding to the need for computer science education in K-12 schools, where the pipeline[1] begins. In 2016, President Obama announced an initiative to dedicate over $4 billion to expand K-12 computer science education [31]. As of 2016, San Fransisco is phasing in computer science courses for all students, starting in pre-school [2]. By 2018, a yearlong computer science course will be a high school graduation requirement in Chicago. By 2025, every New York City public school will be required to offer courses in computer science [22].

As more high schools are required to offer computer science courses, more educational resources must be made available to teachers and students. Since computer science education at this scale is relatively new, we must be willing to experiment in order to discover curricula that lead to the best learning outcomes.

This thesis contributes to the freely and publicly available educational resources for introductory computer science courses. I present a high school-level introductory computer science curriculum that uses Elm. To my knowledge, it is the first high school-level course to use the Elm programming language. Given the untraditional language choice, I justify the use of Elm as an introductory language by identifying five features of an ideal introductory language and demonstrating that Elm exhibits each of these features.

The curriculum includes twelve hours worth of lesson plans, eight homework assignments, and a pre / post survey used to measure students' attitudes towards computer science. The curriculum's level of detail is such that a teacher with any level of experience with Elm can take advantage of it. I also developed `learn-elm.com/try`, an open source adaption of the Elm online editor. My adaption houses distribution code for the course and allows students to save and upload programs. All materials will be made freely and publicly available. I am teaching a pilot course using the materials and present preliminary observations that point to the success of the curriculum.

This thesis proceeds as follows. In Section 2, I describe the Elm programming language and its current use in computer science education. In Section 3, I explain why Elm is a good choice of an introductory language. In Section 4, I describe the structure and content of my

[1] The computer science pipeline refers to the educational path that students take towards computer science-related careers.

introductory curriculum. In Section 5, I discuss my experience teaching the introductory course to a group of high school students in Boston, Massachusetts. The Appendix includes lesson plans, modular material, homework assignments, and a pre / post survey.

2 The Elm Programming Language

The introductory computer science curriculum I present in this thesis uses the Elm programming language. In this section, I provide a high-level description of the language and the paradigms it supports. I also describe its current place in computer science education.

2.1 Description

Elm is a functional reactive programming language that allows developers to create simple, responsive graphical user interfaces (GUIs) for the Internet [9]. It was introduced in 2012 and has since cultivated an active developer community. Elm is used commercially by companies including Prezi and NoRedInk [8].

2.2 Functional Reactive Programming

Programming languages can generally be classified as functional or imperative. In functional languages, functions are treated as values. In other words, there is no meaningful difference between the treatment of a function (like $f(x) = x + 2$ in mathematics) and a value (like x or 2) [6]. Functional languages tend to be pure, which means that functions are self-contained (they have no knowledge of or effect upon their environment) and stateless (the same input will always result in the same output) [23]. Imperative is the opposite of pure.

Functional reactive programming (FRP) applies pure functional programming paradigms to time-varying values, known as signals. For example, the position of the user's mouse may be represented as a signal since the value may change over time. For more information on functional reactive programming and how it distinguishes Elm, see [9].

2.3 Elm in Education

Elm is not commonly used in formal educational settings. Two significant exceptions are courses offered by McMaster University Computing and Software Outreach and The University of Chicago. The former uses Elm in an introductory computer science course for middle school students [20]. The University of Chicago uses Elm in an undergraduate course on data structures [30]. To my knowledge, there are no high school-level computer science courses that use Elm.

There are also very few online resources that use Elm to introduce computer science concepts or principles. Many of the online Elm tutorials are out-dated, not available free of charge, or assume significant computer science knowledge.

The curriculum presented in this thesis is the first introductory computer science course for high school students that uses Elm. It will be made publicly and freely available online. Given Elm's absence in computer science education, I dedicate the following section to justifying its use in an introductory curriculum.

3 Elm as an Introductory Language

There is broad consensus that introductory computer science courses should not focus on the details of any particular programming language [3, 10, 11, 16]. Instead, courses should introduce core computer science principles and problem-solving skills. In an introductory course, a programming language should be the means of teaching content, not the content itself. As Demurjian et al. explain, the language serves "as both an explanation and demonstration vehicle for the various course topics. It provides the means through which students can obtain practical experience with the concepts discussed in class" [10].

In an introductory computer science curriculum, the ultimate goal is to teach language-independent principles; however, the choice of programming language can have enormous influence. A programming language can affect not only whether the principles are successfully conveyed, but also how they are perceived. Research by Schollmeyer et al. has found that introductory languages shape students' programming habits and problem-solving skills [26]. McLuhan's famous maxim from media theory—that the "medium is the message"—readily applies to programming. Just as natural language can never be completely removed from the medium of communication, computer science principles are colored by the programming languages that put them into practice [19].

In this Section, I justify the use of Elm in an introductory computer science course by identifying five features of an ideal introductory language. I outline each feature below, drawing on existing research to justify the importance of each, and demonstrate that Elm exhibits each feature.

3.1 Simple Syntax

The syntax of a programming language refers to the ways in which symbols can be combined to create programs. It provides a structural description of the "legal" expressions in the language. Unlike semantics (defined in Section 3.2), syntax does not consider the meaning of the symbols, only their structure [27]. The syntax of the English language specifies that there should be a period at the end of each sentence; the syntax of the Java programming language specifies that there should be a semicolon at the end of each statement.

A good introductory programming language has simple syntax. In particular, the syntax is minimal and easy to understand and adopt.

Mannila et al. found that "verbose and complex syntax" often causes novices "to focus on getting the syntax correct to such an extent that the algorithm becomes a secondary concern" [16]. In other words, complex syntax can distract students from the fundamental concepts the programming language is meant to convey. Simple syntax allows the primary emphasis of the course to remain on computer science principles and problem-solving skills [14].

Simple languages result in fewer syntax errors and, more importantly, fewer logic errors. Studies have also found that learning to program in a syntactically simple languages does not lead to disadvantages when moving to a more complex one [16].

Elm's Syntax

One of Elm's main features is its simple syntax. For example, Elm does not require the use of semicolons, and there are no parentheses around function arguments. These syntactical elements are found in many other languages, particularly imperative languages [8]. In general, modern functional languages tend to be very concise [3].

Traditionally, when a person learns a new programming language, he or she begins by writing a simple program that displays the phrase "Hello, world!" Consider the "hello world" program written in Elm:

```
import Html exposing (text)
main = text "Hello, world!"
```

Compare this to the "hello world" program in Java, the language currently used on the College Board's AP Computer Science A Exam [1]:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

Although the programs do not differ significantly in length, notice how many more symbols are required in the latter. In order to fully explain the syntax of the Java program, one must at least explain the use of braces, square brackets, parentheses, quotation marks, semicolons, and periods, and the key words `public`, `class`, `static`, `void`, and `main`. In order to fully explain the syntax of the Elm program, one only needs to explain the use of the equal sign, quotation marks, and parentheses, and the key words `import` and `main`.

Elm's simple syntax allows students to begin understanding and writing programs quickly

and reduces the chance that the syntax will distract from core content.

3.2 Straightforward Semantics

Semantics refers to the meaning of syntactically valid code. In natural languages, semantics connects phrases with objects, ideas, and experiences. In programming languages, semantics describes how a program executes [27].

A good introductory programming language has straightforward and accessible semantics so that a student can understand how the computer derives meaning (i.e. extracts instructions) from programs. Chakravarty et al. find that clean semantics allow teachers to explain programming techniques clearly. With straightforward semantics, reasoning about programs becomes "simple and natural" [3].

Elm's Semantics

Like most functional programming languages, Elm has straightforward and accessible semantics. In particular, students can understand how an Elm program works by manually simulating the stepwise execution of the program [3]. Since the focus of functional languages is on values and operations on values, the computational model is only a minor extension of the models of high school algebra [11].

Elm's semantics also allow for natural translations from algebraic functions to Elm expressions. For example, in the third lesson of the introductory course presented in this paper, students are asked to write a function that determines the end point of the hour hand on a clock, given the hour. Students can begin by writing the necessary algebraic equations to calculate the x and y coordinates of the end of the hour hand. If the length of the hour hand is 100, then the student might develop the following equations:

$angle = hour \times (360/12)$
$x = 100 \times sin(angle)$
$y = 100 \times cos(angle)$

In Elm, `let foo = bar in` assigns the variable `foo` to `bar` in all of the code in the function that follows the key word `in`. Knowing this, the formulas above translate easily into an Elm function that evaluates to the desired coordinate pair:

```
hourHandEnd hour =
  let angle = hour * (360/12) in
  let x = 100 * sin (degrees angle) in
  let y = 100 * cos (degrees angle) in
    (x,y)
```

For more context and a stepwise evaluation of this function, see Appendix A.3.

Elm's straightforward semantics allow students to quickly gain an understanding of how programs evaluate. This makes it easier for students to understand what a given program does and how to develop their own programs.

3.3 Helpful Error Messages

Error messages are an important part of a programmer's user experience. Ideally, error messages should help programmers progress towards working programs and should help programmers understand the problems that led to each error they encounter [18]. As Marceau et al. explain, error messages are especially critical tools for "novice programmers, who lack the experience to decipher complicated or poorly-constructed feedback" [17].

The infrastructure for an introductory programming language should be able to provide helpful error messages to users. Ideally, each message should use simple vocabulary, guide the user towards the location of the error, and provide information about why the error occurred [11, 18]. Both runtime errors (errors that occur while a program is executing) and compilation errors (errors that occur when the program is being put into a state that the computer can execute) should be clearly reported so that students can address and learn to avoid them [11].

Elm's Error Messages

Elm's error messages are extremely helpful and all occur at compile-time. Elm boasts no runtime exceptions: "Elm's compiler is amazing at finding errors... The only way to get Elm code to throw a runtime exception is by explicitly invoking `crash`" [8].

Suppose a beginner programmer forgets the closing quotation mark in the Elm "hello world" program introduced in Section 3.1. The Elm compiler would report the following error message:

```
SYNTAX PROBLEM
I ran into something unexpected when parsing your code!

4|   text "Hello, World!

I am looking for one of the following things:

    "\""
    "\\"
    "\n"
    "\r"
```

The error message provides the line on which the error occurs. The online programming environment described in Section 3.4 also includes a link to go directly to the error in the text editor [8]. The error message also tells the programmer the type of error that occurred (in this case, a syntax error) and, as best as it can, tries to point the programmer towards the fix (in this case, adding the first character suggested, a quotation mark).

The Elm compiler is also able to provide helpful error messages for more complex errors. For example, the fourth lesson asks students to consider the following program, which draws a red circle with a 100 unit radius at (0,0) in a 500 by 500 unit area:

```
1 import Color exposing(..)
2 import Graphics.Collage exposing(..)
3
4 main =
5   collage 500 500
6     [ circle 100
7         |> filled red
8         |> move (0,0)
9     ]
```

If `red` is replaced with `10`, the compiler reports the following error:

```
TYPE MISMATCH
The argument to function ‘filled’ is causing a mismatch.

7| |> filled 10

Function ‘filled’ is expecting the argument to be:

    Color

But it is:

    number
```

Again, the compiler reports the line on which the error occurs and the type of error. It identifies the type of the expected value (`Color`) and the type of the value it was given (`number`). This level of detail helps new programmers quickly identify and fix errors.

Elm's error messages are incredibly helpful for beginners, who are prone to mistakes. Without such detailed error messages, a small error can completely derail a student. The Elm compiler helps students get quickly back on track.

3.4 User-Friendly Programming Environment

In order to execute a program, it must be compiled (translated to machine code, which a computer can run directly) or interpreted (executed by another program). Often, a compiler or interpreter is made available to the programmer in a programming environment. The simplest of these environments may consist of a text editor and a command-line or graphical interface. More complex programming environments may incorporate additional tools, such as a debugger.

As Felleisen et al. found, the choice of introductory language should take into account the programming environment. The environment "should be a lightweight, easy-to-use tool. That is, it should provide just enough to edit and execute functions and programs, plus some tools for understanding fundamental concepts" [11]. If an environment is unnecessarily complex, teachers will have to take time away from teaching core content to help students learn how to use the tool. The programming environment should facilitate, not encumber, learning.

Elm's Programming Environment

Although one can easily install the Elm compiler locally, the online editor, shown in Figure 1, is a perfect programming environment for beginners. This editor is available at `elm-lang.org/try` and allows the programmer to compile and write code directly in the browser [8]. This is ideal for students since they can access the editor from any computer with an Internet connection, without installing anything or going through a configuration process.

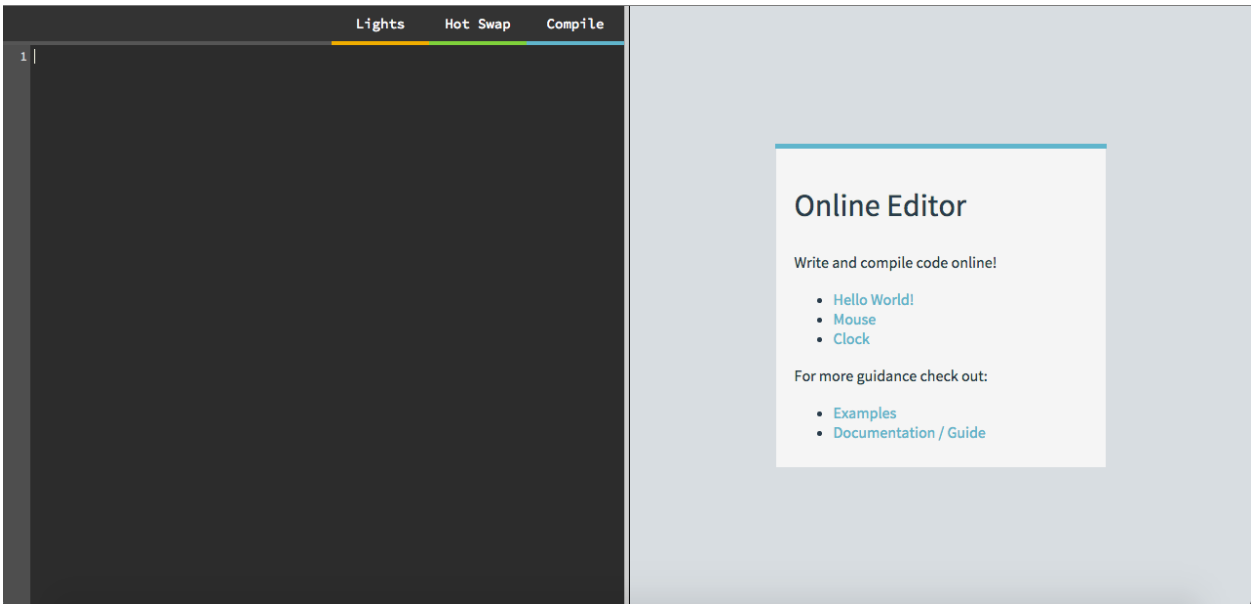


Figure 1: Elm online editor

Using the programming environment is also extremely straightforward. Users can type in an editor on the left side of the screen, press a compile button, and see the results of their program on the right side of the screen. The only other buttons are "lights," which changes the color scheme of the text editor, and "hot swap," which is an important feature for functional reactive programming. The latter button will not be particularly important for beginners, but it does not distract significantly from the more important features of the programming environment.

I created a slightly modified version of Elm's open source online programming environment for the introductory course presented in this paper. Two buttons, "save" and "upload," were added so that students could easily save their programs locally. The editor is available at `learn-elm.com/try`, and the code is available at `github.com/hblumberg/elm-lang.org`. This website also makes it possible to create and share distribution code for the course.

The Elm programming environment, as provided by the original Elm site and `learn-elm.com`, also provides "hints." If the user begins to type the name of a function that is defined in one of the core libraries, a link to the documentation appears above the editor. It does not disrupt the coding process, but it does provide an extremely helpful reference.

More advanced programming tools certainly exist in Elm. For example, Elm has a time-traveling debugger that allows developers to pause time, replay user inputs, and change code during execution [8]. These tools, however, are not primary components of the programming environment. The restricted set of tools available in Elm's online editor keeps the environment straightforward and accessible to beginners.

The online Elm programming environment strikes an ideal balance for an introductory language. It is a clean interface that provides helpful tools, but does not overwhelm or confuse the user.

3.5 Ability to Engage Students

An ideal introductory programming language would provide students with extrinsic or intrinsic motivation to learn computer science. One such extrinsic motivation may be the potential to use the programming language in industry or in future computer science courses [14].

An intrinsic motivation may be the potential for creative expression. Maeda explains, computers provide "a new material for expression" [15]. In their study of the Scratch programming language, Peppler and Kafai found that students can form "specific personal and epistemological connections to their work" when they are given the opportunity to be creative through programming [25].

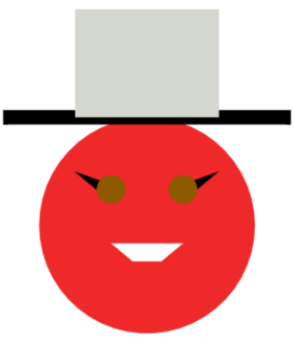
Elm's Ability to Engage Students

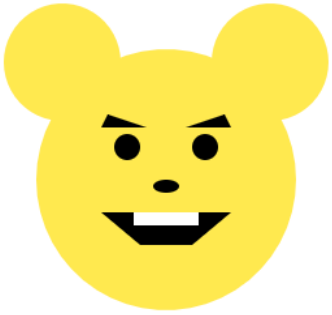
Usually a programming language provides students with extrinsic motivation if it is used in future classes or in industry. At the high school level, it is hard to cater to future classes since a student may not take his or her next computer science course until college and college courses use a wide variety of languages. Industry is also less immediately important at the high school level than it is at the college level, where students are more likely to seek summer internships and full-time employment.

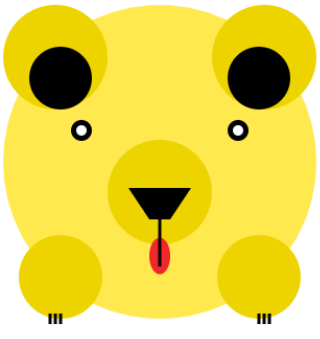
While Elm does not lend itself to significant extrinsic motivations, its emphasis on graphics provides particularly compelling intrinsic motivation. When learning to program in Elm, students immediately have the opportunity to create simple drawings, and they can quickly advance to interactive tools, animations, and games. Studies have found that game-based learning offers many benefits, most notably that it gives students the opportunity to engage with computer science in a familiar context [21]. It is both exciting and empowering for students to use computer science as a creative outlet.

When given the opportunity to be creative, students are often motivated to go beyond the "required" material. For example, the first lesson of the introductory curriculum presented here asks students to draw a smiley face and provides the following example:

Here are the smiley faces three of the students in the pilot course created:







In order to draw these more creative faces, the students learned about more complex shapes than the lesson requires. The opportunity to create their own drawings was enjoyable and

motivating for students, and in a student's words, they liked learning "how to apply computer language to real applications."

Elm is an engaging programming language because it gives students the opportunity to be creative.

4 An Elm-Based Introductory Computer Science Curriculum

In this section, I present an introductory computer science curriculum for high school students that uses Elm. The course consists of twelve hours worth of lessons and eight homework assignments. A complete set of lesson plans and assignments can be found in Appendices A and C respectively. The only background knowledge assumed in this course is a basic understanding of the Cartesian coordinate system.

The course is designed to teach students core computer science principles and problem-solving skills. (Specific topics are outlined in Section 4.1). It provides students with the skills necessary to program in Elm and explore other programming languages, independently or in future courses. The course is also designed to foster students' creativity. Students should leave the course excited about computer science and comfortable with their ability to approach the subject.

I offered this course in the spring of 2016 as a weekly after-school program at a high school in Boston, Massachusetts. I modified the lesson plans based on informal and formal student feedback. I provide further reflections on the course in Section 4.3.

4.1 Topics

The topics of the first eight one-hour lessons are described briefly below. The remaining two lessons, each two hours in length, are dedicated to a self-guided lesson on game development and a free-form final project, respectively. Lesson plans are available in full in Appendix A.

Although the curriculum is short, it covers all of the "core" features of a functional language. According to Felleisen et al., "all a beginner needs [in a functional language] are [sic.] function definition, function application, variables, constants, a conditional form, and possibly a construct for defining algebraic types" [11]. A complete list of topics that ought to be covered by an introductory curriculum is outside the scope of this paper; however, I briefly justify each of the topics I chose to include and explain why I chose to exclude certain potential topics.

- **Lesson 1: Drawing with Elm**

This lesson illustrates the importance of precision (unambiguous instructions) in computer science and introduces students to the Elm programming language. Students leave feeling comfortable creating basic drawings with Elm.

Precision is one of the most fundamental computer science principles as it helps to explain how and why programs are developed. This lesson purposely glosses over topics such as functions, which are introduced but not explained in depth. These topics are revisited and explained in later lessons. The material covered in the first lesson is limited so that students can begin programming themselves as quickly as possible. The emphasis on drawing is meant to pique students' interest and motivate them to continue the course.

- **Lesson 2: Drawing and Functions**

  This lesson teaches students more advanced drawing commands and introduces the concept of functions. Students have the opportunity to define functions and create their own pictures through independent practice.

  This lesson demystifies the basic drawing functions used in the first lesson. Students learn how to use and create their own functions and are formally introduced to arguments. In order to continue an emphasis on graphics and creative expression, the functions that students develop allow them to draw new shapes by combining existing functions.

- **Lesson 3: Functions and Variables**

  This lesson teaches students how to use functions for computation. It also introduces students to the concept and use of variables.

  Because functions are such a fundamental topic in functional languages and computer science more generally, two lessons are devoted to teaching them. While the previous lesson focuses on developing functions that create drawings, this lesson focuses on developing functions that perform computations. Variables are implicitly introduced in previous lessons through arguments, but this lesson explains their use more generally.

- **Lesson 4: Types**

  This lesson introduces students to Elm's type system. Students learn about primitive types, function types, partial application, and type annotations.

  An introductory course could avoid the explicit introduction of types; however, a basic understanding of types helps students determine which operations are valid for a given value. Furthermore, since type annotations serve as a form of documentation, students can more easily learn how to use new functions independently.

- **Lesson 5: Lists and Map**

  This lesson provides students with a more complete understanding of lists, including their purpose and their properties. It also introduces the `map` function, which transforms elements in a list.

Lists and `map` are fundamental topics since they allow for repeated computation and provide an important basis for signals. Since a signal is a value that may change over time, it can be conceptualized as an infinite list of values. Recursion (another important technique for repeated computation) is notably absent from the curriculum and would certainly be included if time permitted. Lists and `map` were included in favor of recursion because of the foundation they create for signals, which in turn allow for the creation of interactive programs.

- **Lesson 6: Conditionals**

  This lesson introduces students to conditionals in Elm. They have the opportunity to use conditionals in both graphical and non-graphical programs.

  Conditionals are one of the "core" topics identified by Felleisen et al. [11]. This lesson follows lists and `map` since it allows students to apply more complex and interesting functions to lists of values. During this lesson, students work though FizzBuzz, a traditional technical interview question that is meant to test problem-solving skills [13].

- **Lesson 7: Mouse Signals**

  This lesson introduces students to signals, focusing on mouse signals. Mouse signals are time-varying values that provide information about the user's mouse; these signals include the position of the mouse and whether or not the mouse is being clicked. Students learn how to use signals to create interactive programs and get further practice with `map` and conditionals.

  This lesson introduces the most fundamental principle of functional reactive programming. While not a general computer science principle itself, signals are a form of I/O (input/output), which is commonly regarded as a fundamental topic [12]. Signals are particularly exciting for students because they can begin to make interactive programs.

- **Lesson 8: State and Pattern Matching**

  This lesson introduces students to `Signal.foldp`, a function that enables programs to accumulate state. Students also work with record types and pattern matching to simplify their increasingly complex programs.

  State and pattern matching are the final topics required to create complex interactive programs. The lesson builds to the creation of a Paint-like program that allows the user to draw using their mouse and change color using the keyboard.

4.2 Lesson Plan Structure

Each lesson plan begins with a high-level objective, a list of key words and concepts, and a specific list of assessments. The assessments are meant to measure the degree to which students learn the material covered in the lesson. They consist of in-class exercises and homework assignments.

The majority of each lesson plan is dedicated to a "timeline." This section includes lecture components and in-class exercises. The lecture components are meant to read like scripts. This format is ideal as it caters to a wide variety of potential users. For example, a computer science teacher who has experience with Elm might only use the lesson plans as a starting point for his or her own lessons. A teacher with no prior experience may learn the language him or herself by working through the curriculum before using the lesson plans to teach his or her students. An independent learner could similarly use the lesson plans to learn Elm outside a formal educational setting.

Following the three step process proposed by Wiggins et al., I developed each lesson's assessments first and used them to guide the development of the lecture components [33].

4.3 Course Evaluations

The course objectives described at the start of Section 4 might be summarized by two general objectives: (1) to provide students with fundamental computer science knowledge and skills and (2) to increase students' interest in computer science.

The first objective can be measured through in-class exercises, homework assignments, exams, and/or final projects. The course presented here primarily takes advantage of in-class exercises and final projects to assess learning. Since the course was piloted as an optional after-school program, there was not enough time to allow for an exam. In addition, mandatory homework could not be assigned. An optional assignment was distributed after each lesson, but as expected, only a few students opted to complete it each week.

The second objective, to increase students' interest in computer science, is more difficult to measure. A thorough evaluation might consist of a longitudinal study of each student in the course. A student might demonstrate interest in computer science by completing more advanced courses, studying computer science independently, or pursuing a career in a computing-related field. Since this sort of evaluation would be costly and time-consuming, a course could instead be evaluated by pre and post surveys. These surveys would ask students to rate their level of comfort and confidence and interest in computer science. The responses before and after the course would be compared; an ideal computer science course would increase each student's comfort, confidence, and interest whenever possible.

The course presented here uses a pre and post survey. This survey was adapted from Wiebe et al.'s *Computer Science Attitude Survey* [32] and is available in Appendix D.

5 Pilot Course

5.1 Structure and Students

I am teaching the introductory course presented in Section 4 to a class of ten high school students at Boston University Academy, a private school in Boston, Massachusetts. The course meets once a week for an hour after school. Participation is not mandatory for any students, all assignments are optional, and no grades are assigned. At the time of writing, half of the classes have been taught (i.e. up to and including lesson 6).

The students in the course chose to participate after hearing a short presentation on the Elm programming language. Of the ten students, nine are in tenth grade, and one is in eleventh. There are five female students, four male students, and one non-binary student.

5.2 Preliminary Observations

Here, I offer some preliminary observations from teaching the first half of the pilot course. I organize my observations according to the features identified in Section 3.

Simple Syntax and Straightforward Semantics

Elm's syntax required little time to explain thoroughly. Although students encountered syntax errors, these errors were usually a result of mistyping rather than misunderstanding and were therefore resolved quickly and easily.

Elm's straightforward semantics allowed me to explain programs by manually simulating their executions. I also encouraged students to perform stepwise evaluation themselves to find and fix logic errors in their programs. This is a useful technique since it helps students understand the cause of each problem.

Helpful Error Messages

While students worked on in-class exercises, I observed their interactions with error messages from the Elm compiler. Students consistently read error messages and attempted to address issues using the given information. Most of the time, the compiler correctly identified both the location and the source of the error. This allowed the students to find and fix their errors quickly.

Although the error messages were largely helpful in guiding students towards resolving errors, there remains room for improvement. In particular, students were confused by error messages that specified lines that didn't contain errors. For example, consider the following

program, which is missing a closing parenthesis (or has an unnecessary opening parenthesis) on line 4:

```
1 main = show (getParity 42)
2
3 getParity n =
4   if (n % 2 == 0
5     then "even"
6   else "odd"
```

Attempting to compile this program results in the following error message:

```
SYNTAX PROBLEM
I ran into something unexpected when parsing your code!

5|     then "even"

I am looking for one of the following things:

    a closing paren ')'
    whitespace
```

The compiler specifies line 5 since it expects a closing parenthesis before the key word `then`. This, however, is confusing for students since they are led to believe the correction must also be made on line 5.

I observed a student as she responded to this error message in lesson 6. She began by looking at line 5, but when was unable to find any errors, she told me she would "just give the computer what it's looking for." Following the error message's suggestion, she first changed line 5 to `then "even")`. When that resulted in the same error, she added spaces at the end of the line. Interactions like the one described here were not uncommon in the first several weeks of the course.

While this example demonstrates that Elm error messages could be improved, students found them to be helpful in the majority of cases.

User-Friendly Programming Environment

Students did not have any difficulties working with the programming environment at `learn-elm.com/try`. Since several students used school-owned computers during the course, it was crucial that the environment could be used from the browser and did not require them to download or configure software. The introduction to the environment was brief, and students did not have any questions about the environment in the first six weeks.

Several changes could be made to the programming environment to enhance the teaching and learning experiences. For teachers, it would be helpful to have a graphical user interface to add distribution code. Currently, only someone with access to the `learn-elm.com` server can add new distribution code. For students, it would be helpful to be able to save and upload programs to and from the server. Currently, programs can only be saved to and uploaded from local storage.

Ability to Engage Students

Elm's emphasis on graphics provides a great deal of opportunity for students to be creative. The first six lessons give students the opportunity to create increasingly complex drawings. Starting with lesson 7, students begin learning the skills necessary to create interactive tools, animations, and games.

I found that students were the most excited and immersed in their work when they had the opportunity to customize their drawings. Students enjoyed sharing their work with classmates and were more motivated to complete goals they set for themselves than tasks assigned to them.

Students were also surprisingly motivated to learn skills that might help them pursue job opportunities. During lesson 6, I mentioned that FizzBuzz (one of the in-class exercises) is a common interview question for jobs in software engineering [13]. The students found this exciting and were more motivated to complete the exercise as a result.

I anticipate that students will be most excited about the last two lessons, which give them the opportunity to work on a final project of their choice.

6 Conclusion

This thesis contributes to the freely and publicly available educational resources for introductory computer science courses. I present a high school-level curriculum that uses the Elm programming language. I justify the unusual choice of Elm as an introductory language by identifying five desirable features of an introductory language and demonstrating that Elm exhibits each. The curriculum includes twelve hours worth of lesson plans, eight homework assignments, and infrastructure including an adaption of the Elm online editor. All materials and tools will be made publicly and freely available and are designed for use by teachers with any level of experience with the Elm language. I am teaching this curriculum in a pilot course with ten high school students in Boston, Massachusetts. I present preliminary observations, which point towards the success of the curriculum.

Future work includes soliciting feedback on the curriculum from the Elm community and improving it accordingly. I also hope to expand the curriculum beyond its current length,

ideally creating a yearlong course. I will also improve `learn-elm.com` based on the observations described in Section 5.2.

Computer science is a rapidly growing and changing field, but educational resources lag behind. Policymakers, educators, and computer scientists must help to bring the innovation that characterizes computer science as a whole to computer science education. This thesis serves as a starting point for experimentation in introductory computer science courses, a starting point for the creation of educational material using Elm, and a starting point for ten high school students' computer science educations.

References

[1] AP Central. (2016). *AP Computer Science A Course Home Page.* [online] Available at: http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/4483.html [Accessed 30 Mar. 2016].

[2] Blythe, G. (2016). *Board Approves Plans to Expand Computer Science Curriculum to All Grades.* [online] San Francisco Public Schools. Available at: http://www.sfusd.edu/en/news/current-news/2015-news-archive/06/board-approves-plans-to-expand-computer-science-curriculum-to-all-grades.html [Accessed 19 Feb. 2016].

[3] Chakravarty, M. and Keller, G. (2004). The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(1), pp.113-123.

[4] Chong, S. (2015). *Lecture 24: Functional Reactive Programming.* [online] Harvard School of Engineering and Applied Sciences – CS 152: Programming Languages. Available at: http://www.seas.harvard.edu/courses/cs152/2015sp/lectures/lec24-frp.pdf [Accessed 8 Mar. 2016].

[5] Code.org, (2016). *What's wrong with this picture?.* [online] Available at: https://code.org/promote [Accessed 19 Feb. 2016].

[6] Cousineau, G. and Mauny, M. (1998). *The functional approach to programming.* Cambridge, U.K.: Cambridge University Press.

[7] CSTA K-12 Computer Science Standards. (2011). [online] Computer Science Teachers Association. Available at: http://csta.acm.org/Curriculum/sub/CurrFiles/CSTA_K-12_CSS.pdf [Accessed 4 Mar. 2016].

[8] Czaplicki, E. (2016). *Elm* [online] Elm-lang.org. Available at: http://elm-lang.org [Accessed 18 Feb. 2016].

[9] Czaplicki, E. and Chong, S. (2013). Asynchronous functional reactive programming for GUIs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 48(6), p.411.

[10] Demurjian, S., Peters, T., Beshers, G., Ellis, H. and Nichols, G. (1992). The (Non)Importance of a Programming Language in a Software Engineering Course. *Computer Science Education*, 3(1), pp.35-52.

[11] Felleisen, M., Findler, R., Flatt, M. and Krishnamurthi, S. (2004). The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4), pp.365-378.

[12] Felleisen, M., Findler, R., Flatt, M. and Krishnamurthi, S. (2009). A functional I/O system or, fun for freshman kids. *International Conference on Functional Programming*, 44(9), p.47.

[13] FizzBuzz Practice Code. (2013). [online] Codingbat. Available at: http://codingbat.com/doc/practice/fizzbuzz-code.html [Accessed 7 Mar. 2016].

[14] Kiper, J. and Abernethy, K. (1996). Language Choice for CS1 and CS2: Experiences from Two Universities. *Computer Science Education*, 7(1), pp.35-51.

[15] Maeda, J. (2014). *Creative Code: Aesthetics + Computation.* Thames & Hudson.

[16] Mannila, L., Peltomki, M. and Salakoski, T. (2006). What about a simple language? Analyzing the difficulties in learning to program. *Computer Science Education*, 16(3), pp.211-227.

[17] Marceau, G., Fisler, K. and Krishnamurthi, S. (2011). Measuring the effectiveness of error messages designed for novice programmers. *ACM Technical Symposium on Computer Science Education.*

[18] Marceau, G., Fisler, K. and Krishnamurthi, S. (2011). Mind your language: On novices' interactions with error messages. *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software.*

[19] McLuhan, M., Fiore, Q. and Agel, J. (1967). *The medium is the massage*. New York: Bantam Books.

[20] McMaster University Computing and Software Outreach. (2015). *Elm Tutorials*. [online] Available at: http://outreach.mcmaster.ca/menu/tutorials.html [Accessed 29 Mar. 2016].

[21] Microsoft Research, (2008). *Transforming Computer Science in the Gaming Age.* [online] Available at: http://research.microsoft.com/en-us/collaboration/papers/usc_uwb_rit.pdf [Accessed 28 Feb. 2016].

[22] Miller, K. (2015). *De Blasio to Announce 10-Year Deadline to Offer Computer Science to All Students*. [online] Nytimes.com. Available at: http://www.nytimes.com/2015/09/16/nyregion/de-blasio-to-announce-10-year-deadline-to-offer-computer-science-to-all-students.html?referrer=&_r=3 [Accessed 18 Feb. 2016].

[23] Msdn.microsoft.com, (2016). *Functional Programming vs. Imperative Programming.* [online] Available at: https://msdn.microsoft.com/en-us/library/bb669144.aspx [Accessed 28 Feb. 2016].

[24] Observable (RxJava Javadoc 1.1.1). (2016). [online] Reactivex.io. Available at: http://reactivex.io/RxJava/javadoc/rx/Observable.html [Accessed 9 Mar. 2016].

[25] Peppler, K. and Kafai, Y. (2016). *Creative Coding: Programming for Personal Expression*. [online] Available at: https://download.scratch.mit.edu/CreativeCoding.pdf [Accessed 27 Mar. 2016].

[26] Schollmeyer, M. (1996). Computer programming in high school vs. college. *Proceedings of the 27th SIGCSE technical symposium on CS education*, 28(1), pp.378-382.

[27] Slonneger, K. and Kurtz, B. (1995). *Formal syntax and semantics of programming languages*. Addison-Wesley Publishing Company.

[28] Su, S. (2010). *Exploring Computer Science, Lecture 2*. [online] Tufts University - Department of Computer Science. Available at: http://www.cs.tufts.edu/ sarasu/courses/comp9-2010sp/pdf/comp9-02-variables-types.pdf [Accessed 7 Mar. 2016].

[29] The definition of preciseness. (2016). [online] Dictionary.com. Available at: http://dictionary.reference.com/browse/preciseness [Accessed 4 Mar. 2016].

[30] The Department of Computer Science at the University of Chicago. (2016). *CS 223: Functional Programming*. [online] Available at: https://www.classes.cs.uchicago.edu/archive/2016/winter/22300-1/index.html [Accessed 29 Mar. 2016].

[31] whitehouse.gov, (2016). *Computer Science For All*. [online] Available at: https://www.whitehouse.gov/blog/2016/01/30/computer-science-all [Accessed 18 Feb. 2016].

[32] Wiebe, E., Williams, L., Yang, K. and Miller, C. (2003). *Computer Science Attitude Survey*. [online] North Carolina State University. Available at: http://www4.ncsu.edu/w̃iebe/www/articles/prl-tr-2003-1.pdf [Accessed 1 Mar. 2016].

[33] Wiggins, G. and McTighe, J. (2005). *Understanding by design.* Alexandria, VA: Association for Supervision and Curriculum Development.

[34] Wilson, C., Sudol, L., Stephenson, C. and Stehlik, M. (2010). *Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age.* [online] Available at: http://runningonempty.acm.org/fullreport2.pdf [Accessed 29 Mar. 2016].

A Lesson Plans

Note on formatting: As described in Section 4.2, the following lesson plans are meant to read like a script. Sections that are written in italics are included as notes to the teacher and should not be read aloud.

A.1 Drawing with Elm

Objective: This lesson illustrates the importance of precision (unambiguous instructions) in computer science and introduces students to the Elm programming language. Students will leave feeling comfortable creating basic drawings with Elm.

Key Words and Concepts: computer science, programming language, Elm, precision, code, program

Assessments:

- Define or describe: "programming language."
- Write precise instructions for how to draw an image.
- Explain what a simple Elm program does.
- Draw a smiley face using Elm.

Timeline:

[0:00 - 0:10]

This semester, we are going to learn about **computer science** using a **programming language** called **Elm**.

Before we get started, please take 10 minutes to fill out a quick survey at `learn-elm.com/csas-survey`.

[0:10 - 0:15]

Computer science is the study of computers and algorithms, including their principles, their designs, their applications, and their impact on society [7].

Programming languages are languages that allow us to communicate instructions to computers. Programming languages differ from human languages in that they are more structured and less permissive. In the same way that there are many human languages (English, Spanish, French, etc.), there are also many programming languages (C, Java, Python, etc.).

Elm is a programming language that was created by a Harvard student named Evan Czaplicki in 2012. This language is especially well-suited for creating graphics for the web. We will be using Elm to create drawings, animations, tools, and games as we learn some fundamental

computer science concepts.

Let's take a look at some websites and games that were made using Elm to get a sense of what kind of projects we'll be working towards:

- To-do list application: `http://todomvc.com/examples/elm/`
- Breakout: `http://daiw.de/games/breakout/`
- Maze: `http://daiw.de/games/maze/`
- Concentration: `http://daiw.de/games/demoscene-concentration/`
- Froggy: `http://thsoft.github.io/froggy/`

[0:15 - 0:25]

Now that we've seen some Elm programs, let's get started on our own! We'll start by learning how to use Elm to draw pictures. Let's think about how we might draw a smiley face on the board.

Divide the board in half; use the left side for instructions and the right side for drawings. This will mimic the style of the editor. Ask the students to give you instructions on how to draw the smiley face. Collect a complete set of instructions before drawing.

When you begin drawing, misinterpret students' instructions at every opportunity. For example, if a student tells you to draw a circle for one of the eyes, draw a circle that is too big or in the wrong position. In particular, have your misinterpretations emphasize the importance of specifying the size and location of the shapes.

If time permits, allow students to change their instructions until the smiley face is drawn correctly. At the end of the exercise, ask the students what lessons they took away. The goal is for students to recognize the importance of giving unambiguous instructions.

There are many different programming languages, but all enable and require you to write **precise** instructions. Precise means "definitely or strictly stated, defined, or fixed" [29]. In other words, precise instructions are unambiguous and leave no room for interpretation. We have to be precise when giving instructions to a computer because if there is more than one way for it to interpret our instructions, it will not know how to proceed.

If we want to draw a shape using Elm, we need to specify:

- the shape
- the size
- the color
- the location

[0:25 - 0:35]

Let's draw a smiley face, starting with a large yellow circle in the center of the screen. Navigate to `learn-elm.com/examples/smile`.

You'll notice that the screen is split in half (just like the board). On the left side of the screen, we have a text editor where we can write instructions. At the top of this editor is a "compile" button, which is what we'll press when we're ready for the computer to execute the instructions.

The right side of the screen is essentially the computer's canvas. Today, we'll be instructing the computer to draw shapes; the computer will do so on the right half of the screen.

Here are the instructions that draw a large yellow circle:

```
import Color exposing (..)
import Graphics.Collage exposing (..)

main =
  collage 300 300
    [ circle 100
        |> filled yellow
        |> move (0,0)
    ]
```

Let's walk through these instructions line by line to get a better understanding of what's going on.

The first two lines tell the computer that we're going to be using something called `Color` and something called `Collage`. We'll see where we use each in just a moment. In future weeks, we'll discuss exactly how these two lines work, but for now we can understand them at a high level.

`main` is a special word in Elm. It tells the computer where the instructions start. We'll always have to have the word `main` somewhere in our instructions. Next we have `collage 300 300`. This tells the computer that we will be drawing in an area that is 300 by 300 units.

Draw a large square on the board to illustrate.

Now the computer will treat this area as a Cartesian coordinate system. This will allow us to specify exactly where we want to draw our shapes.

Draw axes for reference.

Now the computer is going to draw any shapes listed between the square brackets after `collage 300 300`. Here, we're asking the computer to draw a circle. We do this with the word `circle`. We then specify the radius of the circle, in this case `100`. This ensures that the computer will know what size to make the circle.

We need to specify two more things: the color and the location. You'll notice we have the characters `|>`. This tells the computer to apply the following operations to the circle we're drawing. `filled yellow` tells the computer we want to color the circle yellow. Finally, `move (0,0)` tells the computer that we want the center of the circle to be at position (0, 0), the

origin of the coordinate system.

Let's pretend to be the computer and execute these instructions on the board. We will start at the origin then draw a circle with radius 100.

Draw the circle on the board, remembering that the area is 300 by 300. It might be helpful to label the axes where they meet the area's boundary. Clockwise, starting from the top: (0, 150), (150, 0), (0, -150), (-150, 0). Color the circle yellow.

When we press "compile," the computer reads and executes the instructions just like we did on the board.

Notice that the instructions we gave the computer left nothing up to interpretation. Another word for these precise instructions is **code**. When code is put together to accomplish some task, in this case drawing a yellow circle, it is called a **program**.

Congratulations, we've written our first Elm program!

[0:35 - 0:40]

Suppose we want to change our smiley face from yellow to red. All we would have to do is change `yellow` to `red` and press "compile."

Take a few minutes to try changing the color again. Then try making the circle bigger. What's the biggest you can make the circle without cutting off part of the shape? Why?

Reconvene and discuss the answers. In order to increase the size of the circle, change 100 to a number greater than 100 and less than or equal to 150. 150 is the maximum radius of the circle because the collage is 300 units wide and tall.

[0:40 - 0:50]

Now we have our smiley face's face, let's add two eyes and a mouth. We can add the first eye by adding another circle to the collage. Let's add a new line after `move (0,0)` and add a comma. Then we can copy and paste the code that drew the first circle. Now our code looks as follows:

```
main =
  collage 300 300
    [ circle 100
        |> filled yellow
        |> move (0,0)
    , circle 100
        |> filled yellow
        |> move (0,0)
    ]
```

It's very important that each shape is separated by a comma. If you forget this comma, the computer will not understand that we are trying to draw multiple shapes. After checking

that we have a comma between the circles and that we don't have any typos, we can press "compile." Even though our program executes successfully (we don't see any errors), our drawing hasn't changed. This is because the computer is just drawing two identical yellow circles directly on top of one another.

Let's try making the second circle smaller and black, so it looks more like an eye. Now our code is as follows:

```
main = :
  collage 300 300
    [ circle 100
        |> filled yellow
        |> move (0,0)
    , circle 15
        |> filled black
        |> move (0,0)
    ]
```

If we press "compile" again, we'll see a small black circle on our original yellow circle. What would have happened if we had switched the order of these two shapes? Our code would be:

```
main =
  collage 300 300
    [ circle 15
        |> filled black
        |> move (0,0)
    , circle 100
        |> filled yellow
        |> move (0,0)
    ]
```

If we press "compile," we'll see that the black circle disappears! This is because the computer follows our instructions exactly. It draws the shapes in the order that they appear in our code. So it draws a small black circle first then draws the bigger yellow circle on top. If we could peel away the yellow circle, we'd see the black circle hiding underneath.

Returning to our correct code, we notice that our eye is directly in the center of the circle. How can we move it left and up?

Let's start by moving it to the left. Right now, we're drawing the circle at (0, 0). We know that going left on the Cartesian coordinate system means decreasing the value of x. So let's try changing `(0,0)` to `(-30,0)`. If we press "compile," we'll see that the eye moved to the left, as we hoped!

We also know that going up on the Cartesian coordinate system means increasing the value of y. So let's try changing `(-30,0)` to `(-30,30)`. If we press "compile" again, we'll see that the eye moved up, as we hoped!

Our complete code is as follows:

```
main =
  collage 300 300
    [ circle 100
        |> filled yellow
        |> move (0,0)
    , circle 15
        |> filled black
        |> move (-30,30)
    ]
```

[0:50 - 1:00]

Take the remaining time to try adding a second eye to your smiley face.

If you have time left over, try to add a trapezoid for the mouth. The following instruction will create a black trapezoid at position (0, 0):

```
polygon [(-50,25), (-20,0), (20,0), (50,25)]
    |> filled black
    |> move (-30,30)
```

Other shapes you can try out are `rect` and `oval`. For each, you have to specify the desired width and height. For example if you wanted a rectangle 10 units wide and 5 units high, you would write: `rect 10 5`

Ultimately, your smiley face should look something like this:

but feel free to get creative!

Finished code:

(Available at: `learn-elm.com/examples/finished-smile`.)

```
import Color exposing(..)
import Graphics.Collage exposing(..)

main =
  collage 300 300
    [ circle 100
        |> filled yellow
        |> move (0,0)
```

```
, circle 15
    |> filled black
    |> move (-30,30)
, circle 15
    |> filled black
    |> move (30,30)
, polygon [(-50,25), (-20,0), (20,0), (50,25)]
    |> filled black
    |> move (0,-60)
]
```

A.2 Drawing and Functions

Objective: This lesson teaches students more advanced drawing commands and introduces the concept of functions. Students will have the opportunity to define functions and create their own pictures through independent practice.

Key Words and Concepts: function, argument

Assessments:

- Define or describe: "function" and "argument."
- Write a simple Elm function.
- Draw a house using Elm.

Timeline:

[0:00 - 0:10]

Last week, we learned how to use Elm to draw a smiley face. We worked together to draw the face and the left eye with the following code.

```
main =
  collage 300 300
    [ circle 100
        |> filled yellow
        |> move (0,0)
    , circle 15
        |> filled black
        |> move (-30,30)
    ]
```

You then had the opportunity to add the right eye and the smile yourself. To draw the right eye, we'll have to create another small black circle. We can add the following shape to our list:

```
circle 15
  |> filled black
  |> move (30,30)
```

We know from experience that `circle 15` will draw a circle with a radius of 15 units. If we had instead written `circle 20`, the computer would have drawn a circle with a radius of 20 units. In general, `circle` always draws a circle, and the number that follows specifies the radius.

`circle` is an example of a **function** in Elm. Functions are simply pieces of code that take in some input (also known as **arguments**) and perform some action.

In this case, the name of the function is "`circle`." The input it takes in is the radius of the

circle, and the action it performs is creating a circle of the specified radius.

At the end of last week's lesson, we briefly mentioned three new shapes: `rect`, `oval`, and `polygon`. Like `circle`, these are also functions. `rect` and `oval` each take two arguments: the width and height of the desired shape. `polygon` takes one argument: a list of points that define the vertices of the polygon.

Let's see if we can use the `polygon` function to make a triangle. Since a triangle has three vertices, we need three points. Suppose we put the first vertex 30 units above the origin. We can represent this point as (0, 30). Let's put the second vertex 15 units to the left of the origin. This point is (-15, 0). Let's put the third and final vertex 15 units to the right of the origin. This point is (15, 0).

Just like when we create a list of shapes, we're going to put these points inside square braces and separate them with commas. In this case, we would write: `[(0,30), (-15,0), (15,0)]`

This will be our the input to the polygon function. So `polygon [(0,30), (-15,0), (15,0)]` will create a triangle.

[0:10 - 0:20]

Two other functions that we encountered last week are `filled` and `move`.

`filled` takes two arguments: the color that we want to color the shape and the shape itself. So if we wanted to create a black circle with radius 15, we could write:

```
filled black (circle 15)
```

In math, the order of operations (PEMDAS) dictates that operations within parentheses happen first. It works the same way in Elm. So `(circle 15)` first creates a circle of radius 15. This circle is then used as input to `filled`, which creates a black circle.

In our smiley face code, however, we wrote this a little differently:

```
circle 15
  |> filled black
```

If we wanted to write this on one line, it would look like:

```
(circle 15) |> filled black
```

Remember that we said the characters `|>` apply the following function to the circle. So the following two lines of code are equivalent:

```
(circle 15) |> filled black

               filled black (circle 15)
```

Since these statements are the same, we can choose whichever one makes our code easier to read.

Similarly, `move` is a function that takes a point (coordinate pair) and a shape to move to that point. If we want to move a black circle with radius 15 to the point (-30, 30) then we could write:

```
move (-30,30) (filled black (circle 15))
```

When the computer sees this line of code, it first creates a circle of radius 15, then creates a black circle of radius 15, then creates a black circle of radius 15 centered at (-30, 30). You can start to see how this format might get messy or confusing. The following structure is equivalent, but it makes the order of operations much more clear:

```
circle 15
  |> filled black
  |> move (-30,30)
```

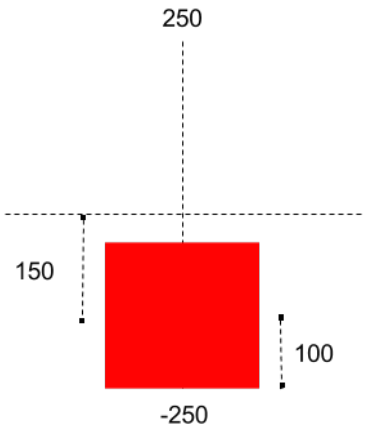
`collage` is also a function. What are its arguments? *A number for the width, a number for the height, and a list of shapes.* What does the function do? *Draws the specified shapes in an area specified by the width and height.*

[0:20 - 0:30]

Let's get back to using Elm to draw. Suppose we want to draw a house. To give ourselves plenty of space, let's start with a collage that is 500 by 500 units.

Let's start by making the base of the house a red square. We could use the rectangle function we introduced last week, but we can use an even simpler function, `square`. This function takes one argument, the length of an edge, and creates a square.

If we want a square that is 200 by 200 units appearing at the bottom of the collage, we should put the center of the square 100 units above the bottom of the collage. So the center of the square should be at (0, -150). Whenever we're unsure how to move our shape exactly where we want it, we can sketch it out on a coordinate plane as follows:



This gives us the following code:

```
main =
```

```
collage 500 500
  [ square 200
      |> filled red
      |> move (0,-150)
  ]
```

Take 5 minutes to try drawing a triangular roof for the house. Recall that we can use the `polygon` function, which takes a list of points as input.

Solution:

```
  , polygon [(-100,0), (100,0), (0,100)]
      |> filled black
      |> move (0,-50)
```

Now suppose that we wanted to add a window to our house. We can add a small white square with two thin black rectangles to create window panes:

```
  , square 50
      |> filled white
      |> move (-50,-100)
  , rect 50 5
      |> filled black
      |> move (-50,-100)
  , rect 5 50
      |> filled black
      |> move (-50,-100)
```

If we add these shapes to our collage and press "compile," we'll see a house with a window. (Code available at: `learn-elm.com/examples/house-1`.)

Now suppose we want to add another a window on the right side of the house. We can copy and paste all of the code we wrote for the first window, just changing the location. For example, we could replace (-50, -100) with (50, -100).

This works perfectly well, but suppose we want yet another window. We would have to copy and paste our code again and change the location of the three new shapes. This would leave us with three sets of three shapes. The repetition would make our program unnecessarily long and difficult to read. Furthermore, if we decided to make any changes to the windows, we'd have to make changes to each of the three different code segments. With each new window, it will become increasingly difficult to make changes or find mistakes.

Luckily, we have a solution to this problem. In the same way that we can *use* functions like `circle` and `square`, we can *make* our own functions.

Let's define a function called `window`, which creates a window using a square and two rectangles, as before. Below `main`, we can write:

```
window = group
  [ square 50
      |> filled white
      |> move (-50,-100)
  , rect 50 5
      |> filled black
      |> move (-50,-100)
  , rect 5 50
      |> filled black
      |> move (-50,-100)
  ]
```

`group` is a new function that simply takes in a list of shapes and groups them together into a single shape. We have just defined a function, called `window`, that creates a window. Now, any time the computer encounters the word `window`, it will know to create a window using the new function we defined. We can now change the `main` function to the following:

```
main =
  collage 500 500
    [ square 200
        |> filled red
        |> move (0,-150)
    , polygon [(-100, 0), (100, 0), (0, 100)]
        |> filled black
        |> move (0,-50)
    , window
    ]
```

[0:30 - 0:35]

Our new function certainly makes our code easier to read and understand, but it still only creates a window in one particular location, in this case centered at (-50, 100). We'd like to be able to specify a new location for each window we create.

In order to do this, we can define our function so that it takes a single argument: the

position of the window in the form (x, y). This is a pair of numbers that represents the x-coordinate and y-coordinate, respectively. We simply add `(x,y)` after the name of the function and before the equal sign. Everywhere we see `(-50,-100)` in our original function, we can instead use `(x,y)`. The computer will replace all instances of `(x,y)` with whatever location is given to the function as an argument.

```
window (x,y) = group
  [ square 50
      |> filled white
      |> move (x,y),
  , rect 50 5
      |> filled black
      |> move (x,y),
  , rect 5 50
      |> filled black
      |> move (x,y)
  ]
```

If we tried to compile our code now, we'd get a bunch of errors. This is because we used the `window` function in the `main` function without giving it any arguments. We need to change `window` to `window (-50,-100)`. This will tell the computer to draw a window at (-50, 100), using the function we defined. Now if we want to draw a second window at (50, -100), we could just add `window (50,-100)` to our list of shapes, as follows:

```
main =
  collage 500 500
    [ square 200
        |> filled red
        |> move (0,-150)
    , polygon [(-100,0), (100,0), (0,100)]
        |> filled black
        |> move (0,-50)
    , window (-50,-100)
    , window (50,-100)
    ]
```

If we press "compile" now, we'll see that the computer now draws two windows, one at each of the specified locations. We could add as many windows as we want without repeating much code at all. Our code is also easier to understand because the name of the window function makes it obvious what it does.

(Code available at: `learn-elm.com/examples/house-2`.)

[0:35 - 1:00]

For the rest of the class, you will have the opportunity to continue working on your house.

Here are a few goals you should try to accomplish:

1. Add a sky and grass behind the house.
2. Define a function that creates a door, and add the door to your house.
3. Define a function that creates a bush, and add several bushes around your house.

Once you finish these tasks, you should spend the rest of your time customizing your house in any way you like! Be sure to create new functions whenever appropriate.

For reference, here's a list of functions that we can use to create shapes (argument descriptions are written in caps):

- `rect WIDTH HEIGHT`
- `oval WIDTH HEIGHT`
- `square EDGE_LENGTH`
- `circle RADIUS`
- `ngon NUMBER_OF_SIDES RADIUS`
- `polygon LIST_OF_VERTICES`

Finished code:

(Available at: `learn-elm.com/examples/finished-house`.)

```
import Color exposing(..)
import Graphics.Collage exposing(..)

main =
  collage 500 500
    [ square 500
        |> filled blue
        |> move (0,0)
    , rect 500 50
        |> filled green
        |> move (0,-225)
    , square 200
        |> filled red
        |> move (0,-150)
    , polygon [(-100,0), (100,0), (0,100)]
        |> filled black
        |> move (0,-50)
    , window (-50,-100)
    , window (50,-100)
    , door
    , bush (-50,-225)
    , bush (-80,-225)
    , bush (50,-225)
    , bush (80,-225)
    ]
```

```
window (x,y) = group
  [ square 50
      |> filled white
      |> move (x,y)
  , rect 50 5
      |> filled black
      |> move (x,y)
  , rect 5 50
      |> filled black
      |> move (x,y)
  ]

door = group
  [ rect 50 80
      |> filled black
      |> move (0,-210)
  , circle 5
      |> filled white
      |> move (15,-215)
  ]

bush (x,y) = group
  [ circle 25
      |> filled darkGreen
      |> move (x,y)
  ]
```

A.3 Functions and Variables

Objective: This lesson will teach students how to use functions for computation. It will also introduce students to the concept and use of variables.

Key Words and Concepts: variable, return, evaluate

Assessments:

- Write Elm functions to
  - draw a simple picture (e.g. cloud, balloon).
  - draw the minute hand on a clock.
  - perform simple mathematical computations (e.g. addition, multiplication).
  - calculate the roots of a second-order polynomial using the quadratic equation.

Timeline:

[0:00 - 0:25]

Last week, we learned how to use functions to make drawing easier. Today we'll see that functions can also be used for computation.

Let's return briefly to our house drawings. Suppose we want to add a tree. We can define a function called `tree` that takes the position as an argument. We'll start by drawing the trunk of the tree as a brown rectangle at the given position:

```
tree (x,y) = group
  [ rect 20 150
      |> filled brown
      |> move (x,y)
  ]
```

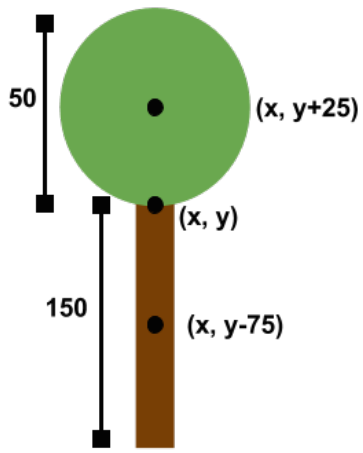
Now suppose we want to add a green circle for the leaves of the tree. We could update our function as follows:

```
tree (x,y) = group
  [ rect 20 150
      |> filled brown
      |> move (x,y)
  , circle 50
      |> filled green
      |> move (x,y)
  ]
```

If we add `tree (-180,-160)` to our list of shapes and press "compile," we'll see that the circle ends up in the center of the rectangle. In the first lesson, we discussed how we could increase the y-coordinate of a shape to move it upwards. If we try changing `-160` to a smaller number

like `-100`, we'll see that the whole shape moves up, but the circle is still in the middle of the rectangle.

Instead, we need to give the circle and rectangle different y-coordinates, each relative to the given location. We could, for example, put the given location `(x,y)` between the leaves and the trunk. Let's sketch out our tree to determine the coordinates we want to give our circle and rectangle.

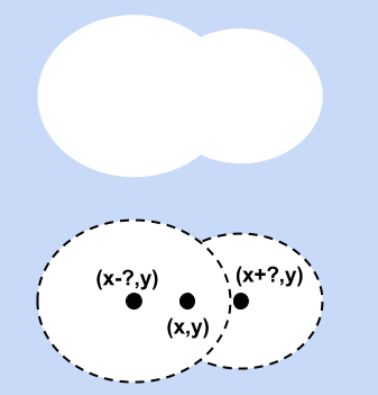


It's clear from our drawing that we want the y-coordinate of the rectangle to be `y - 75` and the y-coordinate of the circle to be `y + 25`. So we can update our code as follows:

```
tree (x,y) = group
  [ rect 20 150
      |> filled brown
      |> move (x,y-75)
  , circle 50
      |> filled green
      |> move (x,y+25)
  ]
```

If we compile our code again, we'll see that the circle now sits on top of the rectangle, as expected.

Take 15 minutes to create a function that draws a cloud using several ovals. For example, your cloud might look like this:



[0:25 - 0:40]

Let's leave our houses behind for now, and look at a new program:

```
import Color
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)

main =
  collage 500 500
    [ image 400 400 "http://learn-elm.com/assets/clock.png"
        |> toForm
        |> move (0,0)
    ]
```

(Code available at: `learn-elm.com/examples/clock-1`.)

There are two new functions here: `image` and `toForm`. `image` is a function that allows us to use an existing image in our drawings. It takes three arguments: a width, a height, and an URL in quotation marks. `image 400 400 "http://learn-elm.com/assets/clock.png"` tells the computer to create a 400 by 400 unit version of the image found at `http://learn-elm.com/assets/clock.png`.

We'll learn more about why we need to use `toForm` next lesson, but for now, know that it allows us to draw the image in the same way that we can draw shapes.

If we compile this code, we'll see that it draws the face of a clock. Let's try to add an hour hand so that the clock shows 12 o'clock. We'll need a few new functions. `segment` is a function that takes two points and creates a line segment connecting them. `traced` takes a line-style and a segment and draws the line segment with the given style. We can put these together to get a piece of code like the following:

```
segment (0,0) (0,100)
  |> traced (solid black)
```

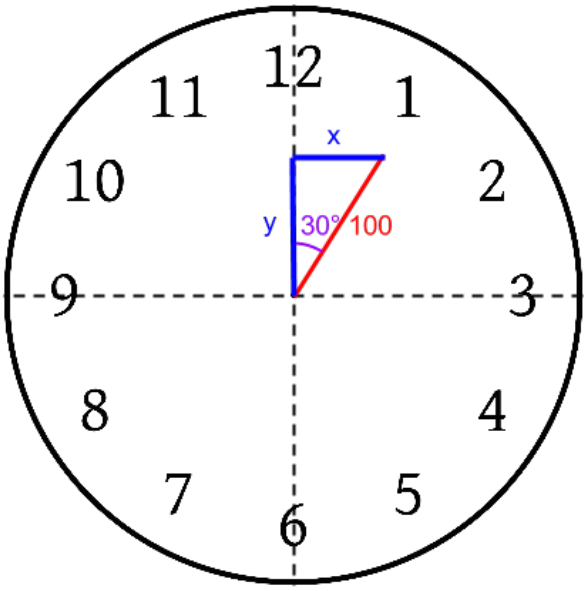
This piece of code draws a solid, black line-segment from (0, 0) to (0, 100). If we add this to our list of shapes and press "compile," we'll see that an hour hand is drawn on our clock!

Suppose we now want to have our hour hand point to the 1 on clock instead of the 12. The line-segment will still start at the center of the clock, (0,0), but we have to figure out where the line-segment should end. We could try to guess points until we found one that looks good, but it would take a long time if we wanted to do this for all of the numbers between 1 and 11. It would be nice if we could come up with a more general formula that will work for any number.

If your students have not yet encountered trigonometry, you can skip straight to the formulas rather than working through their derivation. Replace `angle = hour * 30` *with* `angle = hour * (360 / 12)`*, and explain that 360 is the number of degrees in the circle and 12 is the number of hours on the clock. Also note that 100 is the length of the hour hand.*

Let's go over a little math that will help us come up with this formula. Let's make the line 100 units long; that was the length of the line that we drew to point to 12. Our hour hand shouldn't change length it moves.

There are 360 degrees in a circle. Since the numbers on the clock break the circle into 12 pieces, then we can divide 360 by 12 to get that there are 30 degrees between each number on the clock. Since our goal is to figure out the x and y coordinates of the end of the hour hand, we can label our drawing as follows:



We can use some basic trigonometry to find that $sin(30°) = x/100$ and $cos(30°) = y/100$. The formulas for x and y are therefore $x = 100 \times sin(30°)$ and $y = 100 \times cos(30°)$.

We said that there are 30° between subsequent numbers on the clock. If we want to calculate the appropriate angle for a different hour, we can just multiply the hour by 30°.

Our final formulas are as follows:

$angle = hour \times 30$

$x = 100 \times sin(angle)$
$y = 100 \times cos(angle)$

[0:40 - 0:50]

Let's use these formulas to write a function that will calculate the end position of the clock's hour hand given the hour. As we learned last week, we can define a function by writing the name of the function, followed any arguments that it takes. If we call our function `hourHandEnd` and the argument `hour`, then our function begins

```
hourHandEnd hour =
```

Now we want to calculate angle, x, and y so that we can use x and y to draw our line segment. We can calculate and store these values in **variables**. A variable is simply a symbol that represents a value. In this case, we want 3 variables: `angle`, `x`, and `y`.

In order to tell the computer that we're creating a variable, we use the special words `let` and `in`. To create a variable called `angle` we would write:

```
let angle = hour * 30 in
```

This tells the computer that in the code that follows, replace the word `angle` with the value of `hour * 30`. Similarly, we can create variables for `x` and `y`:

```
let angle = hour * 30 in
let x = 100 * sin (degrees angle) in
let y = 100 * cos (degrees angle) in
```

Notice that these lines of code look almost identical to the formulas we defined earlier. Also notice that we have to include the word `degrees` to tell the computer that angle is measured in degrees. Now that we have calculated the values of `x` and `y`, we can just add `(x,y)` to the end of the function:

```
hourHandEnd hour =
  let angle = hour * 30 in
  let x = 100 * sin (degrees angle) in
  let y = 100 * cos (degrees angle) in
    (x,y)
```

Now, if we use the function `hourHandEnd`, it will give us back the position it calculated, `(x,y)`. In computer science, we say that the function **returns** or **evaluates to** the position, `(x,y)`.

We can now use this function within the code that draws our line segment. For example, if we want to draw the hour hand so that it points to 2, we would write:

```
segment (0,0) (hourHandEnd 2)
    |> traced (solid black)
```

(Code available at: `learn-elm.com/examples/clock-2`.)

Let's go over what the computer is doing when it sees `hourHandEnd 2`. First, it finds the function called `hourHandEnd`. It then executes the code within the function, using the value 2 whenever the variable `hour` is encountered. So when it executes the code

```
let angle = hour * 30 in
```

it computes 2 * 30, using the value 2 for the variable `hour`. The computer will then use the value 60 (since 2 * 30 = 60) whenever the variable `angle` is encountered. Similarly, when the computer executes the code

```
let x = 100 * sin (degrees angle) in
```

it computes $100 * sin(60°)$, using the value 60 for the variable `angle`. The computer will then use the value 86.6 (since $100 * sin(60°) = 86.6$) whenever the variable `x` is encountered. Finally, when the computer executes the code

```
let y = 100 * cos (degrees angle) in
```

it computes $100 * cos(60°)$, using the value 60 for the variable `angle`. The computer will then use the value 50 (since $100 * cos(60°) = 50$) whenever the variable `y` is encountered.

As a result, the last line of the function, `(x,y)`, evaluates to the pair (86.6, 50). This is then used as the end position of the line segment. When we press "compile," we'll see that we drew the hour hand as expected!

[0:50 - 0:60]

On your own, define another function called `minuteHandEnd` that takes the current minute (between 0 and 59) as input and calculates the end position of the minute hand. Remember that there are 360 degrees in a circle and 60 minutes total. Make your minute hand longer than your hour hand and a color other than black.

If you have time left over, try writing an additional function, `hands`, that takes two arguments, `hour` and `minute`, and creates both hands. (In other words, adding `hands 12 15` to your list of shapes should draw the clock's hands to show 12:15.)

Finished code:

(Available at: `learn-elm.com/examples/finished-clock`.)

```
import Color exposing (..)
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)

main =
  collage 500 500
    [ image 400 400 "http://learn-elm.com/assets/clock.png"
        |> toForm
        |> move (0,0)
```

```
    , hands 12 15
    ]

hands hour minute = group
  [ segment (0,0) (hourHandEnd hour)
      |> traced (solid black)
  , segment (0,0) (minuteHandEnd minute)
      |> traced (solid red)
  ]

hourHandEnd hour =
  let angle = hour * 30 in
  let x = 100 * sin (degrees angle) in
  let y = 100 * cos (degrees angle) in
    (x,y)

minuteHandEnd minute =
  let angle = minute * 6 in
  let x = 140 * sin (degrees angle) in
  let y = 140 * cos (degrees angle) in
    (x,y)
```

A.4 Types

Objective: This lesson will introduce students to Elm's type system. Students will learn about primitive types, function types, partial application, and type annotations.

Key Words and Concepts: type, functional language, partial application, type annotation

Assessments:

- Define or describe: "type" and "partial application."
- Identify the types of values and create examples of values with specified types.
- Write type annotations for Elm functions.
- Write a "hello world" program given the type annotations of new functions.

Timeline:

[0:00 - 0:10]

Let's start by looking at a simple program:

```
import Color exposing(..)
import Graphics.Collage exposing(..)

main =
  collage 500 500
    [ circle 100
        |> filled red
        |> move (0,0)
    ]
```

(Code available at `learn-elm.com/examples/types`.)

We've learned that this code creates a 500 by 500 area and draws a red circle with radius of 100 at (0, 0). We can press "compile" to confirm this.

What would happen if we changed `red` to `10`? As humans, we recognize that it makes sense to color a shape red, but it doesn't make sense to color a shape 10. Will the computer recognize this? We can press "compile" to find out.

We receive the following error message:

```
TYPE MISMATCH
The argument to function 'filled' is causing a mismatch.


7|  |> filled 10
```

```
Function ‘filled’ is expecting the argument to be:

    Color

But it is:

    number
```

Seems like the computer recognizes the mistake too. It prints the line with the error and displays the `10` in red to show us that there is something wrong with using `10` as input to the `filled` function. It goes on to tell us that there is a “conflict” between `Color` and `number`. This shouldn’t be too surprising since we knew we were supposed to give `filled` a color (like `red`) and instead we gave it a number (`10`). If we look at the top of the error message, we’ll see that the computer calls this a “type mismatch.”

A **type** is a categorization of a value that determines the operations that can be performed with it [28]. The type of `red` is `Color`, and we’ve seen that we can use `Color`s to fill shapes. `blue`, `yellow`, `green`, etc. also have the type `Color`. The type of `10` is `number`, and we’ve seen that we can use `number`s to specify the sizes of shape. Other values of type number are `-1`, `15`, `0.5`, etc.

`number` is special in that it is a type that includes two more specific types: `Int` and `Float`. `Int` stands for integer and is the type of whole numbers. `Float` is the type for numbers with decimals. `-1` and `15` are `Int`s while `0.5` is a `Float`.

What other types have we seen? Recall that the argument to the function `move` is a *pair* of numbers. The type of `(0,0)` is `(number, number)`. The type of a pair is made of up of the types of each of the values in the pair.

In the same way that we can have a pair of type `(number, number)`, we can have a pair of type `(Color, Color)`. For example `(red, red)` and `(red, black)` both have type `(Color, Color)`. We can also have a pair that contains values with two different types. For example, `(50, black)` has type `(number, Color)`. Note that `(black, number)` has type `(Color, number)`. In a pair, order matters!

Pairs can contain values of any type. Pairs can even contain pairs! For example, the type of `((1, 2), (3, 4))` is `((Int, Int), (Int, Int))`.

What is an example of an expression with type

`((Color, Color), (Int, Int))`?

Possible solution: ***`((red, black), (10, 20))`***

`(Float, (Int, Color))`?

Possible solution: ***`(0.5, (10, red))`***

What is the type of the expression

`((1.5, 0.5), red)`?

Solution: *`((Float, Float), Color)`*

`(red, (blue, (yellow, black)))`?

Solution: *`(Color, (Color, (Color, Color))`*

[0:10 - 0:20]

Elm is a **functional language**. This means that functions are treated as values. Since all functions are values, this means that functions must also have types. Similar to pairs, the types of functions are made up of other types. Consider the following function:

```
double x = 2 * x
```

This is a function that takes a number as input and evaluates to a number. The type of `double` is therefore `number -> number`.

Consider another function:

```
add_pair (x,y) = x + y
```

This is a function that takes a pair of numbers as input and evaluates to a number. The type of `add_pair` is therefore `(number, number) -> number`.

Consider another similar function:

```
add x y = x + y
```

This is a function that takes two numbers as input and evaluates to a single number. The type of `add` is therefore `number -> number -> number`.

Consider one more similar function:

```
mult_pair (x,y) z = (x*z, y*z)
```

This is a function that takes a pair of numbers and a number as input and returns a pair of numbers. The type of `mult_pair` is therefore `(number, number) -> number -> (number, number)`. Like pairs, notice that the order matters.

What is the type of

`add_pair z (x,y) = (x+z, y+z)`?

Solution: *`number -> (number, number) -> (number, number)`*

`add_pairs (x,y) (z,w) = (x+z, y+w)`?

Solution: `(number, number) -> (number, number) -> (number, number)`

What is an example of a function that has the type

`(number, number) -> (number, number)`?

Possible solution: `switch (x,y) = (y,x)`

`number -> (number, number)`?

Possible solution: `twice x = (x,x)`

[0:20 - 0:30]

Let's return to the function add:

```
add x y = x + y
```

As we saw earlier, the type of `add` is `number -> number -> number`.

What is the type of `add 1 2`? We know that `add 1 2` would return `3`, which is a number. So the type of `add 1 2` is `number`. What is the type of `add 1`? This is essentially a function that takes a number and returns the sum of 1 and the input. The type is therefore `number -> number`. If we give a function some, but not all, of its arguments, it is called **partial application**.

We can use our `add` function to define a function `inc` (short for increment) that takes one argument and adds 1 to it.

```
inc x = add 1 x
```

Now if we want to add 1 and 2, we can use `add 1 2` or `inc 2` since `inc` is equivalent to `add 1`.

Suppose we had a function called `sub`, defined as follows:

```
sub x y = x - y
```

If we wanted to write a function `dec` that decrements (subtracts 1) a number, we could **not** write:

```
dec x = sub 1 x
```

This will subtract the argument from 1 instead of subtracting 1 from the argument. More generally, remember that the order of the arguments matters. The correct function would be:

```
dec x = sub x 1
```

[0:30 - 0:40]

We can include the types of values in our programs using **type annotations**. For example:

```
add : number -> number -> number
add x y = x + y
```

Type annotations begin with the name of the value and a colon, followed by the type of the value. Type annotations are especially useful for functions, since they help to explain how to use the function. In particular, they give the types of the arguments and the type to which the function evaluates. Type annotations also allow the computer to verify that we are using values appropriately. For example, if we tried to compile

```
add : Color -> number -> number
add x y = x + y
```

we would get another "type mismatch" between `Color` and `number` because the computer realizes we can't add a value of type `Color` and a value of type `number` and expect to get a value of type `number`.

Let's come up with type annotations for a few more functions. Consider the program we saw at the beginning of the lesson:

```
main =
  collage 500 500
    [ circle 100
        |> filled red
        |> move (0,0)
    ]
```

This program involves a number of types we haven't yet named. An `Element` is a graphical element that can be displayed in the browser. A `Shape` is simply a shape, like a rectangle, circle, etc. A `Form` is a graphical element that can be manipulated or changed. For example, a `Form` can be moved, rotated, and scaled.

Here are the type annotations for the functions in this program:

```
main : Element
collage: Int -> Int -> List Form -> Element
circle : Float -> Shape
filled : Color -> Shape -> Form
move : (Float, Float) -> Form -> Form
```

These type annotations fit with our understanding of these functions, but it would have been difficult to figure them out ourselves. Luckily, the people who wrote these functions wrote a type annotation for each function, which we can reference. Once you become comfortable with types, you can look up any function and very quickly get an idea of how to use it!

[0:40 - 1:00]

Last week, we saw a new type; let's give it a name. One of the inputs to the `image` function is a URL contained in quotation marks. Any series of characters contained in quotation marks

is called a `String`. For example, `"Hello, world"` is a `String`, as is `"Elm"` and `"computer science"`.

Note that strings need quotation marks. `hello`, for example, is not a string. Why not? *We need to be able to distinguish between variable names and strings.*

Below are some new functions. Using the types to guide you, use these functions to create a program that displays a string — perhaps "Hello, world!" or your name. Experiment with different styles, and don't forget to add type annotations to your functions!

```
fromString : String -> Text
Convert a string into text which can be styled and displayed.

height : Float -> Text -> Text
Set the height of some text.

color : Color -> Text -> Text
Set the color of some text.

bold : Text -> Text
Make text bold.

italic : Text -> Text
Make text italic.

text : Text -> Form
Create some text.
```

A.5 Lists and Map

Objective: This lesson will provide students with a more complete understanding of lists, including their purpose and their properties. It will also introduce the `map` function, which transforms elements in a list.

Key Words and Concepts: list, element, map

Assessments:

- Define or describe: "list."
- Write a program that uses `List.map` to transform a list of integers.
- Write a program that uses `List.map`, `List.map2`, or `List.map3` to draw a
  - flower.
  - colored target.
  - grayscale.

Timeline:

[0:00 - 0:05]

Let's start by looking at a simple program that draws a flower:

```
import Color exposing(..)
import Graphics.Collage exposing(..)

main =
  collage 500 500
    flower

petal x y deg =
  oval 100 50
    |> filled yellow
    |> rotate (degrees deg)
    |> move (x,y)

flower =
  [ petal -60 0 0
  , petal 60  0 0
  , petal 0 -60 90
  , petal 0 60 90
  , petal -45 -45 45
  , petal -45 45 -45
  , petal 45 -45 -45
  , petal 45 45 45
  , circle 35
      |> filled black
```

```
      |> move (0,0)
  ]
```

(Code available at `learn-elm.com/examples/flower-1`.)

As a review of types, let's take a few minutes to add type annotations to each function.

Solution:

```
main : Element
petal : Float -> Float -> Float -> Form
flower : List Form
```

Let's focus on the type of `flower`. We've seen lists before (as argument to `collage` and `polygon`), but we have not yet discussed them in depth. **Lists** are ordered collections of things. We call each thing in a list an **element**. In Elm, lists are required to contain elements of the same type. This allows us to give types to lists. In the case of `flower`, we have a list of `Form`s.

[0:05 - 0:10]

So far, we've only seen lists of `Form`s, but we can have a list of elements of any type. For example, consider a list of `Int`s:

```
[1,2,3,4,5,6,7,8,9,10]
```

Notice that just like lists of `Form`s, lists of `Int`s are contained in square brackets and the elements are separated by commas. If we want all integers in some range, we can also use the following shorthand:

```
[1..10]
```

If we want to be able to see the list displayed on the right side of the screen, we can use a function called `show`.

```
show : a -> Element
Convert anything to its textual representation and make it displayable in the browser.
```

The following program will display a list of integers between 1 and 10, inclusive:

```
import Graphics.Element exposing (..)

main : Element
main = show [1..10]
```

Take a few minutes to write a program that displays the even numbers between 2 and 20, inclusive.

Solution:

```
import Graphics.Element exposing (..)

main : Element
main = show [2,4,6,8,10,12,14,16,18,20]
```

[0:10 - 0:20]

What if instead of creating a list of even numbers between 2 and 20, we wanted to create a list of even numbers between 2 and 200? We could take the time to write each of the 100 numbers out, but this would be tedious and error-prone.

We know that it's easy to make a list of integers between 1 and 100, inclusive: `[1..100]` If we could simply multiply each of these integers by 2, then we'd have our list of even numbers between 2 and 200. Let's write a function that multiplies a given integer by 2:

```
mult2 : Int -> Int
mult2 x = 2 * x
```

Since we want to apply the same function to each element in a list, we can use a special function called `map`. `List.map` takes a function and a list and returns the result of applying the function to each element in the list. (The result will also be a list.)

So if we write `List.map mult2 [1..100]`, it will apply `mult2` to each integer in the list. We can now define `main` as follows:

```
main : Element
main =
  show (List.map mult2 [1..100])
```

(Code available at: `learn-elm.com/examples/map-mult2`.)

If we make these changes and press "compile," we'll see the even integers between 2 and 200, inclusive.

Take a few minutes to write a program that uses `map` to display all multiples of 3 between 0 and 150, inclusive.

[0:20 - 0:25]

Let's return to our flower example. Suppose we want to make our flower twice its current size. One option is to go through the list of forms and change the arguments to each of the shape functions. But with so many forms (each with one or two arguments), that would be tedious and error-prone. Another option is to use the `scale` function:

```
scale : Float -> Form -> Form
Scale a form by a given factor.
```

Since `scale` takes a float and a form, we could apply `scale 2` to each of the forms in the list. To simplify this even further, we could define a function that takes a form as input and returns the same form, scaled by a factor of 2:

```
double : Form -> Form
double f = scale 2 f
```

Since we want to apply `double` to each form in the list, we can use `List.map` again:

```
main : Element
main =
  collage 500 500
    (List.map double flower)
```

(Code available at `learn-elm.com/examples/flower-2`.)

If we press "compile," we'll see that flower doubles in size, as desired.

[0:25 - 0:30]

Let's consider the type of `List.map`:

```
map : (a -> b) -> List a -> List b
Apply a function to every element of a list.
```

Notice that `a` and `b` are not types; they are just *placeholders* for types. For example, we used `map` with `double`, which has type `Form -> Form`. In this case, both `a` and `b` are `Form`. This tells us that the second argument to `map` must be a list of forms (`List Form`) and the output of `map` will be a list of forms (`List Form`).

We can also use `map` with types other than `Form`. For example, suppose we wanted to divide each integer in a list by 2. We would begin by writing a function that takes an integer and returns half its value. Since the result might not be an integer, this will be a function from integers to floats. In order to treat the input as a float, we can use the function `toFloat`.

```
toFloat : Int -> Float
Convert an integer into a float.
```

We can define our functions as follows:

```
half : Int -> Float
half i = (toFloat i) / 2
```

If we use `half` as the first argument to `map`, then we know the second argument should be a list of integers (`List Int`), and the output will be a list of floats (`List Float`). In other words, we replace `a` in the type of `map` with `Int` and we replace `b` with `Float`.

Now we can use `map` to apply this function to the integers between 1 and 10, inclusive:

```
List.map half [1..10]
```

Our entire program would be:

```
import Graphics.Element exposing (..)

main : Element
main =
  show (List.map half [1..10])

half : Int -> Float
half i = (toFloat i) / 2
```

[0:30 - 0:40]

Let's see how we can use `map` to transform a list of `Int`s into a list of `Form`s. Suppose we wanted to draw a row of 10 circles across the screen. If our canvas is 500 by 500 units, each circle will have a diameter of $500/10 = 50$ units and a radius of $50/2 = 25$ units. We'll start with a list of numbers between 1 and 10, inclusive. We want to write a function that takes an integer and creates a form. We might start with the following code:

```
import Color exposing (..)
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)

main : Element
main = collage 500 500
  (List.map draw_circle [1..10])

draw_circle : Int -> Form
draw_circle i =
  circle 25
    |> filled red
    |> move(0,0)
```

(Code available at `learn-elm.com/examples/circles-1`.)

This program draws 10 circles, each with a radius of 25, but it draws them all on top of one another. If we want to spread them out over the screen, we'll have to take advantage of our input. We can start by changing the x-coordinates of the circles to be dependent on `i`:

```
draw_circle : Int -> Form
draw_circle i =
```

```
  circle 25
    |> filled red
    |> move(toFloat i,0)
```

We need to use the function `toFloat` so that the given integer is treated as a float (the type that `move` expects). If we press "compile," we'll see that the circles have spread out very slightly, but not across the screen as we had hoped. We want the distance between the centers of two adjacent circles to be 50 units. So we could update our code as follows:

```
draw_circle : Int -> Form
draw_circle i =
  circle 25
    |> filled red
    |> move(toFloat (i*50),0)
```

Now the circles are spread out across the screen as we expect, but they start at the center of the screen instead of at the left-hand side. Our last change will be to shift the entire row of circles to the left by decreasing all of the x-coordinates:

```
draw_circle : Int -> Form
draw_circle i =
  circle 25
    |> filled red
    |> move(toFloat (i*50-275),0)
```

If we press "compile," we'll see that we successfully drew 10 circles in a row across the screen!

(Code available at `learn-elm.com/examples/circles-2`.)

[0:40 - 0:45]

Suppose we have two lists of integers, and we want to add corresponding elements in each list. We can easily write a function that adds two integers:

```
add : Int -> Int -> Int
add x y = x + y
```

We want to apply this function to the elements of two lists. Here, we can use the `map2` function. This takes two lists and transforms them into a single list, using the given function.

```
map2 : (a -> b -> result) -> List a -> List b -> List result
Combine two lists, combining them with the given function. If one
   list is longer, the extra elements are dropped.
```

We can use `map2` as follows:

```
import Graphics.Element exposing (..)

main : Element
```

```
main =
  show (List.map2 add [1..10] [11..20])

add : Int -> Int -> Int
add x y = x + y
```

This program will go through both lists, applying the `add` function to corresponding elements. The result will be a single list, containing the sums.

[0:45 - 1:00]

In the time remaining, use map to transform the lists: `[6,5,4,3,2,1]` and `[red,orange, yellow,green,blue,purple]` into the following picture:

What other drawings can you create using `List.map`?

Finished code:

(Available at `learn-elm.com/examples/rainbow-target`.)

```
import Color exposing (..)
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)

main : Element
main = collage 500 500
  (List.map2 draw [6,5,4,3,2,1]
    [red,orange,yellow,green,blue,purple])

draw : Float -> Color -> Form
draw r c =
  circle (r * 40)
    |> filled c
    |> move (0,0)
```

A.6 Conditionals

Objective: This lesson will introduce students to conditionals in Elm. They will have the opportunity to use conditionals in both graphical and non-graphical programs.

Key Words and Concepts: boolean, conditional

Assessments:

- Define or describe: "boolean" and "conditional."
- Construct conditionals English.
- Implement FizzBuzz [13].
- Write a function that determines
  - the quadrant of a point.
  - if two points are in the same quadrant.
- Draw a checkerboard.

Timeline:

[0:00 - 0:10]

Suppose we want to check if an integer is even. We know that even numbers are divisible by two. In other words, when we divide an even integer by 2, the remainder is 0. If we want to calculate the remainder when `a` is divided by `b` in Elm, we can write `a % b`. We read this as "a mod b."

So if `n % 2` is equal to `0`, then `n` is even. We can test if two numbers are equal using two equal signs: `==`. (Recall that we use one equal sign for assignment.) Using this information, let's write a function that determines if a given integer is even.

```
isEven n = n % 2 == 0
```

When given an argument, this function will evaluate to either true or false. A value that can either be true or false is called a **boolean**. In Elm, the type of a boolean value is `Bool`. So the type of `isEven` is `Int -> Bool`.

Suppose that we want to write a new function that displays "even" if a given integer is even and "odd" otherwise. In order to accomplish this, we'll need to use a **conditional**. A conditional in Elm is something of the form:

```
if boolean
  then expression1
else expression2
```

An example of a conditional in English is "If it is raining, then I will wear rain boots; otherwise, I will wear flip-flops," The boolean in this conditional is "it is raining." The first statement is "I will wear rainboots" and the second statement is "I will wear flip-flops."

A second example is "if it is a weekend, then I will sleep in; otherwise, I will wake up for school." What is the boolean? *It is a weekend.*

Take a few minutes to come up with some of your own conditionals in English.

Let's return to our original problem: if a given integer is even, we want to display the word even; otherwise, we want to display the word odd. The boolean is "a given integer is even." The first expression is "display the word even", and the second expression is "display the word odd". This translates nicely to Elm code:

```
if isEven n
  then show "even"
else show "odd"
```

Suppose we now want to write a function that displays "positive" if the argument is greater than 0, "negative" if the argument is less than 0, and "zero" if the argument is 0. Whereas our previous problem had two cases (even and odd), notice that this problem has three cases (positive, negative, and zero).

We can start with a standard conditional. If our argument is `n`, we might start with

```
if n > 0
  then show "positive"
else ...
```

Now we've covered the case where n is greater than 0. If `n` is *not* greater than 0, it is either less than 0 or equal to 0. As a result, we want the second statement to be another conditional that checks if `n` is less than 0.

```
if n > 0
  then show "positive"
else if n < 0
  then show "negative"
else ...
```

Now we've covered the cases where `n` is greater than 0 and and where `n` is less than 0. Since only one case remains (`n` is equal to 0), we can just use the else as usual:

```
if n > 0
  then show "positive"
else if n < 0
  then show "negative"
else show "zero"
```

[0:10 - 0:20]

A popular interview question for software engineering candidates is called FizzBuzz. The instructions are as follows:

- Print the numbers 1 through 100, but...

- For multiples of 3, print "Fizz" instead of the number
- For multiples of 5, print "Buzz" instead of the number
- For multiples of 3 and 5, print "FizzBuzz" instead of the number

If you want to check if two booleans are both true, you can use "logical and:" `&&`. The boolean `b1 && b2` is only true if both `b1` *and* `b2` are true.

Take ten minutes to try implementing FizzBuzz in Elm! (Hint: the function `toString` takes an integer as an argument and returns its string representation.)

If you have time leftover, create your own variation of FizzBuzz and implement it (or swap with a partner and have him or her implement it).

[0:20 - 0:25]

Review the solution, available at ***learn-elm.com/examples/fizz-buzz.***

```
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)

main : Element
main = show (List.map fizzBuzz [1..100])

fizzBuzz : Int -> String
fizzBuzz n =
  if n % 3 == 0 && n % 5 == 0
    then "FizzBuzz"
  else if n % 3 == 0
    then "Fizz"
  else if n % 5 == 0
    then "Buzz"
  else toString n
```

[0:25 - 0:35]

Suppose we want to write a function that determines the quadrant of a Cartesian coordinate `(x,y)`.

Recall that a coordinate is in...

- quadrant I if x $>$ 0 and y $>$ 0
- quadrant II if x $<$ 0 and y $>$ 0
- quadrant III if x $<$ 0 and y $>$ 0
- quadrant IV if x $>$ 0 and y $>$ 0

If x or y is equal to 0, then the point is not in any quadrant.

Let's try to write this function using conditionals. A natural first step is "check if x or y is equal to 0." We know how to check if x is equal to zero (`x == 0`) and we know how to check

if y is equal to zero (`y == 0`). We want to combine these booleans using "or." In Elm, the symbol for or is `||`. The boolean `b1 || b2` is true if `b1` *or* `b2` (or both) are true.

So `x == 0 || y == 0` means `x` is equal to 0 or `y` is equal to zero. If `x == 0 || y == 0`, then the point is not in any quadrant. In this case, the function should evaluate to `"none"`.

Recall that the symbol for "and" is Elm is `&&`. So if we want to check if `x > 0` and `y > 0`, we would write `x > 0 && y > 0`. In this case, the function should evaluate to `"quadrant I"`.

Navigate to `learn-elm.com/examples/cartesian-1` and complete the program:

```
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)

main : Element
main = show (quadrant (-10, 10))

quadrant : (Int, Int) -> String
quadrant (x,y) = "none"
```

If you have time leftover, change your program so that it also determines which axis the point is on if not in one of the quadrants.

[0:35 - 0:40]

Review the solution, available at `learn-elm.com/examples/finished-cartesian`.

```
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)

main : Element
main = show (quadrant (-10, 10))

quadrant : (Int, Int) -> String
quadrant (x,y) =
  if x == 0 || y == 0
    then "none"
  else if x > 0 && y > 0
    then "quadrant I"
  else if x < 0 && y > 0
    then "quadrant II"
  else if x < 0 && y < 0
    then "quadrant III"
  else "quadrant IV"
```

[0:40 - 0:45]

Last week, we wrote the following code to draw a row of 10 red circles:

```
import Color exposing (..)
```

```
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)

main : Element
main = collage 500 500
  (List.map draw_circle [1..10])

draw_circle : Int -> Form
draw_circle i =
  circle 25
    |> filled red
    |> move (toFloat (i*50-275),0)
```

(Code available at `learn-elm.com/examples/circles-2`.)

How could we modify this code so that every other circle is purple instead of red? We could start by defining a variable that will store the color of the circle. If `i` is even, we can set the variable to purple, and otherwise, we can set the variable to red.

```
let c = if (i % 2 == 0) then purple else red in
```

Now we can replace `filled red` with `filled c`.

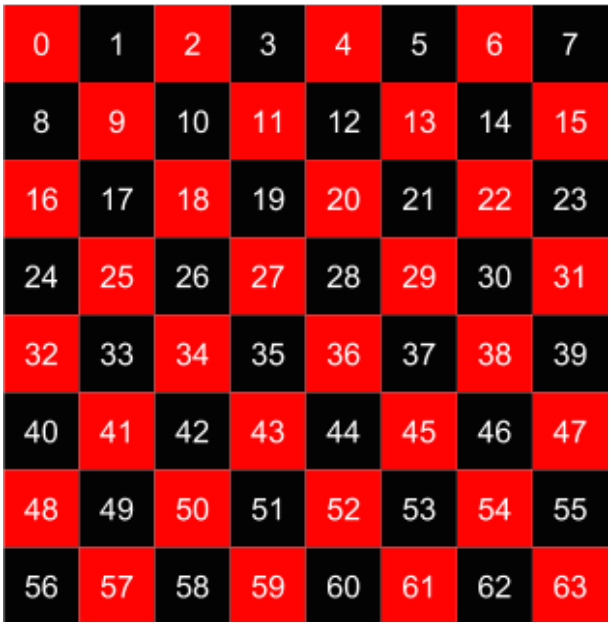
```
draw_circle : Int -> Form
draw_circle i =
  let c = if (i % 2 == 0) then purple else red in
circle 25
    |> filled c
    |> move (toFloat (i*50-275),0)
```

If we press "compile," we'll see that the colors alternate as desired!

[0:45 - 1:00]

Navigate to `learn-elm.com/examples/checkerboard`. Complete the program so that it draws a checkerboard.

Note that `//` performs integer division (`a//b` is number of times `a` "goes into" `b`). The following numbering might be helpful!



If you have time left-over, try making a more elaborate pattern.

Finished code:

(Available at `learn-elm.com/examples/finished-checkerboard`.)

```
import Color exposing (..)
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)

main : Element
main =
  collage 400 400 (List.map checkerBoard [0..63])

checkerBoard : Int -> Form
checkerBoard n =
  let row = n // 8 in
  let column = n % 8 in
  let c =
    if (row + column) % 2 == 0
      then black
    else red in
  square 50
    |> filled c
    |> move
        (-175 + (toFloat column) * 50, -175 + (toFloat row) * 50)
```

A.7 Mouse Signals

Objective: This lesson will introduce students to signals, focusing on mouse signals. Students will learn how to use signals to create interactive programs and will get further practice with map and conditionals.

Key Words and Concepts: signal

Assessments:

- Define or describe: "signal."
- Draw shapes whose color, size, and/or type change according to mouse signals.
- Implement "buried treasure" game using mouse signals and conditionals.

Timeline:

[0:00 - 0:15]

At the beginning of the semester, we looked at a few Elm programs, including a number of games. These games allow the user to interact with the program using the mouse or the keyboard. Today, we will learn how we can get input from the user so we can create our own games.

Let's start with the following program:

```
import Graphics.Element exposing (..)
import Mouse exposing (..)

main : Signal Element
main =
  Signal.map show Mouse.position
```

(Code available at `learn-elm.com/examples/mouse-position`.)

Before we discuss the code, let's see what the program does. There is a pair of integers in the top-left corner, and if we move our mouse around, the numbers change. This program displays the current position of the mouse.

Now that have a sense of what the program does, let's take a look at the code. `Mouse.position` is a function provided by Elm. It has type `Signal (Int, Int)`. A **signal** is a time-varying value (a value that changes over time). In this case, the time-varying value is a pair of integers that represent the current position of the mouse.

For example, if we started at the origin and moved the mouse to the right, the value of `Mouse.position` would start at `(0,0)` and change to `(1,0)`, then `(2,0)`, then `(3,0)`, and so on as we move the mouse across the x-axis.

`Signal.map show` applies the function `show` to the current mouse position. If we think of

a signal as an infinite list of values, we can see that `Signal.map` works like the `List.map` function we've worked with. While `List.map` works on a finite list, `Signal.map` works on "infinite lists." The type of `Signal.map` is as follows:

```
map : (a -> b) -> Signal a -> Signal b
Apply a function to a signal.
```

Each time a the value of `Mouse.position` changes, the function `show` is reapplied. As a result, `Signal.map show Mouse.position` always shows the current position of the mouse. Note that up until now, `main` has always had the type `Element`. Now, `main` is an `Signal Element`. This means that what is displayed (in this case the text in the upper-left corner of the screen) may change over time.

Let's define a function `draw` that will draw a circle given a pair of integers. We will use it to draw a circle at the current mouse position.

```
draw : (Int,Int) -> Element
draw (x,y) =
  collage 500 500 [
    circle 50
      |> filled red
      |> move (toFloat x, toFloat y)
  ]
```

Notice that we have to use `toFloat` to make `x` and `y` (integers) appropriate arguments to move (which expects a pair of floats).

If we replace `show` with `draw`, `Signal.map` will apply `draw` to the current mouse position so that the circle is drawn wherever the mouse is. If we press "compile," we'll see that the circle moves around as we move the mouse, but it doesn't follow the mouse as we had hoped. This is because the mouse position is measured from the top-left corner of the screen, with x increasing to the right and y decreasing down. This is in contrast to the collage's coordinate system, whose origin is at the center of the collage.

Let's transform the mouse position so that it matches the coordinate system used for collages. We know that placing the mouse in the top-left corner of the screen will give the position `(0,0)`. We want the top-left corner of the screen to be `(-250,250)`. (Recall that the center of a 500 by 500 collage is `(0,0)` with x ranging from -250 to 250 left-to-right and y ranging from -250 to 250 bottom-to-top.) We can change the code as follows:

```
draw : (Int,Int) -> Element
draw (x,y) =
  collage 500 500 [
    circle 50
      |> filled red
      |> move (toFloat x - 250, 250 - toFloat y)
  ]
```

If we press "compile" again, we'll see that the circle follows the mouse around, as desired.

(Code available at `learn-elm.com/examples/mouse-circle`.)

[0:15 - 0:20]

Let's pause our discussion of signals to improve the design of our program. Notice that in our `draw` function, `500` and `250` are used multiple times. What if we wanted to change the size of our collage? We'd have to change four numbers, which is cumbersome and error-prone.

In order to avoid these problems, let's define a variable `halfWidth` whose value will be half the width of our collage. Above `draw`, add the following:

```
halfWidth = 250
```

This defines a variable, called `halfWidth`, that will be available throughout the program. We can then make the following changes to `draw`:

```
draw : (Int,Int) -> Element
draw (x,y) =
  collage (halfWidth * 2) (halfWidth * 2) [
    circle 50
      |> filled red
      |> move (toFloat x - halfWidth, halfWidth - toFloat y)
  ]
```

Now we can change the size of our collage by changing the value of one, easy to identify variable.

[0:20 - 0:35]

Let's go ahead and add a second signal to our program: a signal of booleans that tells us whether or not the mouse is pressed. This signal is called `Mouse.isDown` and is provided by Elm.

Let's change our `main` function to

```
main : Signal Element
main =
  Signal.map2 draw Mouse.position Mouse.isDown
```

`Signal.map2` applies the a given function to two signals. In this case, it applies function `draw` to (1) a `(Int,Int) Signal` representing the position of the mouse and (2) a `Bool Signal` that is true when the mouse is clicked and false otherwise. The type of `draw` must then change to `(Int,Int) -> Bool -> Element`.

We can add a second argument to `draw` as follows:

```
draw (x,y) isClicked =
```

Take a few minutes to change `draw` so that the circle is blue if `isClicked` is true and red otherwise. If you have time left over, try changing the size depending on whether or not the mouse is clicked.

Go over solution, available at `learn-elm.com/examples/mouse-click-circle`:

```
import Color exposing (..)
import Graphics.Element exposing (..)
import Graphics.Collage exposing (..)
import Mouse exposing (..)

halfWidth = 250

main : Signal Element
main =
  Signal.map2 draw Mouse.position Mouse.isDown

draw : (Int,Int) -> Bool -> Element
draw (x,y) isClicked =
  let c = if isClicked then blue else red in
  collage (halfWidth * 2) (halfWidth * 2) [
    circle 50
      |> filled c
      |> move (toFloat x - halfWidth, halfWidth - toFloat y)
  ]
```

[0:35 - 1:00]

Now we'll create our first game. In our game, there will be some buried treasure at a secret location. The player is a small purple circle that can move around the screen using the mouse. The player must look for the secret location. If he or she finds it, the circle should turn green.

We'll start with the code that draws a purple circle at the mouse's position. We'll define variables `secretX` and `secretY` to hold the x and y coordinates of the secret location. We also have a function `distance` that determines the approximate distance between two points.

```
import Color exposing (..)
import Graphics.Element exposing (..)
import Graphics.Collage exposing (..)
import Mouse exposing (..)

secretX = 10
secretY = 100
halfWidth = 250

main : Signal Element
main = Signal.map draw Mouse.position
```

```
draw : (Int,Int) -> Element
draw (x,y) =
  let mouseX = toFloat x - halfWidth in
  let mouseY = halfWidth - toFloat y in
  let c = purple in
  collage (halfWidth * 2) (halfWidth * 2) [
    circle 10
      |> filled c
      |> move (mouseX,mouseY)
  ]

distance : (Float,Float) -> (Float,Float) -> Int
distance (x1,y1) (x2,y2) =
  round (sqrt ((x1 - x2)^2 + (y1 - y2)^2))
```

(Code available at `learn-elm.com/examples/treasure-1`.)

In order to make the circle turn green if the player has found the secret location, we first need to determine if the mouse is within 10 units (the radius of the circle) of the secret location. To accomplish this, we can use the following boolean:

```
distance (mouseX,mouseY) (secretX,secretY) <= 10
```

Begin by modifying the program so that the circle turns green if the mouse is within 10 units of the secret location.

When you're done, try playing this game yourself. You'll notice that it's pretty tricky. Make this game easier by adding a "hint" feature. When the user clicks the mouse, the circle should turn red if the mouse is more than 100 units away from the secret location, orange if it is between 100 and 50 units away, and yellow if it is between 50 and 10 units away.

When you're done, you can pair up with a partner and play his or her game! (In order to hide the values of `secretX` and `secretY`, simply drag the line separating the editor from the output to the left.)

Finished code:

(Available at `learn-elm.com/examples/finished-treasure`.)

```
import Color exposing(..)
import Graphics.Collage exposing(..)
import Graphics.Element exposing(..)
import Mouse exposing(..)

secretX = 10
secretY = 100
halfWidth = 250

main : Signal Element
```

```
main = Signal.map2 draw Mouse.position Mouse.isDown

draw : (Int,Int) -> Bool -> Element
draw (x,y) isClicked =
  let mouseX = toFloat x - halfWidth in
  let mouseY = halfWidth - toFloat y in
  let d = distance (mouseX,mouseY) (secretX,secretY) in
  let c =
    if d <= 10 then green
    else if isClicked && (d > 100) then red
    else if isClicked && (d <= 100) && (d > 50) then orange
    else if isClicked && (d <= 50) && (d > 10) then yellow
    else purple
  in
  collage (halfWidth * 2) (halfWidth * 2) [
    circle 10
      |> filled c
      |> move (mouseX,mouseY)
  ]

distance : (Float,Float) -> (Float,Float) -> Int
distance (x1,y1) (x2,y2) =
  round (sqrt ((x1 - x2)^2 + (y1 - y2)^2))
```

A.8 State and Pattern Matching

Objective: This lesson will introduce students to `Signal.foldp`, a function that enables the program to accumulate state. Students will also work with record types and pattern matching to simplify their increasingly complex programs.

Key Words and Concepts: `foldp`, record type, pattern matching

Assessments:

- Define or describe: "`foldp`" and "record type."
- Create a drawing program where the "paint"
  - is controlled by `Mouse.isDown`.
  - is erased by space key.
  - color is changed by letter keys.
- Write a program that draws a circle that grows each time the mouse is clicked and returns to its original size whenever the space key is pressed.

Timeline:

[0:00 - 0:10]

Last week, we used signals to draw a circle that moves with the mouse. Suppose we wanted to modify this program so that it draws a circle at every mouse location, leaving a "trail" of circles. This is not possible with the tools we have now because he have no way to accumulate or collect values. Right now, we can only assign and calculate values.

In order to write this program, we'll need a new function, `Signal.foldp`:

```
foldp : (a -> state -> state) -> state -> Signal a -> Signal state
```

Notice that `foldp` takes three arguments: a function, a state, and a signal. `foldp` allows us to "collect" the values from the given signal into a single state. The initial state is specified by the second argument. Each time the signal produces a new value, the function specified by the first argument is applied to the new value and the existing state to produce a new state. The new state becomes a new value in the output signal. This function is called `foldp` because it performs a "fold from the past" on the input signal's values [4].

Let's consider a concrete example that "collects" the number of mouse clicks:

```
import Graphics.Element exposing (..)
import Mouse exposing (..)

main : Signal Element
main =
  Signal.map show (Signal.foldp incrementTotal 0 Mouse.isDown)

incrementTotal : Bool -> Int -> Int
```

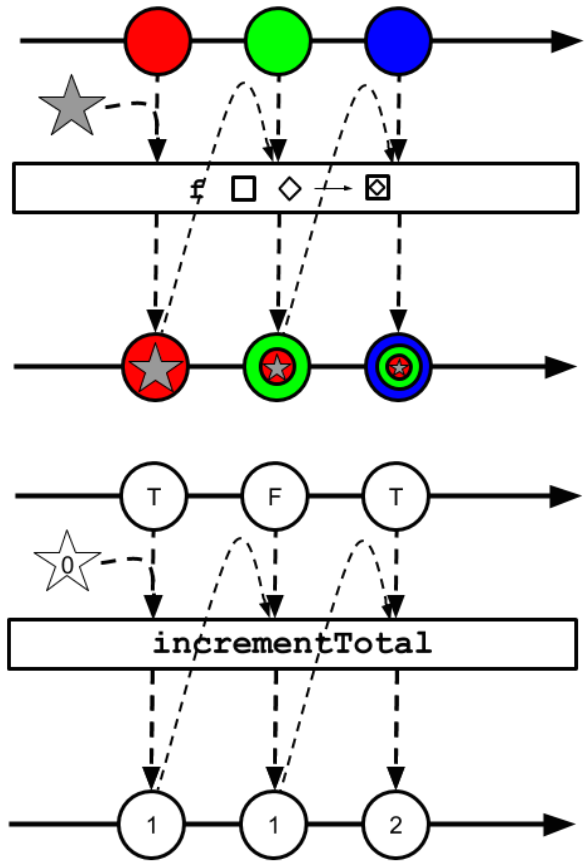
```
incrementTotal isClicked total =
  if isClicked then total + 1 else total
```

(Code available at `learn-elm.com/examples/foldp-clicks`.)

`foldp` is applied to three arguments: `incrementTotal`, `0`, and `Mouse.isDown`. In this case, the state is an integer that represents the number of mouse clicks. The initial state is `0` (specified by the second argument). Every time `Mouse.isDown` has a new value, `incrementTotal` is called. It takes the current value of `Mouse.isDown` (a boolean) and the current state (an integer) and produces a new state (an integer).

In `main`, we can use `Signal.map` and `show` to display the current value of the state (i.e. the counter).

The following diagram (inspired by [24]) may help you understand how `foldp` operates in the general case and in the case described above.



[0:10 - 0:20]

Now that we have the necessary tools, let's return to the program we described earlier. We want to be able to "draw" by moving the mouse around the screen. Here, our state will be a list of forms that we want to draw. Each time the mouse moves to a new position, we want

to add a new circle to the list.

First, we'll need a function that takes a mouse position and a list of forms and adds a new circle at the mouse position to the list. In order to add a new element to a list, we can use the function `List.append`, which takes two lists and combines them.

```
drawCircle : (Int,Int) -> List Form -> List Form
drawCircle (x,y) circleList =
  List.append circleList [
    circle 10
      |> filled red
      |> move (toFloat x - halfWidth, halfWidth - toFloat y)
  ]
```

Now we can use `drawCircle` with `foldp`. The initial state will be an empty list and the input signal will be the mouse position.

```
Signal.foldp drawCircle [] Mouse.position
```

This will give us a signal of lists of forms. We can see the forms accumulate using the `show` function:

```
main : Signal Element
main =
  Signal.map show (Signal.foldp drawCircle [] Mouse.position)
```

(Code available at `learn-elm.com/examples/drawing-0`.)

In order to actually draw these forms, we'll need one more function:

```
draw : List Form -> Element
draw form_list =
  collage (halfWidth * 2) (halfWidth * 2) form_list
```

In `main`, we can then map this function over the signal of lists of forms:

```
main : Signal Element
main =
  Signal.map draw (Signal.foldp drawCircle [] Mouse.position)
```

(Code available at `learn-elm.com/examples/drawing-1`.)

If we try this program out, we'll see that it draws circles as we move our mouse around!

[0:20 - 0:30]

After using this program for a while, we'll see that it's pretty difficult to draw anything when the computer always draws a circle wherever the mouse is. It would be better if the computer only drew circles when the mouse is clicked.

In order to do this, we'll need two signals: `Mouse.position` and `Mouse.isDown`. We can combine them into a single signal with the following function:

```
inputSignal : Signal ((Int,Int), Bool)
inputSignal = Signal.map2 (,) Mouse.position Mouse.isDown
```

`(,)` is shorthand for a function that takes two values and returns a pair consisting of those values. We could implement a function `f` that performs the same operation: `f x y = (x,y)`

`inputSignal` is a signal of a pair of a pair of integers and a boolean representing the mouse position and whether or not the mouse is clicked. For example, if the mouse were at the origin and the mouse button was clicked, `inputSignal` would emit the value `((0, 0), True)`. If we replace `Mouse.position` with `inputSignal` in `main`, then the first argument to `drawCircle` must be a pair of a pair of integers and a boolean.

```
drawCircle : ((Int,Int), Bool) -> List Form -> List Form
drawCircle ((x,y), isClicked) circleList = ...
```

Navigate to `learn-elm.com/examples/drawing-2` and take a few minutes to change `drawCircle` so that it only adds a circle to the state if the mouse is clicked. Otherwise, it should return the given state unchanged.

If you have time left over, try changing the color of the circles depending on which quadrant the mouse is in.

Review solution:

```
drawCircle : ((Int,Int), Bool) -> List Form -> List Form
drawCircle ((x,y), isClicked) circleList =
  if isClicked
    then (List.append circleList [
      circle 10
        |> filled red
        |> move (toFloat x - 250, 250 - toFloat y)
    ])
  else circleList
```

[0:30 - 0:40]

Now suppose we want to be able to erase everything we've drawn by clicking the space key. We can use a new signal, `Keyboard.space`. Similar to `Mouse.isDown`, `Keyboard.space` is a signal of booleans that indicates whether or not the space key is pressed.

Try to modify your program so that if the space key is pressed, the drawing is erased. A few hints to get you started:

1. You'll want to change `inputSignal` to combine three signals. `map3` and `(,,)` will be useful.
2. In order to "erase" the drawing, the state should be set to an empty list: `[]`.

Review solution (available at ***learn-elm.com/examples/drawing-3****):*

```
import Color exposing(..)
import Graphics.Collage exposing(..)
import Graphics.Element exposing(..)
import Keyboard exposing(..)
import Mouse exposing(..)

halfWidth = 250

inputSignal : Signal ((Int,Int), Bool, Bool)
inputSignal =
  Signal.map3 (,,) Mouse.position Mouse.isDown Keyboard.space

main : Signal Element
main =
  Signal.map draw (Signal.foldp drawCircle [] inputSignal)

draw : List Form -> Element
draw form_list = collage (halfWidth * 2) (halfWidth * 2) form_list

drawCircle : ((Int,Int), Bool, Bool) -> List Form -> List Form
drawCircle ((x,y), mouseIsClicked, spaceIsClicked) circleList =
  if spaceIsClicked
    then []
  else if mouseIsClicked
    then (List.append circleList [
      circle 10
        |> filled red
        |> move (toFloat x - halfWidth, halfWidth - toFloat y)
    ])
  else circleList
```

[0:40 - 0:45]

Our program is starting to become very cluttered with all of the different inputs. We can define a new type to cleanly represent all three inputs:

```
type alias Input =
  { mousePos : (Int,Int)
  , mouseIsClicked : Bool
  , spaceIsClicked : Bool
  }
```

This allows us to create a new type, `Input`, that is made up of the types of our three inputs. `Input` is known as a **record type**. Record types allow us to store related values in a single type.

We could explicitly construct a value of type `Input` as follows:

```
inputExample =
  { mousePos = (0,0)
  , mouseIsClicked = True
  , spaceIsClicked = False
  }
```

In order for to create a `Input Signal` with (1) the current location of the mouse, (2) whether or not the mouse is being clicked, and (3) whether or not the space key is being clicked, we can change `inputSignal` to the following:

```
inputSignal : Signal Input
inputSignal =
  Signal.map3 Input Mouse.position Mouse.isDown Keyboard.space
```

This will take the values of the three signals and construct a value of type `Input`. Now we can change `drawCircle` accordingly:

```
drawCircle : Input -> List Form -> List Form
drawCircle userInput circleList = ...
```

If we make these changes and press "compile," we'll see that the computer no longer knows what `x`, `y`, `mouseIsClicked`, and `spaceIsClicked` are. These values are now fields within `userInput`. We can access `(x,y)` with `userInput.mousePos`, `mouseIsClicked` with `userInput.mouseIsClicked`, and `spaceIsClicked` with `userInput.spaceIsClicked`. Notice that `userInput` is the name of the value with type `Input` and the word after the period is the name of the field within `Input`.

We can easily replace `mouseIsClicked` with `userInput.mouseIsClicked` and `spaceIsClicked` with `userInput.spaceIsClicked`, but how do we get `(x,y)` from `userInput.mousePos`? We can do this with **pattern matching**.

```
let (x,y) = userInput.mousePos in ...
```

Pattern matching allows us to name each of the two integers in the integer pair, `mousePos`. If we put this at the top of our function, we'll see that our program compiles without any errors.

We can also use pattern matching to avoid writing `userInput.spaceIsClicked` and `userInput.spaceIsClicked`. At the top of our function, we can write:

```
let { mousePos, mouseIsClicked, spaceIsClicked } = userInput in
let (x,y) = mousePos in
```

Now we can use `x`, `y`, `mouseIsClicked`, and `spaceIsClicked` as before. With our new `Input` type, we can even add more input signals without changing the type of the first argument to the `draw` function.

(Code available at `learn-elm.com/examples/drawing-4`.)

[0:45 - 1:00]

Our drawings would be a lot more interesting if we were able to change the color of the circle. We can use `Keyboard.presses`, which gives us the key code for the most recently pressed key on the keyboard. For example, if the user presses the 'b' key, we could draw blue circles instead of red ones.

Try modifying your program to allow the user to change colors by clicking different keys on the keyboard. The type of `Keyboard.presses` is `Signal KeyCode`.

You might also find `Char.fromCode` useful:

```
fromCode : KeyCode -> Char
Convert from unicode.
```

Finished code:

(Available at `learn-elm.com/examples/drawing-5`.)

```
import Char exposing (..)
import Color exposing (..)
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)
import Keyboard exposing (..)
import Mouse exposing (..)

type alias Input =
  { mousePos : (Int,Int)
  , mouseIsClicked : Bool
  , spaceIsClicked : Bool
  , keyCode : KeyCode
  }

halfWidth = 250

inputSignal : Signal Input
inputSignal = Signal.map4 Input
  Mouse.position Mouse.isDown Keyboard.space Keyboard.presses

main : Signal Element
main =
  Signal.map draw (Signal.foldp drawCircle [] inputSignal)

draw : List Form -> Element
draw form_list =
  collage (halfWidth * 2) (halfWidth * 2) form_list

drawCircle : Input -> List Form -> List Form
drawCircle userInput circleList =
```

```
let { mousePos, mouseIsClicked, spaceIsClicked, keyCode } =
  userInput in
let (x,y) = mousePos in
if spaceIsClicked then [] else
let c =
  if Char.fromCode keyCode == 'r' then red
  else if Char.fromCode keyCode == 'y' then yellow
  else if Char.fromCode keyCode == 'b' then blue
  else black in
if mouseIsClicked
  then (List.append circleList [
    circle 10
      |> filled c
      |> move (toFloat x - halfWidth, halfWidth - toFloat y)
  ])
else circleList
```

A.9 Paddle Game

Today, you'll have the opportunity to work through the development of your first game. The finished product will look like this: `learn-elm.com/examples/paddle/result`. We'll walk you through the steps, introducing any new syntax or functions that you'll need!

1. Let's start by creating a ball. We can define a new type for the ball using `type alias`, which we saw last week. For now, you'll probably want to give the ball an x-coordinate, a y-coordinate, and a radius.

   Solution:

   ```
   type alias Ball =
     { x : Float
     , y : Float
     , r : Float
     }
   ```

2. Now you'll want to create an actual ball that has the type you just defined. Recall that last week we defined the type `Input` as follows:

   ```
   type alias Input =
     { mousePos : (Int,Int)
     , mouseIsClicked : Bool
     , spaceIsClicked : Bool
     }
   ```

   In order to create a value of type of `Input`, we wrote:

   ```
   inputExample =
     { mousePos = (0,0)
     , mouseIsClicked = True
     , spaceIsClicked = False
     }
   ```

   Using this example for guidance, create an initial ball (called `initBall`) with type `Ball`.

   Solution:

   ```
   initBall =
     { x = 0
     , y = 0
     , r = 15
     }
   ```

3. Now define a function `draw` that takes an argument of type `Ball` and evaluates to an `Element`. In other words, write a function that draws a ball. Remember that you can

access values inside a record type using a period.

You might also want to define a variable called `halfWidth` that specifies half the width of the playing area (e.g. `250`). This value will be used throughout your program so defining (and using) a variable to the store it will help make your code cleaner and easier to read. You might also want to draw a colored rectangle that is the size of the collage so that you can see the boundaries of the playing area.

Solution:

```
draw : Ball -> Element
draw ball =
  collage (halfWidth * 2) (halfWidth * 2)
    [ rect (halfWidth * 2) (halfWidth * 2)
        |> filled gray
        |> move (0,0)
    , circle ball.r
        |> filled black
        |> move (ball.x, ball.y)
    , rect paddle.width paddle.height
        |> filled purple
        |> move (paddle.x, paddle.height / 2 - halfWidth)
    ]
```

4. At this point, you should be able to write a `main` function that draws the ball.

   Solution:

   ```
   main : Element
   main = draw initBall
   ```

5. Now let's make the ball bounce around the screen. Start by adding a horizontal velocity (`vx`) and a vertical velocity (`vy`) to the type `Ball` and your initial ball. The velocity specifies how fast the ball moves and in what direction. For example, if `vx` and `vy` were both 1, the ball would move up and to the right at a speed of 1 unit per time unit.

   Here, realize that you will be modifying code you wrote previously (instead of writing code from scratch, as we have up until this point).

   Solution:

   ```
   type alias Ball =
     { x : Float
     , y : Float
     , vx : Float
     , vy : Float
   ```

```
  , r : Float
  }

initBall =
  { x = 0
  , y = 0
  , vx = 4
  , vy = 8
  , r = 15
  }
```

6. At some regular interval, we'd like to increment `x` by `vx` (i.e. add `vx` to `x`) and increment `y` by `vy` (i.e. add `vy` to `y`). In order to do this, we'll need a new function that an Elm library provides:

   ```
   fps : number -> Signal Time
   ```

   `Time.fps 30` will give us a signal that emits a new value (the elapsed time since the last value) 30 times per second. `fps` stands for frames per second.

   Write a new function `updateBall` that takes a `Time` and a `Ball` and increments `x` by `vx` and `y` by `vy`. In order to change certain values in a record, we'll need new syntax. If `ball` is the name of the ball, then

   ```
   { ball |
         x = ball.x + ball.vx,
         y = ball.y + ball.vy
   }
   ```

   gives us a record that has the same values as `ball`, except `x` is replaced with `x + vx` and `y` is replaced with `y + vy`.

   Solution:

   ```
   updateBall : Time -> Ball -> Ball
   updateBall t ball =
     { ball |
         x = ball.x + ball.vx,
         y = ball.y + ball.vy
     }
   ```

7. Now, using `Signal.map`, `Signal.foldp`, and `Time.fps 30`, update `main` so that the ball moves each time `Time.fps 30` emits a value.

   Solution:

```
main : Signal Element
main = Signal.map draw
  (Signal.foldp updateBall initBall (Time.fps 30))
```

At this point, if you press "compile," the ball should start moving and disappear off the screen.

8. We now want the ball to "bounce" off the sides of the screen. Start by defining two functions: `isTouchingSide` and `isTouchingTopOrBottom`. The first function should return true only if the ball is touching the left or right sides of the screen and the second function should return true only if the ball is touching the top or bottom of the screen. Drawing a picture might help you work out the math!

   Solution:

```
isTouchingSide : Ball -> Bool
isTouchingSide ball =
  ball.x < (-halfWidth + ball.r) ||
    ball.x > (halfWidth - ball.r)

isTouchingTopOrBottom : Ball -> Bool
isTouchingTopOrBottom ball =
  ball.y < (-halfWidth + ball.r) ||
    ball.y > (halfWidth - ball.r)
```

9. Modify `updateBall` so that vx is negated if the ball is touching a side of the screen and `vy` is negated if the ball is touching the top or bottom of the screen.

   Solution:

```
updateBall : Time -> Ball -> Ball
updateBall t ball =
  let newVx = if isTouchingSide ball then -ball.vx
    else ball.vx in
  let newVy = if isTouchingTopOrBottom ball then -ball.vy
    else ball.vy in
  { ball |
      x = ball.x + newVx,
      y = ball.y + newVy,
      vx = newVx,
      vy = newVy
  }
```

   If you press "compile" at this point, the ball should bounce around the screen.

10. Now let's add the paddle. Start by defining a new type, `Paddle`, and an initial paddle, `initPaddle`. `Paddle` should have an x-coordinate, a height, and a width. You might

also want to define a new type, `Game`, and an initial game, **`initGame`**. A value of type `Game` should have a value of type `Ball` and a value of type `Paddle`.

Solution:

```
type alias Paddle =
  { x : Float
  , width : Float
  , height : Float
  }

initPaddle =
  { x = 0
  , width = 50
  , height = 20
  }

type alias Game =
  { ball : Ball
  , paddle : Paddle
  , playing : Bool
  }

initGame =
  { ball = initBall
  , paddle = initPaddle
  , playing = True
  }
```

11. Modify **`draw`** so that it takes a `Game` as an argument and draws both the paddle and the ball.

Solution:

```
draw : Game -> Element
draw game =
  let (ball, paddle) = (game.ball, game.paddle) in
  collage (halfWidth * 2) (halfWidth * 2)
    [ rect (halfWidth * 2) (halfWidth * 2)
        |> filled gray
        |> move (0,0)
    , circle ball.r
        |> filled black
        |> move (ball.x, ball.y)
    , rect paddle.width paddle.height
        |> filled purple
        |> move (paddle.x, paddle.height / 2 - halfWidth)
```

```
    ]
```

12. Create a value called `inputSignals`:

```
inputSignals : Signal (Int,Int)
inputSignals = Signal.sampleOn (Time.fps 30) (Mouse.position)
```

    At a rate of 30 frames per second, `inputSignals` will emit the current position of the mouse. This ensures that the ball (which moves at a rate of 30 frames per second) and paddle (which will be controlled by the mouse) will move at the same speed.

13. Define a new function called `update` that takes a pair of integers (the mouse's position) and a `Game` and returns an updated `Game`. In particular, the ball should bounce around the screen as before and the paddle should follow the x-coordinate of the mouse. Replace `initBall` with `initGame` and `updateBall` with `updateGame`. You may want to use `updateBall` in `updateGame`, but notice that `updateBall` will no longer need an argument of type `Time`.

    Solution:

```
updateGame : (Int,Int) -> Game -> Game
updateGame (mouseX, mouseY) game =
  let (ball, paddle) = (game.ball, game.paddle) in
  let newPaddle =
    { paddle | x = toFloat mouseX - halfWidth } in
  let newBall = updateBall ball newPaddle in
  { game |
      ball = newBall,
      paddle = newPaddle
  }

updateBall : Ball -> Paddle -> Ball
updateBall ball paddle =
  let newVx =
    if isTouchingSide ball
      then -ball.vx
    else ball.vx in
  let newVy =
    if (isTouchingTop ball || isTouchingPaddle ball paddle)
      then -ball.vy
    else ball.vy in
  { ball |
      x = ball.x + newVx,
      y = ball.y + newVy,
      vx = newVx,
      vy = newVy
  }
```

If you press "compile" at this point, the ball should bounce around the screen and the paddle should move with the mouse.

14. If the ball hits the paddle, we want it to bounce off. Create a new function called `isTouchingPaddle` that returns true if the ball is touching the paddle and false otherwise. Incorporate this function into `updateGame` or `updateBall` so that the ball bounces off the paddle.

    Solution:

```
isTouchingPaddle : Ball -> Paddle -> Bool
isTouchingPaddle ball paddle =
  ball.x > (paddle.x - paddle.width / 2) &&
    ball.x < (paddle.x + paddle.width / 2) &&
    (ball.y - ball.r) < paddle.height - halfWidth
```

15. If the ball hits the bottom of the screen, we want the game to stop. Split the function `isTouchingBottomOrTop` into two functions: one that checks if the ball is touching to bottom and one of the ball is touching the top. If the ball touches the bottom, we should stop the game. This may require adding a boolean field to `Game`, `hasLost`. At the beginning of the `updateGame` function, check if `hasLost` is true and return the game unchanged if so.

    Solution:

```
isTouchingTop : Ball -> Bool
isTouchingTop ball = ball.y > (halfWidth - ball.r)

isTouchingBottom : Ball -> Bool
isTouchingBottom ball = ball.y < (-halfWidth + ball.r)

updateGame : (Int, Int) -> Game -> Game
updateGame (mouseX, mouseY) game =
  if game.hasLost
    then game
  else
    let (ball, paddle) = (game.ball, game.paddle) in
    if isTouchingBottom ball
      then { game | hasLost = True }
    else
      let newPaddle =
        { paddle | x = toFloat mouseX - halfWidth } in
      let newBall = updateBall ball newPaddle in
      { game |
          ball = newBall,
          paddle = newPaddle
```

```
        }
```

16. If you finish early, try to give the user a way to pause or restart the game!

Finished code:

(Available at `learn-elm.com/examples/paddle`.)

```
import Color exposing (..)
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)
import Time exposing (..)
import Mouse exposing (..)

type alias Ball =
  { x : Float
  , y : Float
  , vx : Float
  , vy : Float
  , r : Float
  }

initBall =
  { x = 0
  , y = 0
  , vx = 4
  , vy = 8
  , r = 15
  }

type alias Paddle =
  { x : Float
  , width : Float
  , height : Float
  }

initPaddle =
  { x = 0
  , width = 50
  , height = 20
  }

type alias Game =
  { ball : Ball
  , paddle : Paddle
  , hasLost : Bool
  }
```

```
initGame =
  { ball = initBall
  , paddle = initPaddle
  , hasLost = False
  }

halfWidth = 250

main : Signal Element
main = Signal.map draw (Signal.foldp updateGame initGame
   inputSignals)

inputSignals : Signal (Int, Int)
inputSignals = Signal.sampleOn (Time.fps 30) (Mouse.position)

draw : Game -> Element
draw game =
  let (ball, paddle) = (game.ball, game.paddle) in
  collage (halfWidth * 2) (halfWidth * 2)
    [ rect (halfWidth * 2) (halfWidth * 2)
        |> filled gray
        |> move (0,0)
    , circle ball.r
        |> filled black
        |> move (ball.x, ball.y)
    , rect paddle.width paddle.height
        |> filled purple
        |> move (paddle.x, paddle.height / 2 - halfWidth)
    ]

isTouchingSide : Ball -> Bool
isTouchingSide ball =
  ball.x < (-halfWidth + ball.r) || ball.x > (halfWidth - ball.r)

isTouchingTop : Ball -> Bool
isTouchingTop ball = ball.y > (halfWidth - ball.r)

isTouchingBottom : Ball -> Bool
isTouchingBottom ball = ball.y < (-halfWidth + ball.r)

isTouchingPaddle : Ball -> Paddle -> Bool
isTouchingPaddle ball paddle =
  ball.x > (paddle.x - paddle.width / 2) &&
    ball.x < (paddle.x + paddle.width / 2) &&
    (ball.y - ball.r) < paddle.height - halfWidth
```

```
updateGame : (Int, Int) -> Game -> Game
updateGame (mouseX, mouseY) game =
  if game.hasLost
    then game
  else
    let (ball, paddle) = (game.ball, game.paddle) in
    if isTouchingBottom ball
      then { game | hasLost = True }
    else
      let newPaddle =
        { paddle | x = toFloat mouseX - halfWidth } in
      let newBall = updateBall ball newPaddle in
      { game |
          ball = newBall,
          paddle = newPaddle
      }

updateBall : Ball -> Paddle -> Ball
updateBall ball paddle =
  let newVx =
    if isTouchingSide ball
      then -ball.vx
    else ball.vx in
  let newVy =
    if (isTouchingTop ball || isTouchingPaddle ball paddle)
      then -ball.vy
    else ball.vy in
  { ball |
      x = ball.x + newVx,
      y = ball.y + newVy,
      vx = newVx,
      vy = newVy
  }
```

A.10 Final Projects

Today, you'll have the opportunity to work on a final project! You can create anything of interest to you - an animation, a game, a tool, or even a website. The project should be at least as complex as the paddle game we developed. Here are three examples of appropriate final projects:

1. A "falling objects" game: `learn-elm.com/examples/dog/result`
   (code available at: `learn-elm.com/examples/dog`)
2. Tic-tac-toe: `learn-elm.com/examples/tic-tac-toe/result`
   (code available at: `learn-elm.com/examples/tic-tac-toe`)
3. A tool to create photo filters: `learn-elm.com/examples/filters/result`
   (code available at: `learn-elm.com/examples/filters`)

You may choose to recreate one of these examples or you can work on something completely new.

When we worked on the paddle game last week, we built up the game step-by-step. Before you begin programming, create a step-by-step plan for developing your final project. The plan does not need to be as detailed as last week's plan, but ensure that each step is testable. In other words, after completing each step, you should be able to compile your program and determine if some piece of the project is working.

Along with each step, include the "test" that will tell you whether you've completed the step successfully. You should also include relevant notes or questions.

For example, if we writing a plan to develop the paddle game, it might look as follows:

1. Create a bouncing ball.
   (a) Define a type and initial value for the ball. At the end of this step, the program should not have any type errors.
   (b) Write a function that can draw the ball.
   (c) Make the ball bounce.
      - Make the ball move at some time interval.
      - Make the ball bounce off the sides.
2. Create a moving paddle.
   (a) Define a type and initial value for the paddle. At the end of this step, the program should not have any type errors.

   (b) Might want to create a game type and value that includes the paddle and ball (so that they can be easily updated and drawn together).
   (c) Update the function that draws the ball to also draw the paddle.
   (d) Make the paddle move left and right according to the position of the mouse.

3. Determine when the game is over.
   (a) Determine if the ball hits the ground and update the status of the game.
   (b) Stop everything if the game is over.

4. Possible extensions:
   (a) Enable the user to pause or restart the game.
   (b) Keep track of points.

Creating a new project from scratch can seem really daunting; your plan will help you break it into smaller, more manageable pieces.

Here are a few tips that you might find helpful as you develop your final project:

- We've seen how to use `type alias` to create new types from existing types. This can be useful since there are often functions already defined for existing types.

  You might, however, want to create completely new types that do not use any existing types. This is appropriate if you want to define all operations on your type. For example, if you are creating a game of Tic-tac-toe, you might want a type called `Player` to represent the current player or the player whose piece is in a given spot. The possible values of `Player` might be `X`, `O`, or `None`.

  You can define the type `Player` as follows: `type Player = X | O | None`. This creates a new type called `Player` where the values of type `Player` are `X`, `O`, or `None`. There are no operations defined for values of type `Player`. You can now define them!

  For example, you might define a function that evaluates to the string representation of a `Player`:

  ```
  toString : Player -> String
  toString player =
    if player == X then "X"
    else if player == O then "O"
    else "None"
  ```

- In order to access a particular element of a list, include the `nth` function available at `learn-elm.com/examples/nth` in your program. This function takes three arguments: the index of the element you want to retrieve, the list you want to retrieve from, and a default value (in case there is no element at the given index in the given list). The default value must have the same type as the type of the elements in the list.

- If you want to perform some operation that we haven't seen in class, there might be a helpful library or function available. The best place to look is the documentation for the core Elm libraries: `package.elm-lang.org/packages/elm-lang/core/3.0.0/`.

Below are plans for each of the sample programs. Your students' plans should be at least as detailed as these plans, but they do not have to be identical.

Falling Objects Game

1. Create a character.
   (a) Define a type and initial value for the character. At the end of this step, the program should not have any type errors.
   (b) Write a function that can draw the character.
   (c) Make the character move left and right according to the arrow keys.
2. Create a "good" object and a "bad" object.
   (a) Define a type and initial value for the good object. At the end of this step, the program should not have any type errors.
   (b) Have the good object start at the top of the screen and fall to the bottom.
   (c) If the good object reaches the bottom of the screen, have it return to a random location at the top of the screen.
      - Will need a random number generator!
   (d) Repeat steps 2(a) – 2(c) for the bad object.
3. Add a point system.
   (a) Add a value to keep track of points.
   (b) Determine if the character "catches" the good object and add points if so.
   (c) Determine if the character "catches" the bad object and subtract points if so.
   (d) Ensure objects return to the top of the screen if they are "caught."

Tic-Tac-Toe

1. Create the game.
   (a) Define a type and initial value for the game, including the pieces on the board and the current player. At the end of this step, the program should not have any type errors.
2. Write drawing functions.

(a) Write a function to draw the board.

- Draw the board's lines.
- Draw the boards pieces.

(b) Write a function that draws an X given an index (0-8, corresponding to the spaces on the board). Test with all indices.

(c) Write a function that draws an O given an index. Test with all indices.

3. Update the game.

(a) When the player clicks on a space, update the board's value.

- Determine the index of the space given the mouse's position.
- Determine if the move is valid (no other piece is in the space).
- If the move is valid, add the player's piece to the board and update the current player.

Photo Filter Tool

1. Create a filter.

(a) Define a type and initial value for the filter, including the red, green, blue, and alpha values of the filter. At the end of this step, the program should not have any type errors.

2. Display an image, filter, and sliders.

(a) Display an image.

(b) Draw a colored rectangle over the image, using the red, green, blue, and alpha values specified by the filter.

(c) Draw four sliders (red, green, blue, alpha value) such that the position of each slider represents the current values of the filter.

3. Update the filter.

(a) When the user moves a slider, update the filter.

- When the user presses the mouse, determine on which (if any) slider's circle the mouse is.
- If it is on a slider's circle, move the circle left and right with the mouse and update the filter accordingly.

B Modular Material

Below are short lessons (less than 5 minutes each) that can be taught whenever time permits. Note that the first two topics, comments and imports, are fundamental and should be incorporated into the curriculum as soon as possible. They are not included in any particular lesson because there is some flexibility in when they can be taught; these topics are not strict "prerequisites" for other material. The `rgba` function was written at the request of students in the pilot course.

B.1 Comments

Programmers commonly revisit their own code or read, use, or modify code that other programmers have written. Especially if these programs are long or confusing, it is helpful to have "notes," written in a natural language, alongside the code.

Most programming languages allow us to write "notes" in our code. These notes are called comments. In Elm, comments start with `{-` and end with `-}` or start with `--` and end at the end of the line. The computer knows to ignore anything between or following these characters so you're free to write words that the computer won't necessarily understand.

For example, we can add descriptive comments to a simple program as follows:

```
import Color exposing (..)
import Graphics.Collage exposing (..)

{- Draws a red circle with a radius of 100 units on a 500 by 500
   unit collage. -}
main =
  collage 500 500
    [ circle 100
        |> filled red
        |> move (0,0) -- Centers the circle at the center of the collage.
    ]
```

Documenting your code with comments will become increasingly helpful as we develop longer, more complex programs.

B.2 Imports

At the beginning of each Elm program that we have seen, there have been a number of expressions that begin with the word `import`. These `import` statements tell the computer that we want to use values that have been defined outside of our program. These statements take the following form:

```
import Module exposing (values)
```

where `values` is a list of values we want to use and `Module` is where the values are defined. A module is simply a program that contains definitions of values.

For example, we can write `import Color exposing (red)` to tell the computer we want to have access to the value `red` from the module `Color`. We can also write `import Color exposing (red,blue)` to tell the computer we want to have access to the values `red` and `blue` from the module `Color`. If we want to have access to *all* of the values defined in the `Color` module, then we can write `import Color exposing (..)`.

B.3 `rgba`

It is best to introduce this material during or after lesson 2 so that students know the terminology (function and argument).

As you have been creating drawings, you've probably found that you've wanted to use a color that is not made available by Elm. For example if you try to use "`filled pink`," you'll get an error from the computer.

If you want to create a custom color, you can use a function called `rgb`. This function takes three arguments: three integers that represent the amount of red, green, and blue respectively. Each integer should be between 0 and 255.

So you could create the color red with `rgb 255 0 0`, green with `rgb 0 255 0`, and blue with `rgb 0 0 255`. Since purple is a mix of red and blue, you could create a shade of purple with `rgb 150 0 150`. If we want to create a shade of pink, we might use `rgb 255 20 145`.

There is a similar function, `rgba` that allows you to vary the "alpha component" of the color in addition to the amount of red, green, and blue. The alpha component is essentially the opacity of the color; an alpha component of 0 means that it is completely transparent and an alpha component of 1 means that it is completely opaque.

The `rgba` function takes four arguments: three integers that represent the amount of red, green, and blue respectively and a floating point number the represents the alpha component. Each integer should be between 0 and 255, and the floating point number should be between 0 and 1.

C Homework Assignments

C.1 Drawing with Elm

1. Define or describe: "programming language"
2. Write instructions (in English) for drawing the flag of a country of your choosing.
3. Describe what the following program does:

```
import Color exposing(..)
import Graphics.Collage exposing(..)

main =
  collage 200 200
    [ circle 25
        |> filled black
        |> move (0,0)
    , circle 20
        |> filled black
        |> move (-25,25)
    , circle 20
        |> filled black
        |> move(25,25)
    ]
```

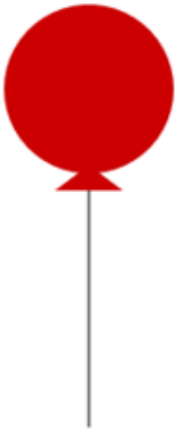
C.2 Drawing and Functions

1. Define or describe: "function"
2. Define or describe: "argument" ("input")
3. Write a function that takes a coordinate pair as an argument and draws a target (like the one below) centered at the specified location.

   Complete the code at `learn-elm.com/examples/target`.

C.3 Functions and Variables

1. Write a function that takes a coordinate pair as an argument and draws a balloon (like the one below) centered at the specified location.



Below, you will write a few Elm functions that perform mathematical computations.

For example, here's a function that doubles its argument:

```
double x = 2 * x
```

2. Write a function that adds its arguments.
3. Write a function that multiplies its arguments.
4. (Bonus) Write a function that calculates the roots of a second-order polynomial using the quadratic equation.

C.4 Types

1. Define or describe: "type"
2. Define or describe: "partial application"
3. Write a type annotation for the following function: `square x = x * x`
4. Write a type annotation for the following function: `difference (x,y) = x - y`
5. Write a type annotation for the following function:

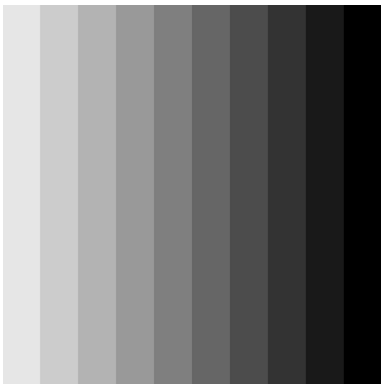
```
dot c =
  circle 10
    |> filled c
    |> move (0,0)
```

6. Write a function with the following type annotation:

```
combine : (number,number) -> number
```

C.5 Lists and Map

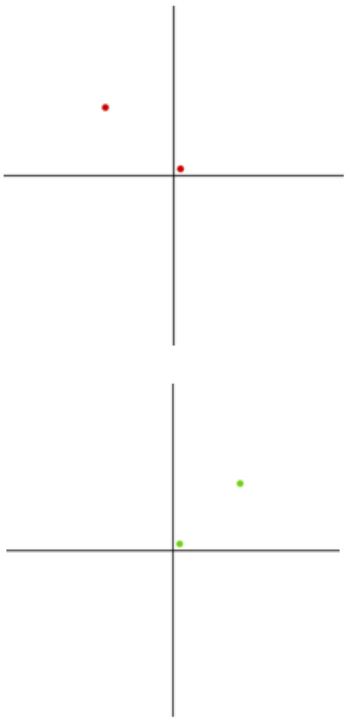
1. Define or describe: "list"
2. Use `List.map3` to simplify the code at `learn-elm.com/examples/flower-1`.
   (N.B. You can combine lists using `++`.)
3. Write an Elm program that draws the grayscale below using `List.map`.
   (Hint: `http://package.elm-lang.org/packages/elm-lang/core/3.0.0/Color#grayscale`)



C.6 Conditionals

1. Define or describe: "boolean"
2. Define or describe: "conditional"
3. Write an Elm program that, given two points, plots them on the Cartesian coordinate plane. Both points should be green if they are in the same quadrant and both points should be red if not.

   See examples below. Complete the code at `learn-elm.com/examples/draw-points`.



C.7 Mouse Signals

1. Define or describe: "signal"
2. Modify `http://learn-elm.com/examples/mouse-circle` so that the circle becomes a square whenever the mouse is clicked.
3. Write an Elm program that draws a circle that changes color according to the mouse position.

   For example, as the mouse moves from the leftmost point to the rightmost point, the circle could change from white to black.

   (Hint: `http://package.elm-lang.org/packages/elm-lang/core/3.0.0/Color#grayscale`*)*

C.8 State and Pattern Matching

1. Define or describe: "`foldp`"
2. Define or describe: "record type"
3. Write an Elm program that draws a circle that grows each time the mouse is clicked and returns to its original size whenever the space key is pressed.

D Pre/Post Survey

This survey was adapted from Wiebe et al.'s Computer Science Attitude Survey [32].

Instructions

Please note that your answers will be kept anonymous.

This survey contains a series of statements.
1. Read each statement.
2. Think of the extent to which you agree or disagree with each statement.
3. Mark your response.

Please remember:
- There are no right or wrong answers. Don't be afraid to put down what you really think.
- Don't spend a lot of time on any one questions. Move quickly!
- Complete all of the questions.

Respond to each of the following questions, using the following scale:
(1) strongly agree
(2) agree, but with reservations
(3) neutral, neither agree nor disagree
(4) disagree, but with reservations
(5) strongly disagree

Questions

1. I plan to major in computer science in college.
2. I generally feel secure attempting computer programming problems.
3. I am sure that I could do advanced work in computer science.
4. I am sure that I can learn programming.
5. I think I could handle difficult programming problems.
6. I can get good grades in computer science.
7. I have a lot of self-confidence when it comes to programming.
8. I'm not good at programming.
9. I don't think I could do advanced computer science.
10. I'm not the type to do well in computer programming.
11. I'll need programming for my future work.

12. I want to study programming because I know how useful it is.
13. Knowing programming will help me earn a living.
14. Computer science is a worthwhile and necessary subject.
15. I'll need a firm mastery of programming for my future work.
16. I will use programming in many ways throughout my life.
17. Programming is of no relevance to my life.
18. Programming will not be important to me in my life's work.
19. I see computer science as a subject I will rarely use in my daily life.
20. Taking computer science courses is a waste of time.
21. In terms of my adult life, it is not important for me to well in computer science in college.
22. I expect to have little use for programming when I get out school.