Rethinking How We Teach Programming with Elm

Patrick Ester

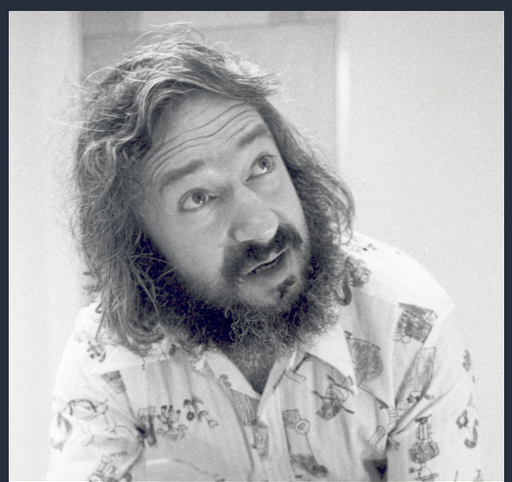
Good afternoon, my name is Patrick Ester and I am a computer teacher at the Dallas International School. I would like to share with you some design principles on getting beginner students to program. I will also talk about how the Elm programming language aligns or does not align with the principles.

Theory

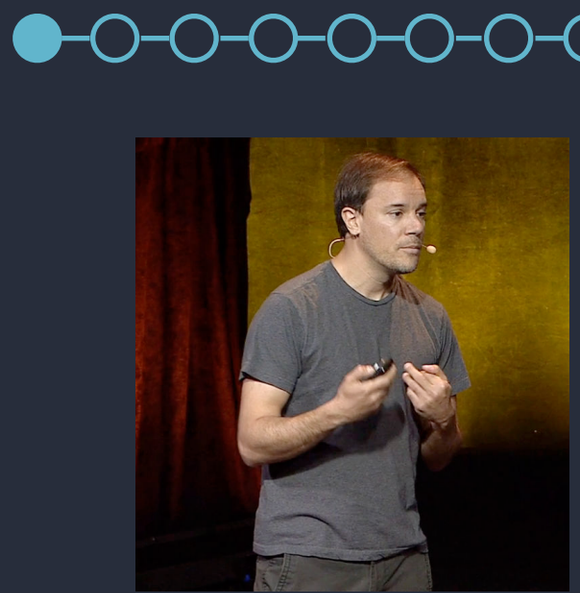
How did I get here?

What led me to these ideas? Culmination of recent work.

First and foremost, this talk is greatly influenced by my job as a teacher. Last year, the Dallas International School implemented mandatory computer science classes for students 6th to 9th grades. It was much harder than I anticipated.



Seymour Papert wrote a seminal book, Mindstorms, on children and learning through computers. Even after all of these years, Papert’s book is just as relevant.

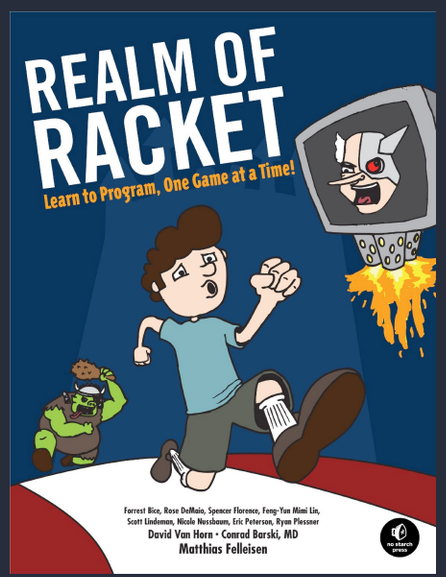


Former Apple designer Bret Victor wrote a tremendous article entitled Learnable Programming. In it, Victor lays out how a computer language and development environment should work with the user. In addition to his ideas, Victor made several demonstrations of what learnable programming would look like.

Last spring I graduated with a Masters in Emerging Media and Communication from UTD. For my capstone project, I created a site to help kids program a video game with the Elm language.

Finally, I learned a lot from Elm, a functional front-end web development language. It compiles down to HTML/CSS/JS.

The most important design principle is Constructionism, Papert’s idea that knowledge is formed through hands-on interaction.



This summer, I picked up this book to learn about Racket, a language in the Lisp family. I thought that learning through making games would be useful in my work as an educator. I was a bit dismayed to find out that there is lengthy introduction to Racket and the history of Lisp. Really? GET TO THE PROGRAMMING!!

Demo

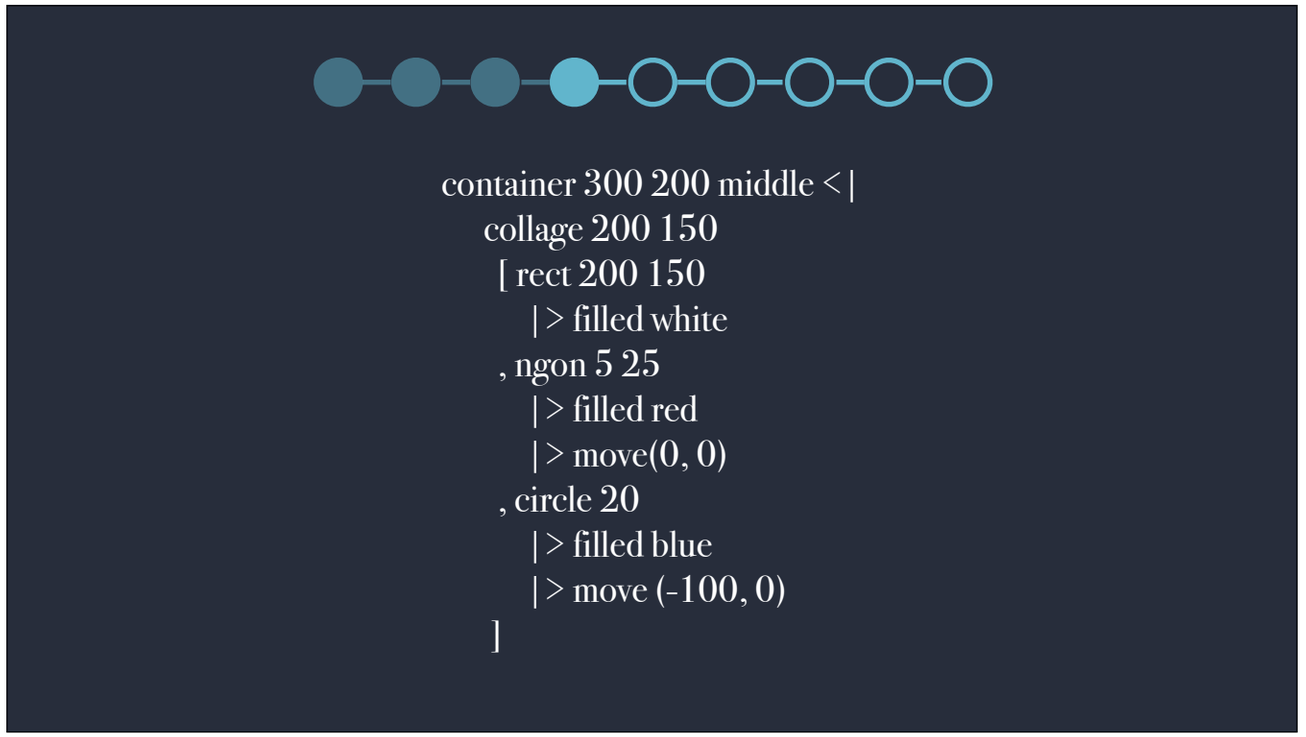
Elm Play

I would like to show you my capstone project called Elm Play. I put in three clicks and very little reading before the student is programming. If I want them to learn, then I need them to be writing the code. You cannot expect them to absorb knowledge passively. Break tradition of enlightened individual passing along knowledge.

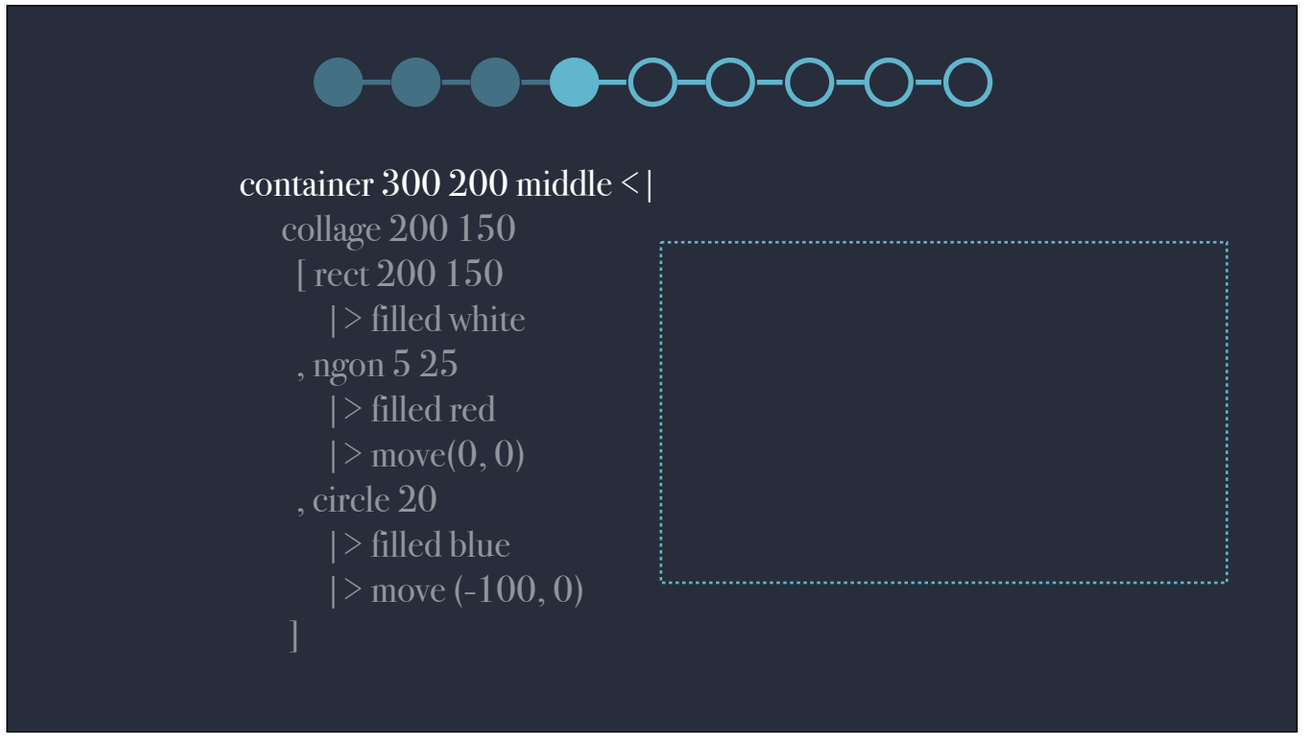
Second design principle is “Situate the Learning”. Build the thing you want to build. Doesn’t sound very insightful does it? Really, this seems like common sense stuff, but you would be surprised.

Every year, this code.org video makes the rounds during the Hour of Code.
Three types of people, hobbyists, tech workers, and the filthy rich; confirmation bias.
Adolescents won't think something is cool just because adults tell them it is; you need buy-in
Last year, my intro to CS talk was, "We're gonna program, who's with me?" Handful of students.
This year, I wised up; my talk was "You're gonna make your own video game!" Big difference.
No more "Hello world."
No more lessons that teach programming independent of something kids actually care about.
In the age of ubiquitous computing (smartphones, tablets, cheap laptops) kids have high expectations
Make games, & don't waste their time w/ superfluous information just because that's the way you learned

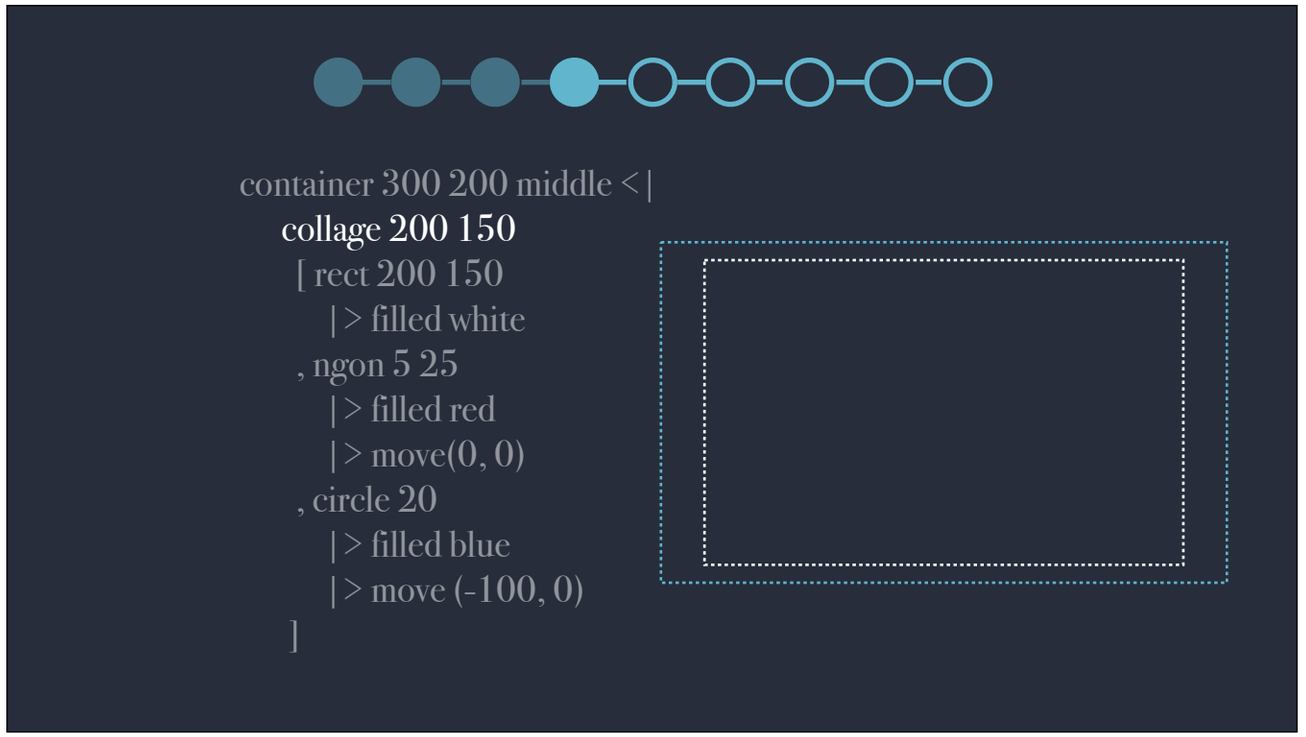
3rd design principle is readability. At first glance, this means readable syntax. And this is true; beginners need a language that is easy to read. Elm has a syntax that can be quite easy to read.



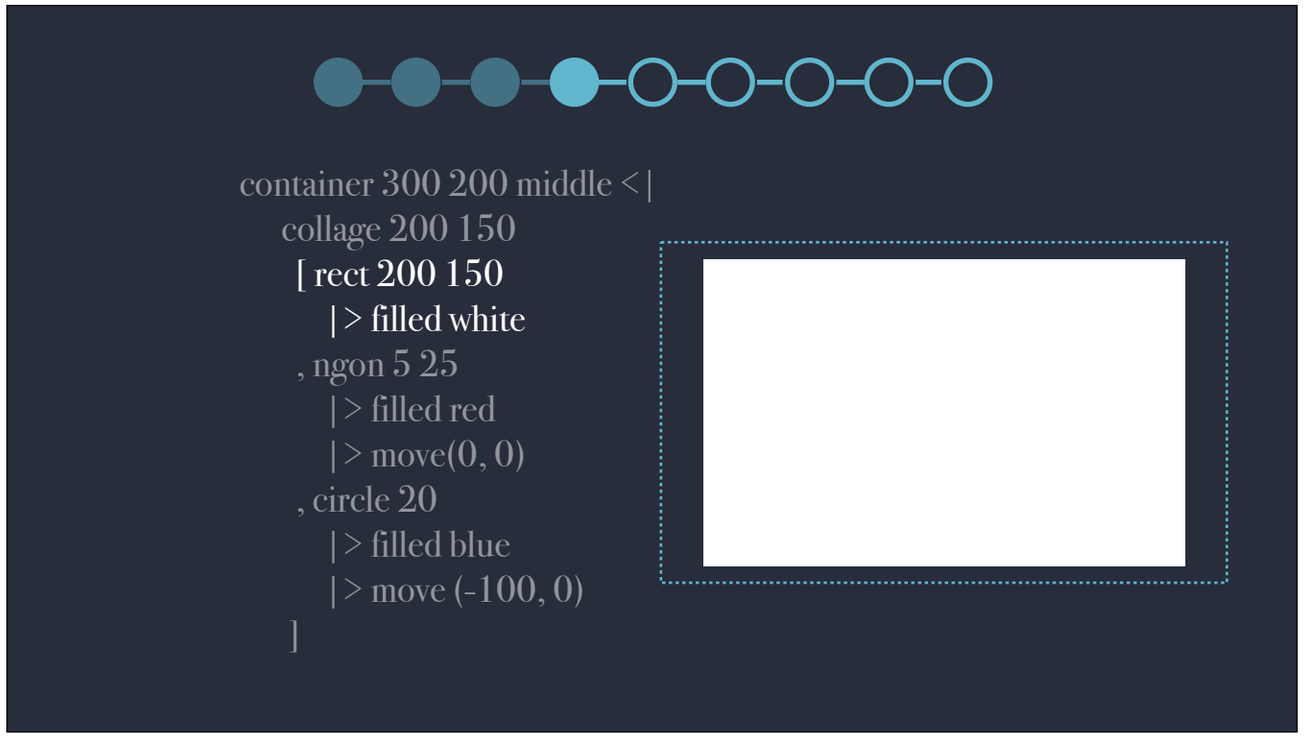
Here is some sample code that would draw a background and two shapes to the screen. I'd like to take a bit of a closer look.



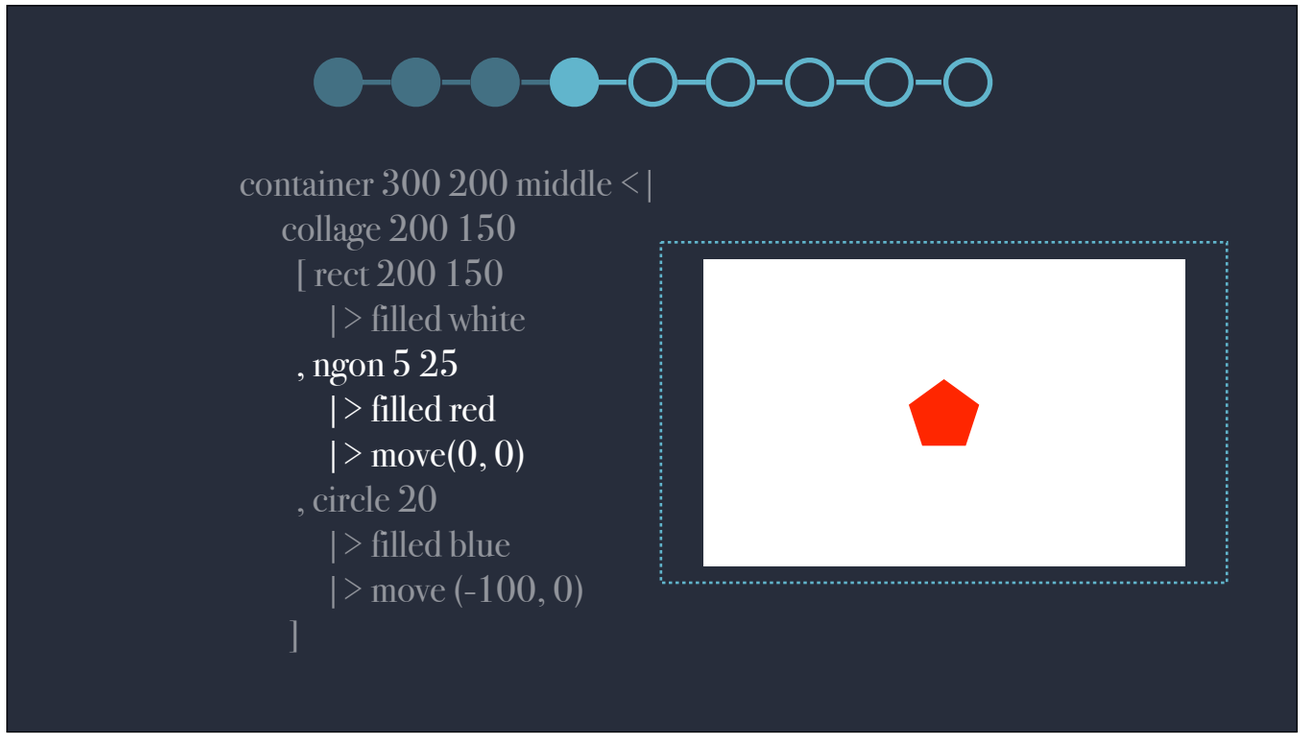
First, we have a container of size 300 by 200. Since containers hold things, the object held will be in the middle of the container.



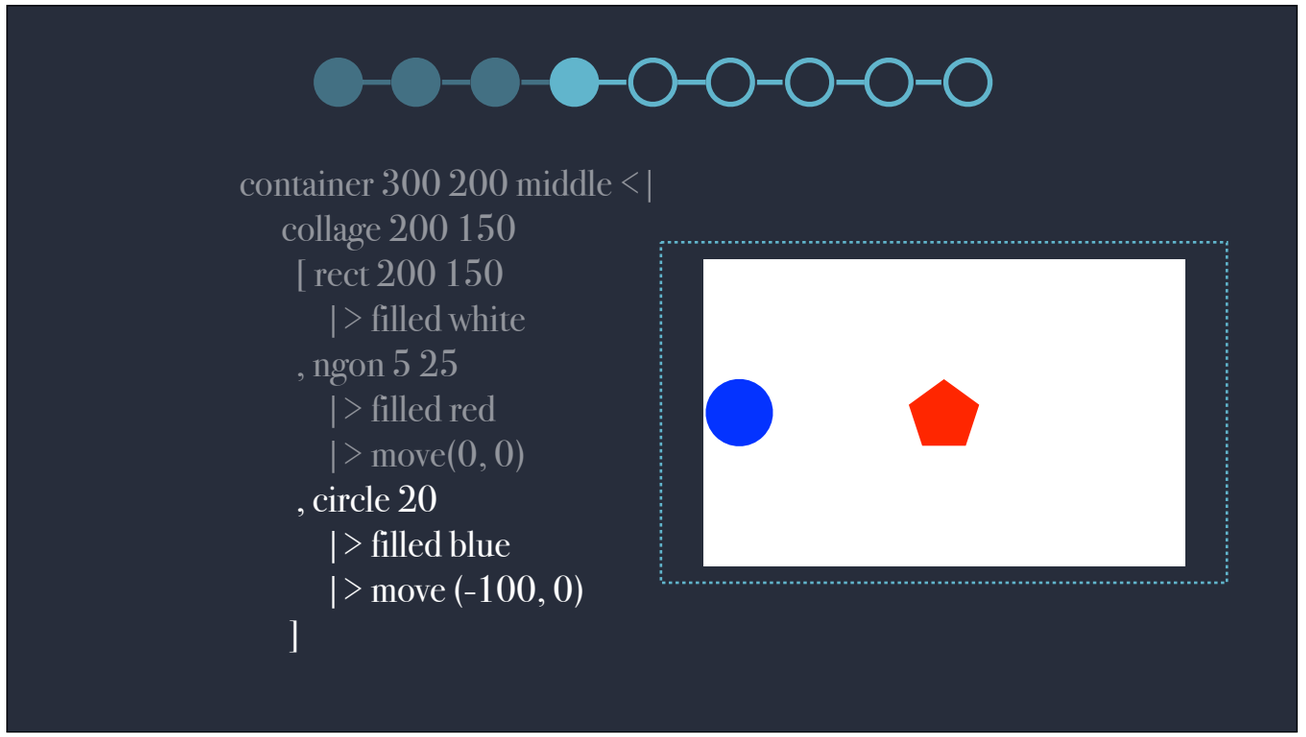
The container holds a collage of size 200 by 150. Since a collage is made of up many objects, a list of objects follows.



The first object is a rectangle of the same size as the collage, and it is filled with the color white.

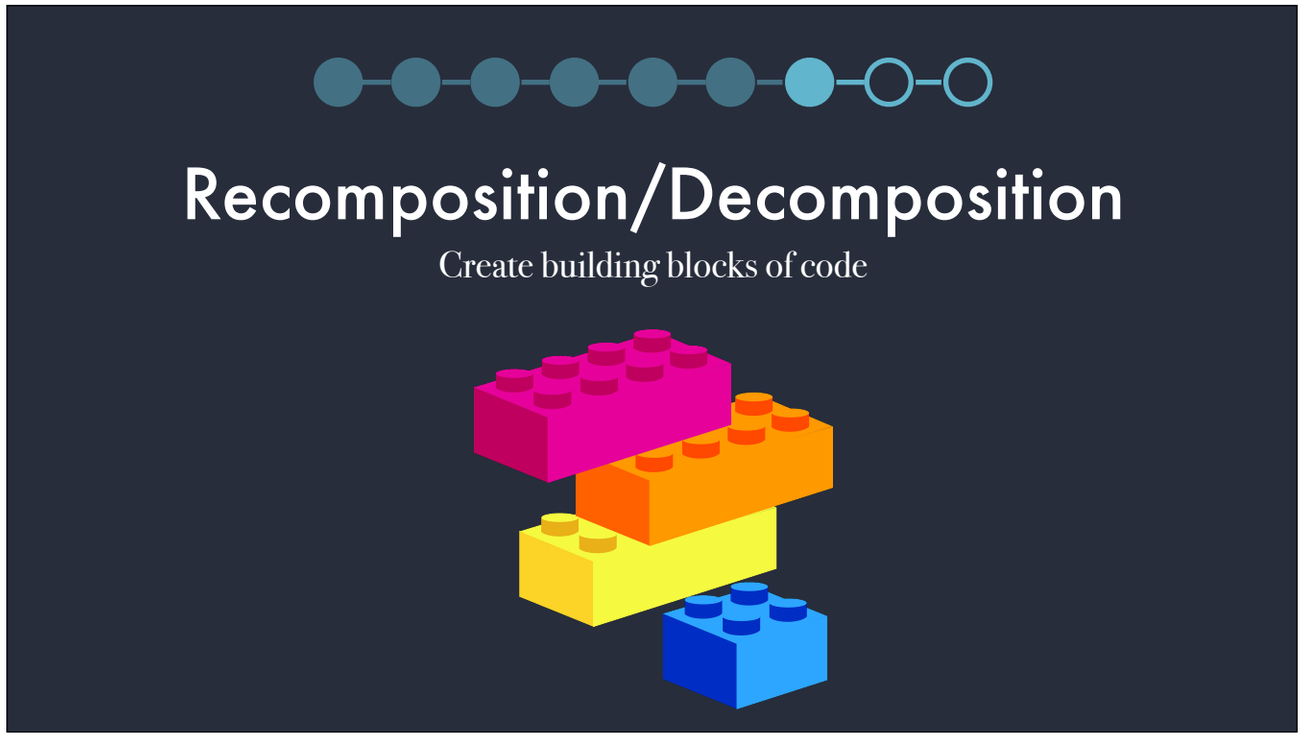


The second object is a polygon of five sides. The pentagon has a radius of 25, is filled with the color red, and is moved to the location (0, 0).

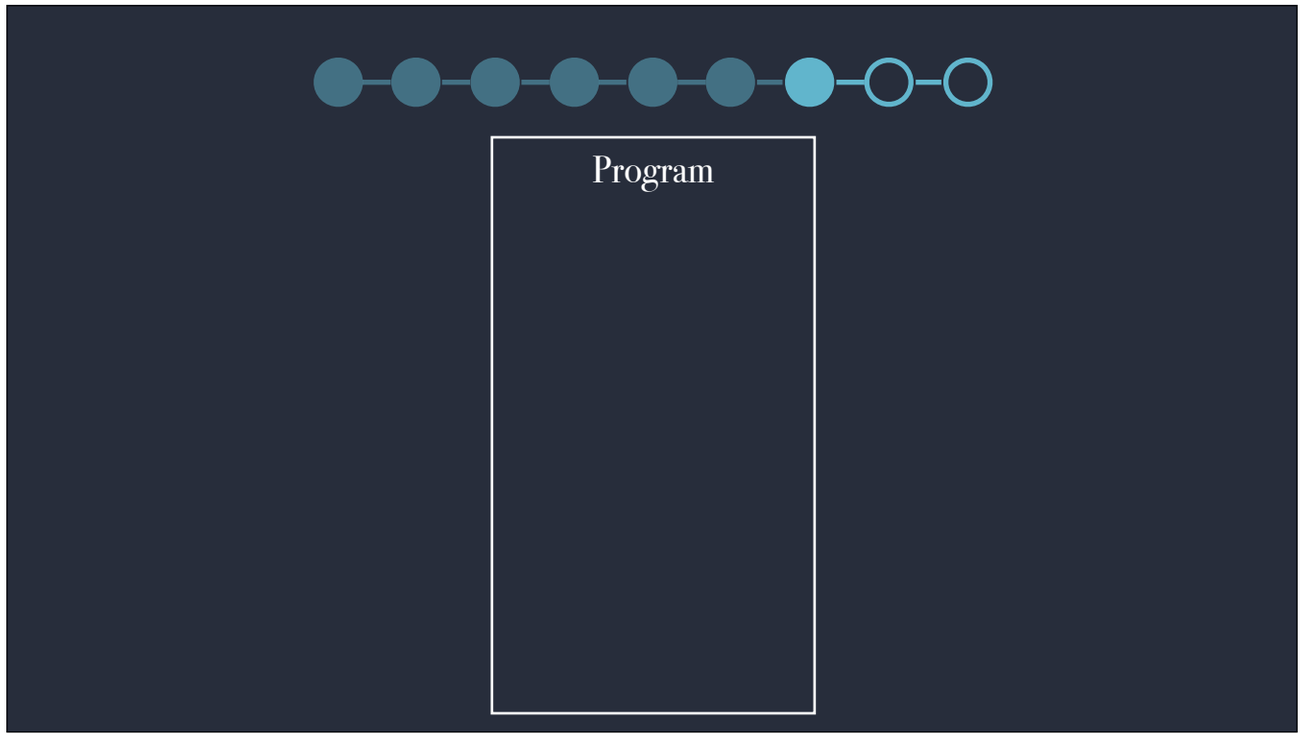


The final object is a circle with a radius of 20, filled with the color blue, and moved to the location (-100, 0). Admittedly, Elm is not the only language with a readable syntax. Ruby has a good reputation. Moreover, not all Elm code is as readable as this code here. I still believe, however, that Elm has readable syntax, an important trait for beginners.

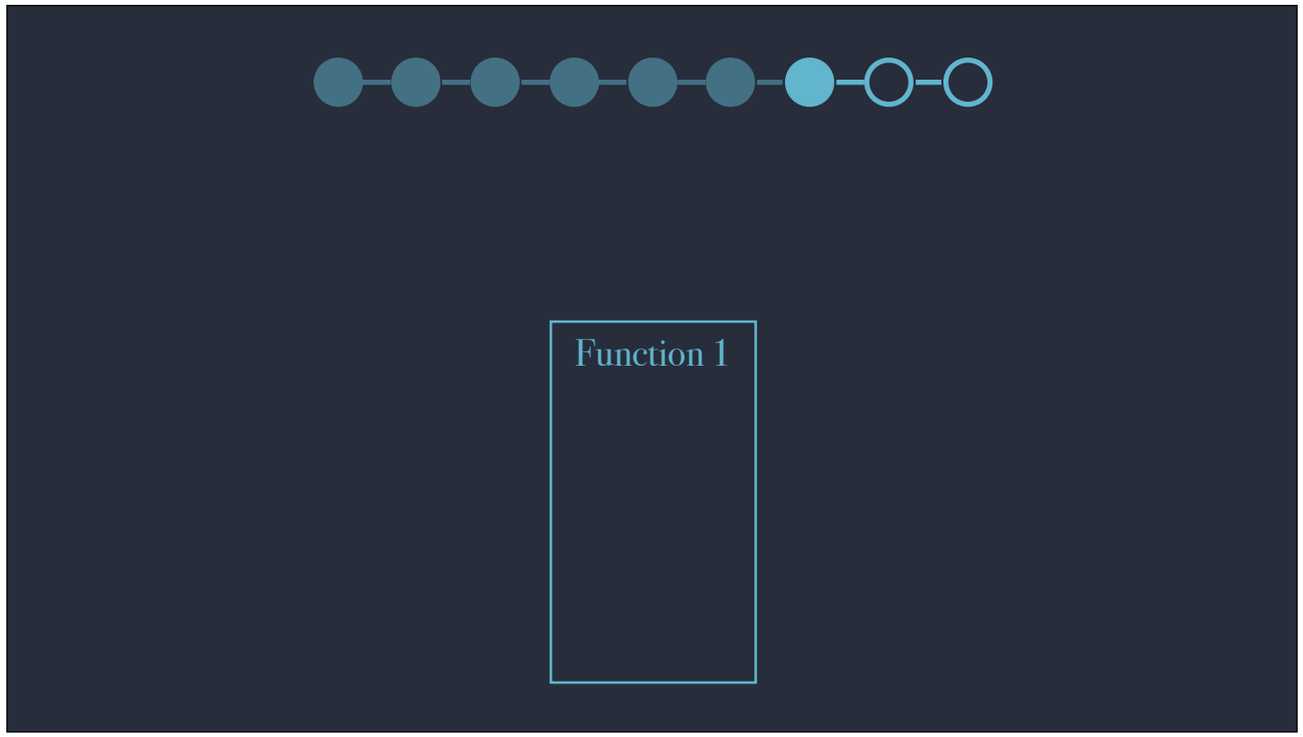
Readability goes beyond just reading the syntax. Victor calls for the ability to “decode the code.” You want students to see the bigger picture of the code they are writing. I’d like to return to Elm Play. As you mouse over the code provided, you will see what purpose the code serves. I’m trying to get students to see how the end product comes from the combination of several “ingredients” as it were.



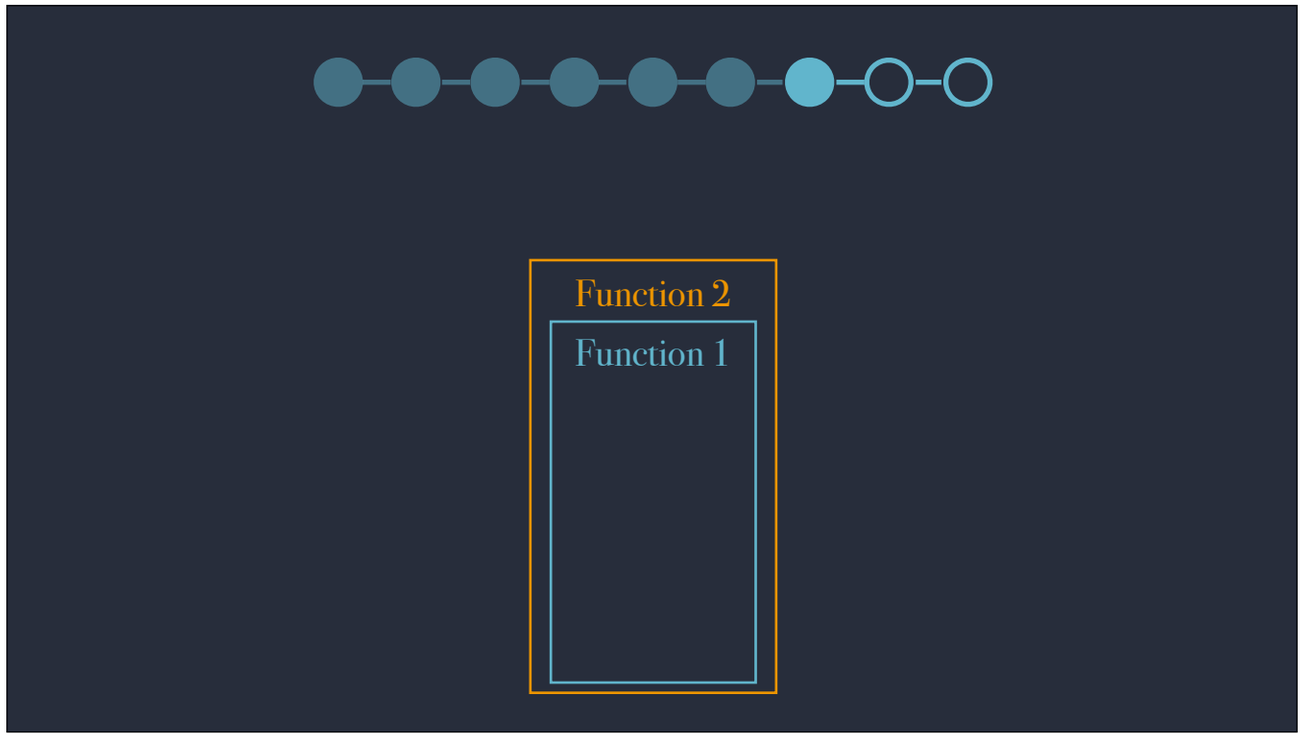
The 4th design principle is recomposition and decomposition. Basically, you are looking for small blocks of code that can be understood on their own and then combined with other blocks to form larger, more easily understood objects. If this doesn't sound new and cutting edge, it's because it isn't. Functional programming does exactly that. Everything is a function. And Lisp, a functional language, has been around since the 1950's.



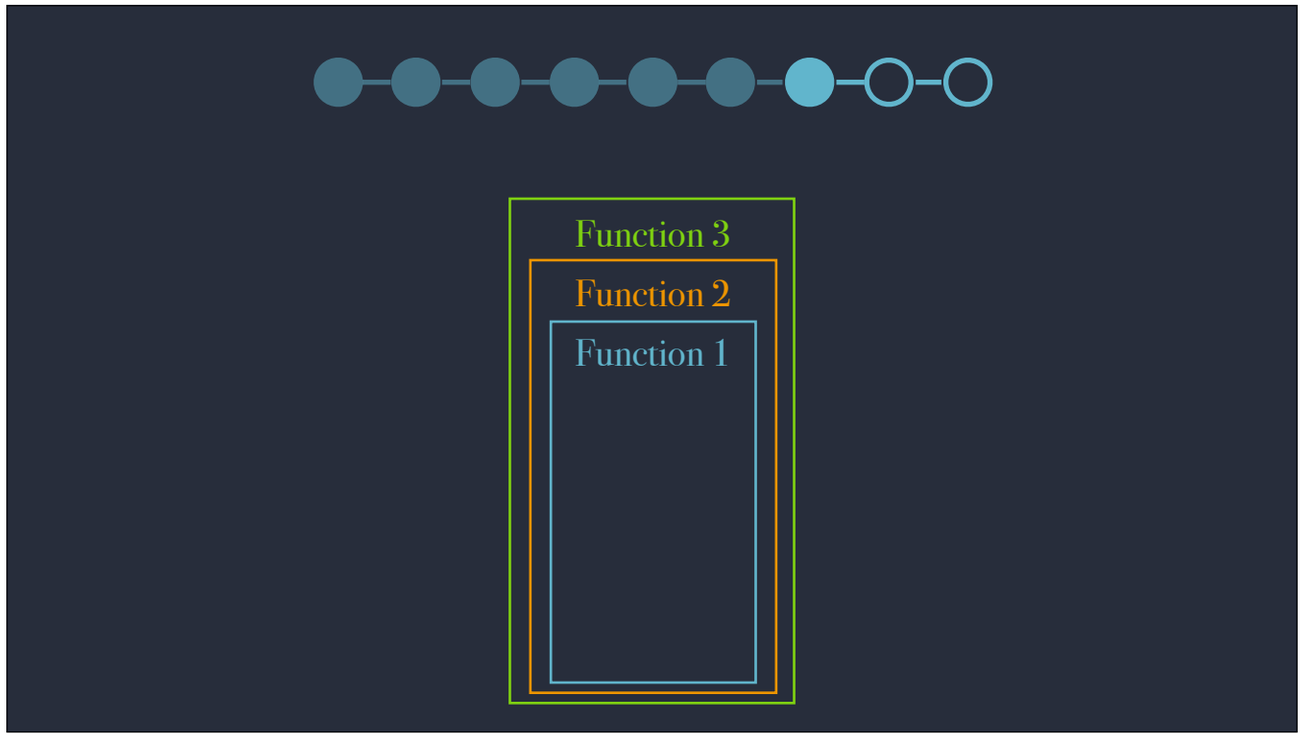
When you give a student an entire program, that is a lot of code for them to understand or debug. This is not beginner friendly.



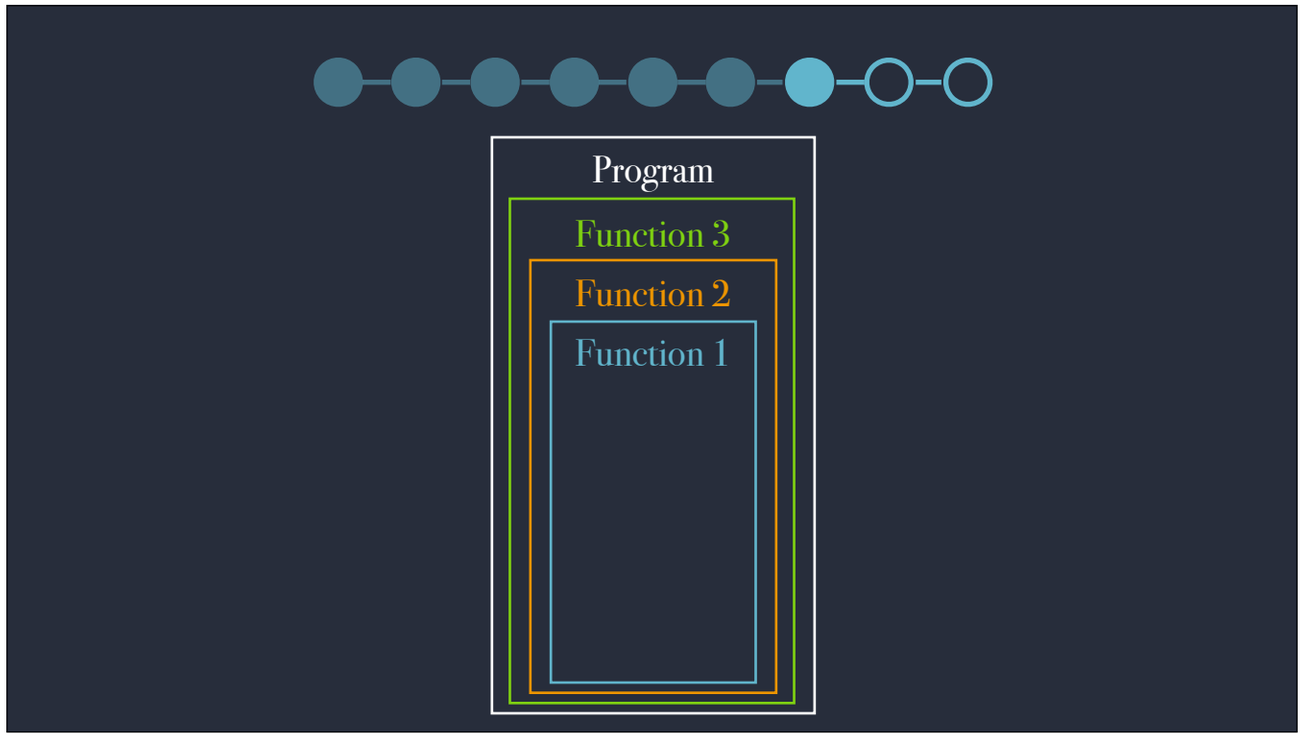
Instead, write a simple function that does one task. For the time being, don’t try to understand the program, just have the student focus on the function.



Functional languages have something called first class functions. This means that functions can be passed as an argument to another function. Once you understand function 1, understanding function 2 becomes easier.



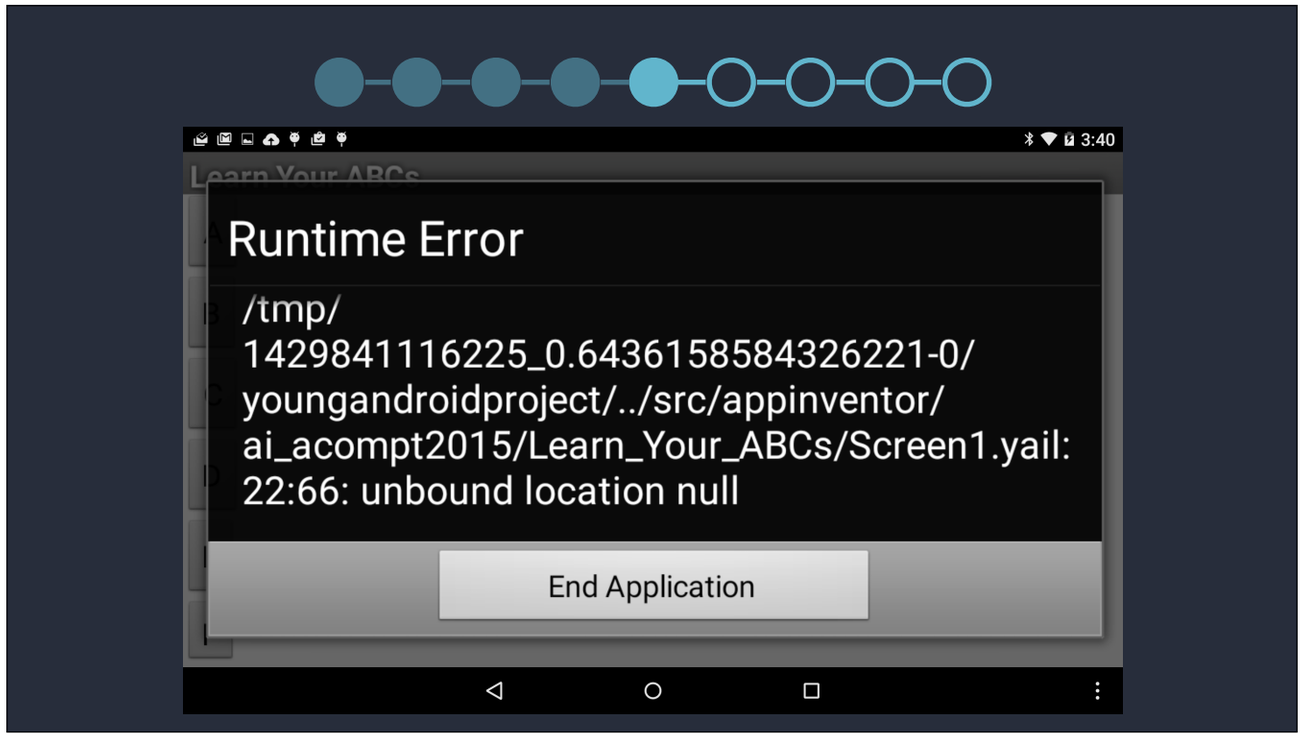
As you add more functions, you increase the complexity but you maintain a level of readability. Students can see how the functions come together to accomplish a task.



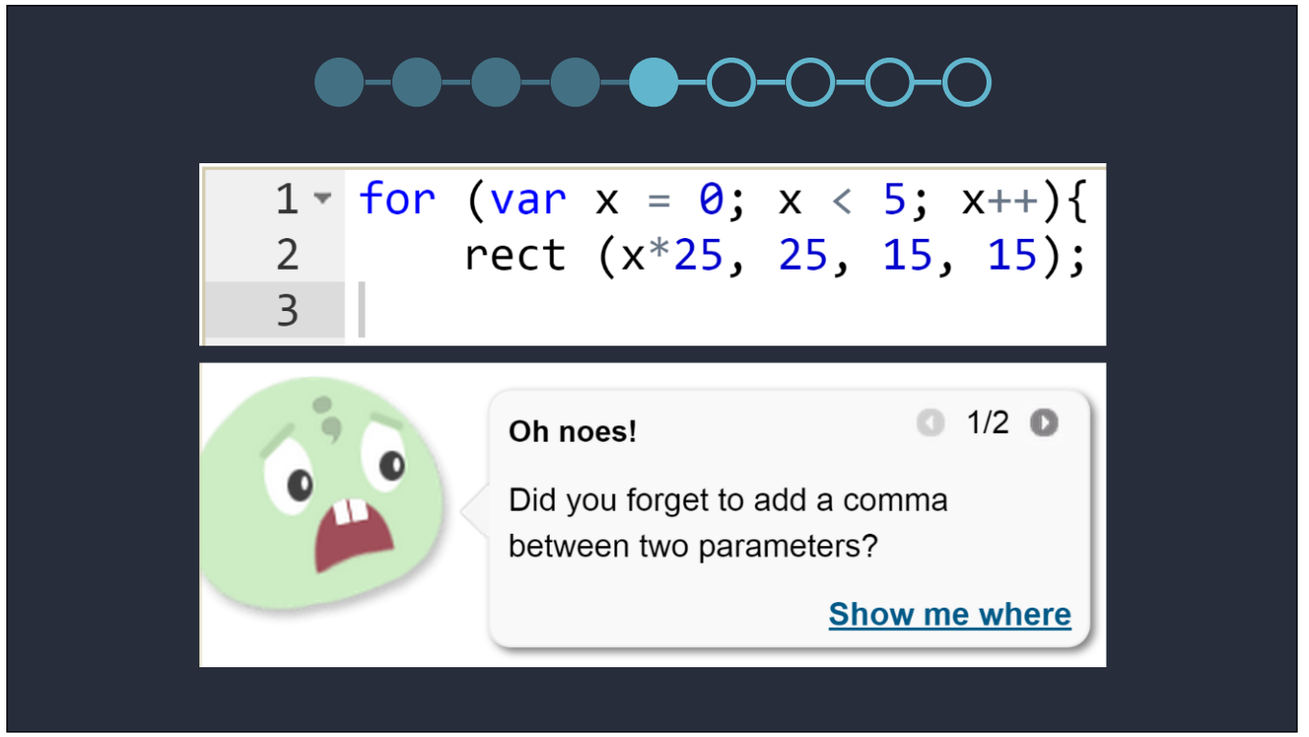
Now, the program is made up of a series of functions instead of one big blob of code. Functional programming allows for more readable code through the recomposition and decomposition of code through the use of functions.

The 5th design principle is helpful error messages. Error messages exist to inform the user of some sort of problem. But if the error message itself is so esoteric as to confuse the user, then you have just added another problem to the mix.

Last year, a 4th grade student wanted to build an app for his Android tablet. I told him I did not know how to do that, but I had heard of something called Android App Inventor. The idea is to create an easy way for beginners to write apps for Android devices. I invited him to work in the lab before school started. One day he said he needed help with a problem he was having. He did not know how to resolve the issue. I asked him to replicate the problem.

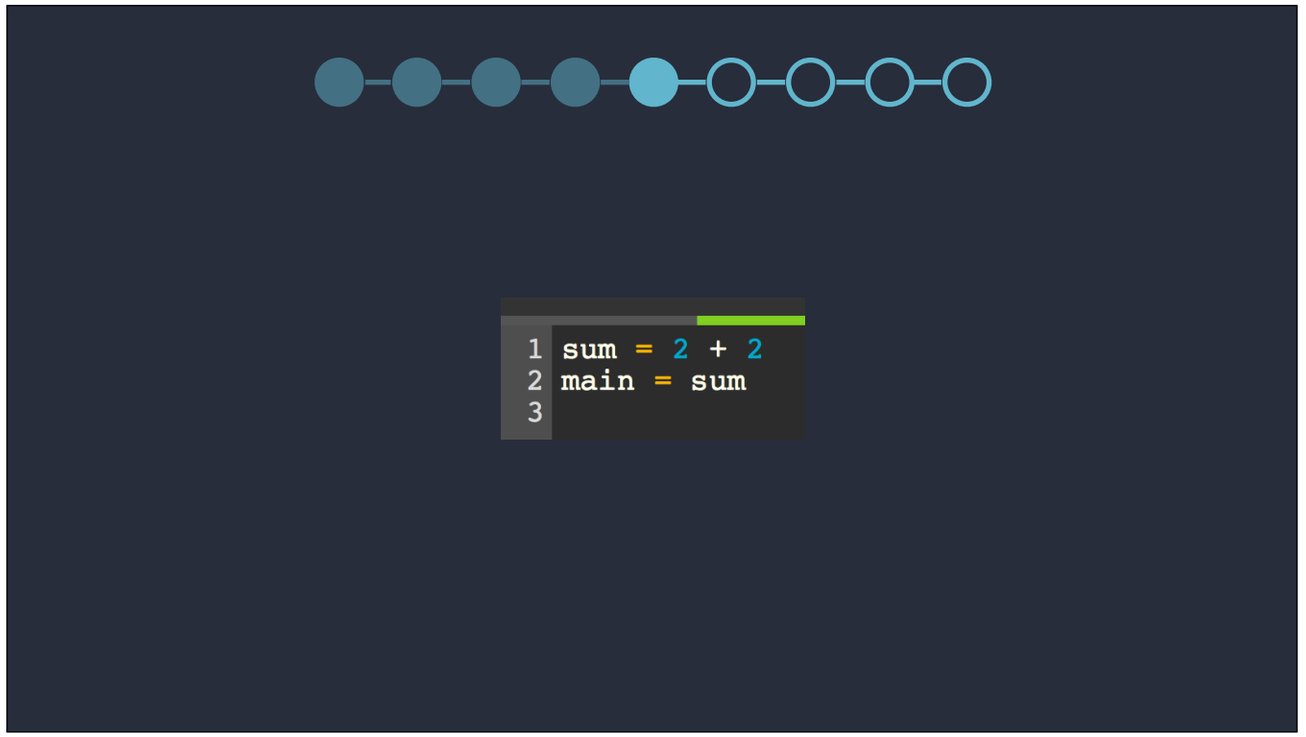


This is the error message he got. Unsurprisingly, he did not know what to do. So he stopped working on the app. That's a problem. Error messages should not hinder progress. Is it really reasonable to expect a 4th grade beginner to take to Stack Overflow, a mailing list, or an IRC channel? If we want beginners to learn, we have to do better than this.

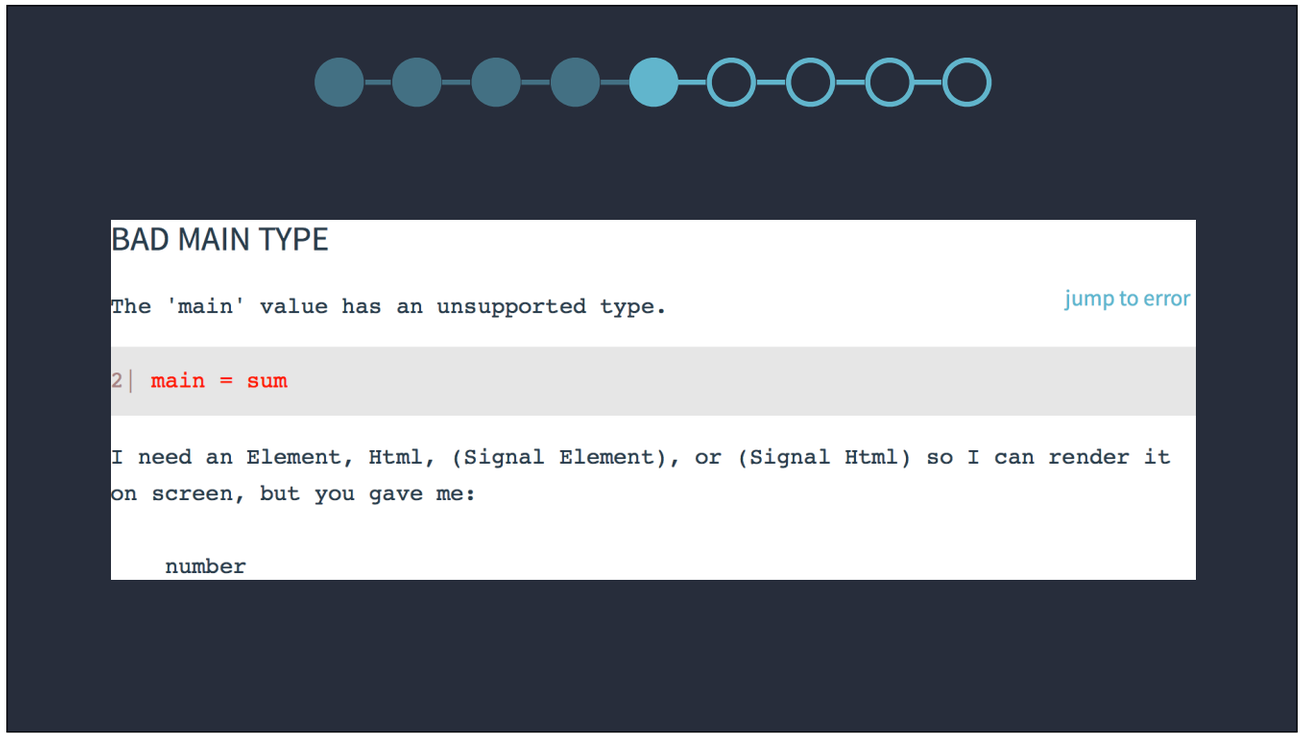


To be fair, error messages are hard. Here is some JS code on Khan Academy. The code above is very similar to something I did last year with my students. I saw this error message or similar error messages a lot. This is not to pick on Khan Academy. They are trying, and they are doing some brilliant things when it comes to getting kids to program.

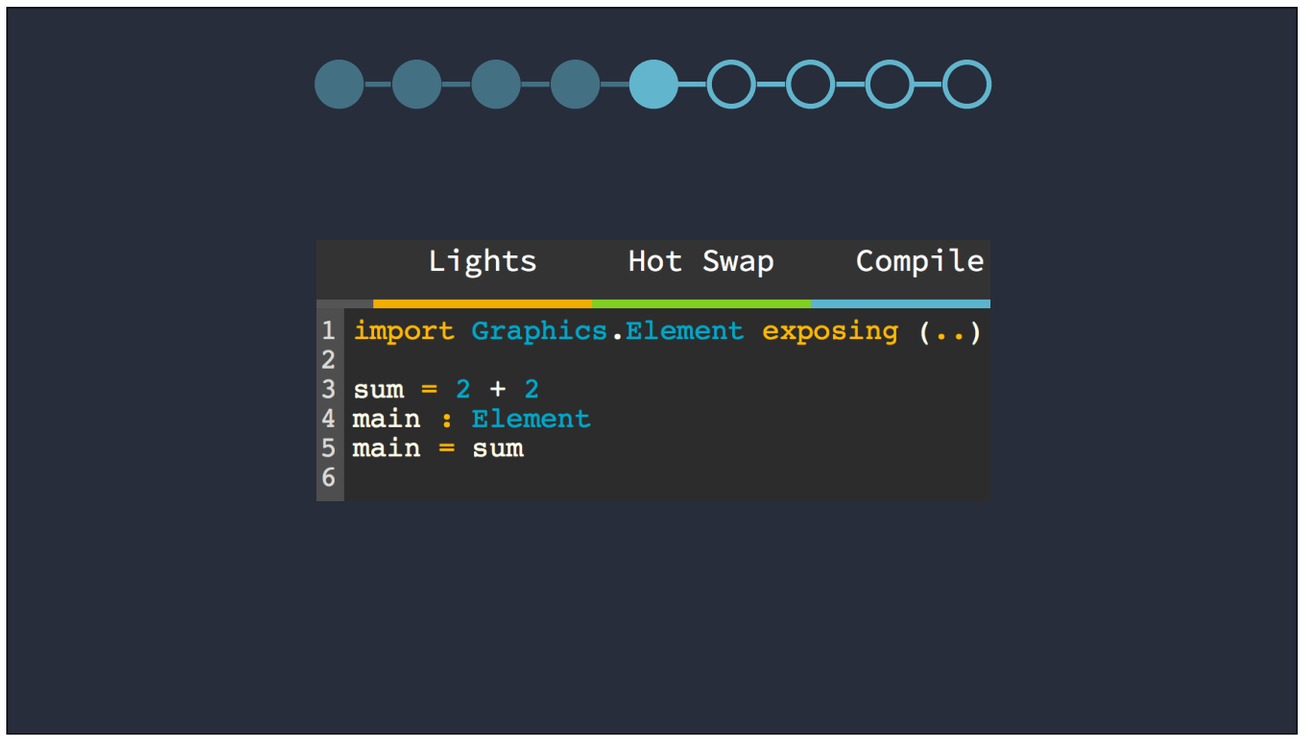
So when one of my students saw this message, they read it literally. They assumed that the problem was a missing comma. After all, why would the error message steer them in the wrong direction. In reality, the problem was a missing curly brace, not a comma.



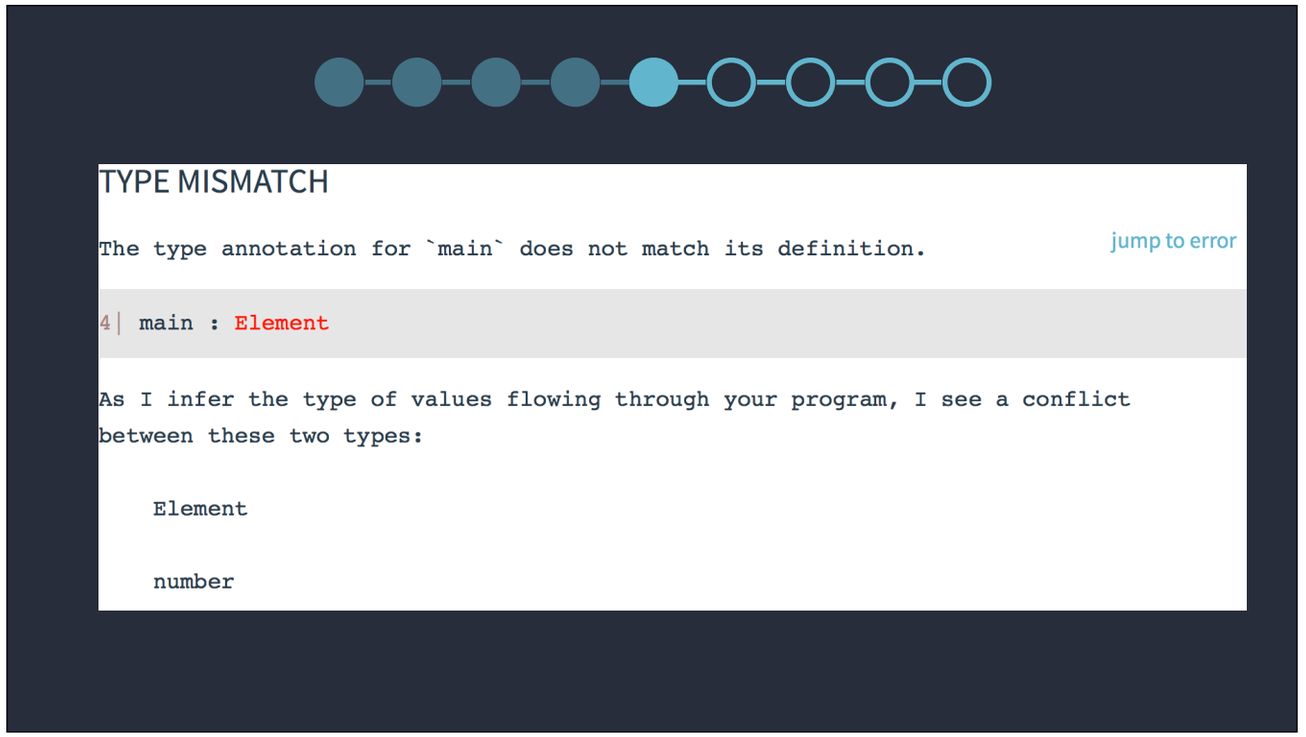
So, how does Elm address error messages? Well, with the recent 0.15 release, they have put a major emphasis on error messages. Here is some code. Now every Elm program has a “main.” So I have this variable called “sum” that has the value of 2 + 2. I give “main” the value of “sum” hoping that it would write it to the screen. But there’s an error.



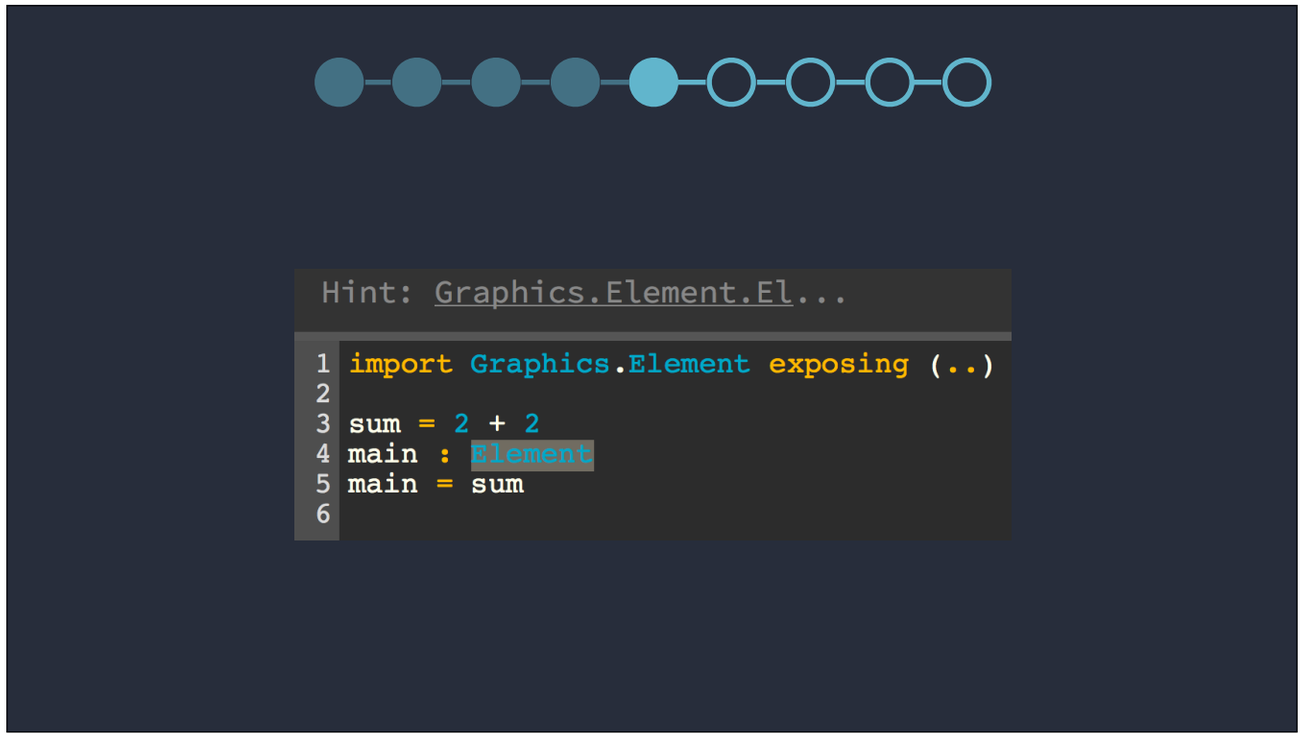
I like this error message. It is nice and verbose. It is very clear that “main” needs an element for it to work properly. Okay, so let’s add an element to the code.



I’ve imported the element library, and I gave “main” the type element. Surely it will work now.



No. Even though the problem is exactly the same as last time, the compiler gives me a different message. It states very plainly, “main” is of type element, but I am giving it a number. I need to fix this. Notice the “jump to error” in the top right. That’s a link.

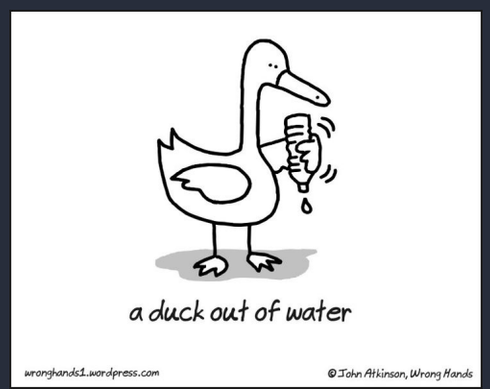


When I click on it, I see the word “Element” in my code highlighted. Up top where it says hint, that’s a link to the Graphics.Element API page. By clicking that, i can read up on elements and how to transform a number to an element so it can be written to the screen.

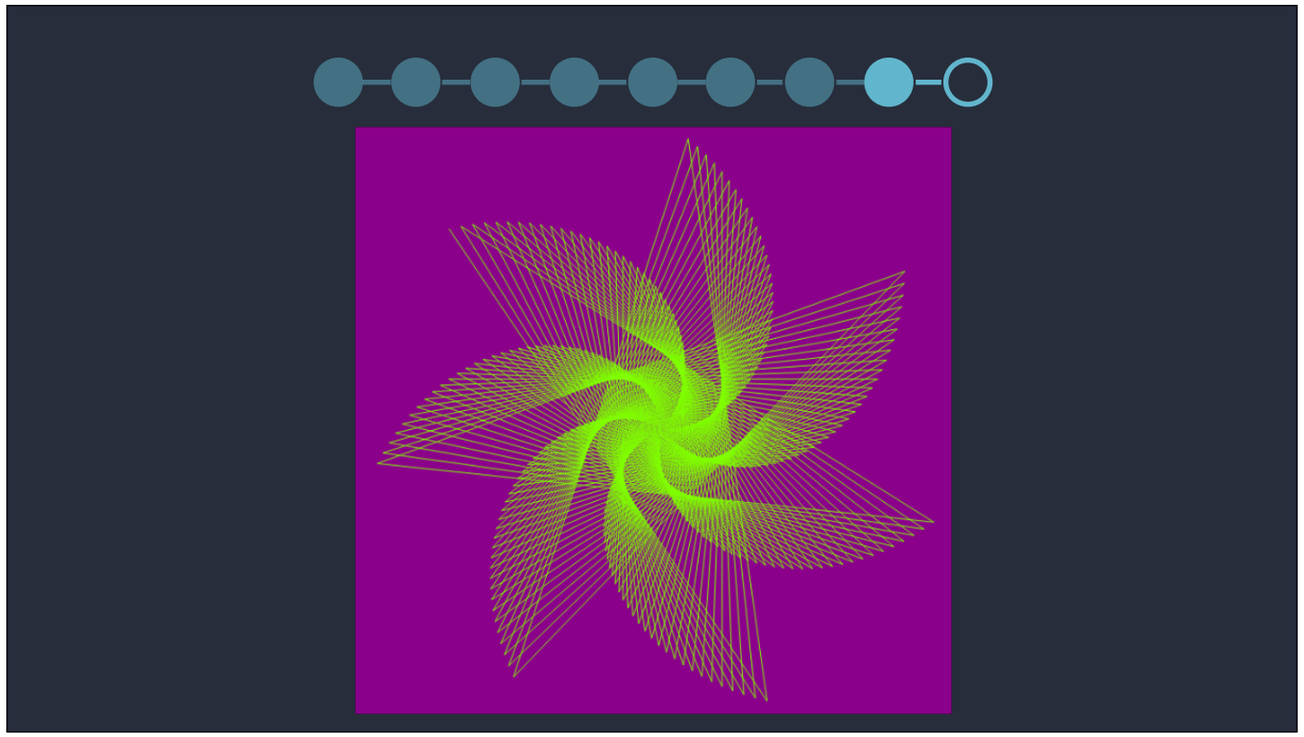
Error messages are important, especially to beginners. They are going to make tons of mistakes. We need to make sure that error messages serve their intended purpose. They should help students resolve the problems in their code, not create more problems.

Metaphor

Removing abstraction



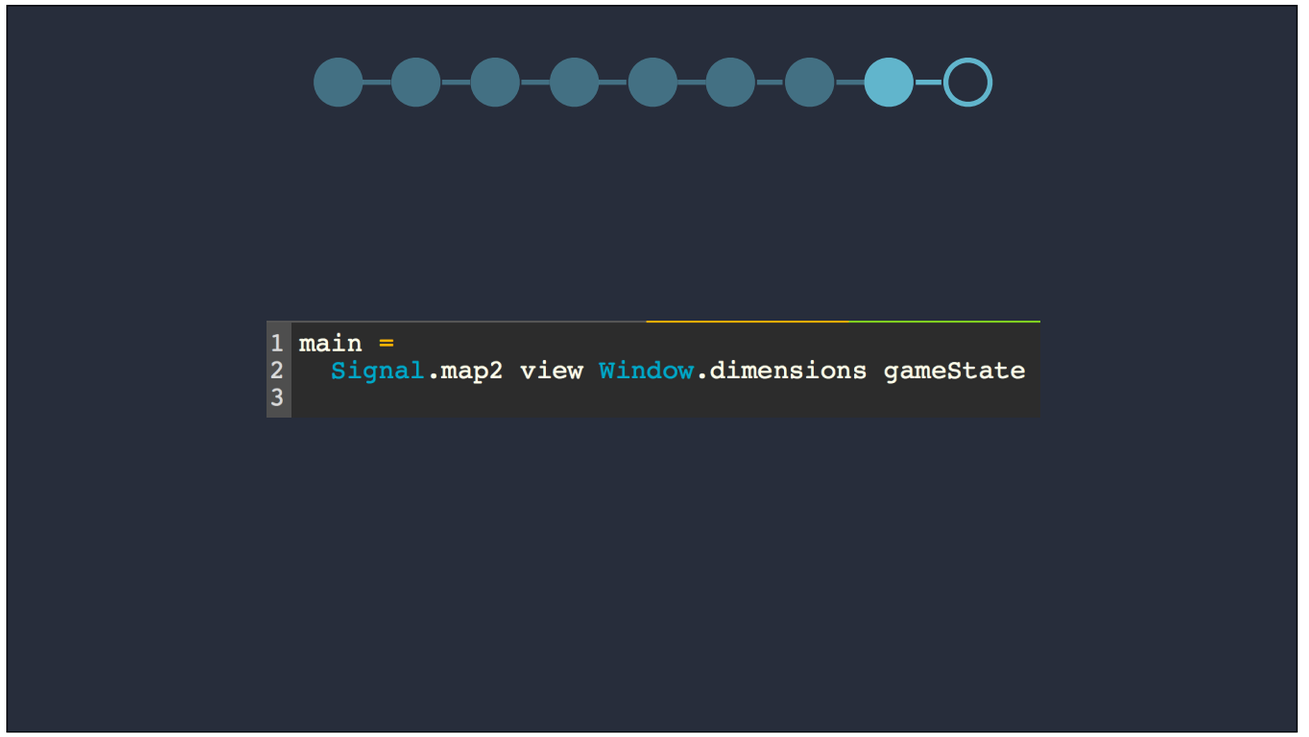
Programming is an incredibly abstract task. When I teach 6th graders, they are just beginning to dip their toes into the waters of abstract ideas. As such, they need help when writing code. Seymour Papert invented the Logo programming language, he came up with a powerful metaphor, that of the turtle. The 6th design principle is having a metaphor.



When you told the turtle to go forward or backward, turn to the left or to the right, it would draw a line on the screen according to your commands. In fact, this was created by one of my students.

Last year, I invited a handful of 4th grade boys to the lab before school started to learn some programming. We used a program called Kids Ruby. Kids Ruby implemented Papert's turtle but with Ruby syntax. When they made a mistake, I'd tell the student to stand up and pretend they are the turtle. Every so often you'd see somebody get up and start walking around the lab. Then it would hit them. By walking around, they would see where the mistake in their code was. This, of course, goes back to the first principle Constructionism. The kids didn't have to rely on me to tell them how to fix the problem. The could do it themselves (thus building knowledge) through the use of the metaphor.

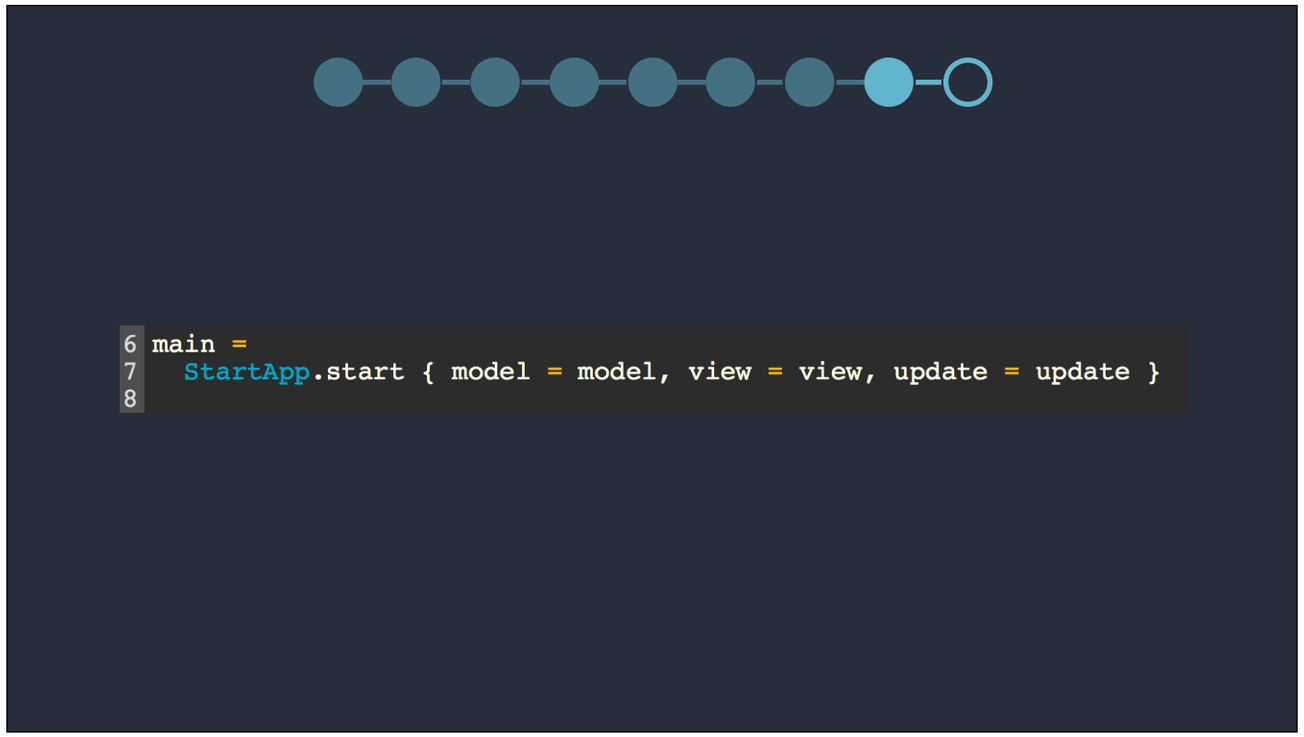
The turtle metaphor, despite its power, is quite limited. It is not possible to do tasks like string manipulation with the turtle metaphor. The trick is to find a metaphor that retains its power while allowing for greater flexibility.



```
main =
  Signal.map2 view Window.dimensions gameState

```

When I was making my capstone project, Elm did not have a metaphor. So this is what my “main” looked like.
Explain what “map” does, apologize for the poor definition
Explain map2 and map3 and the limits of map
Before the metaphor, “main” can be a tricky spot
To make our lives easier, there is now a metaphor in Elm



```
6 main =
7   StartApp.start { model = model, view = view, update = update }
8
```

The metaphor is called StartApp. When you use the metaphor, your “main” will always look this. StartApp has three characteristics: model, view, and update. You set the value of these characteristics to functions for model, view, and update that you write. There is no more worrying about using the correct “map” or worrying about mapping the right signal to the proper function. StartApp takes care of this. All you have to worry about is the model (what does the data structure look like), the view (what does the output on the screen look like), and update (how does the model get updated, which in turn changes the view).

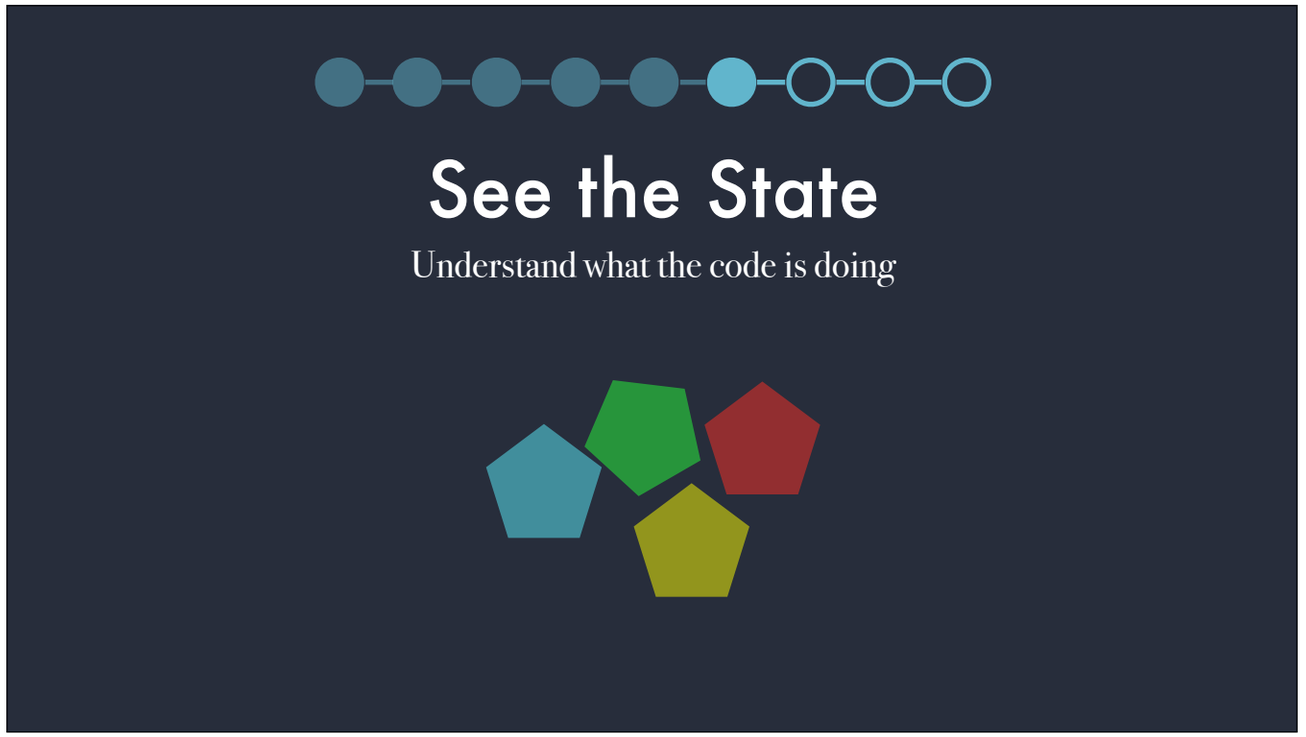
```elm
model = 0


view address model =
  div []
    [ button [ onClick address Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick address Increment ] [ text "+" ]
    ]


type Action = Increment | Decrement


update action model =
  case action of
    Increment -> model + 1
    Decrement -> model - 1
```

Here are the model, view, and update functions that go with StartApp. The idea is that every Elm program will have the same “main” and that every Elm program will have a model, view, and update. There should be a consistent architecture.



The 7th design principle is “See the State.” As previously mentioned, compiler error messages should help users fix their code. But what about when the program compiles and the output is not what the user expects? Here, the tools are a lot more primitive. You could always write the value of a variable to the console. But all you get is a stream of integers let’s say filling up the console. This is only useful if you can take those numbers and give them context. For example, let’s say a variable represents gravity. You then have to be able to make the relationship between the players in your video game and how gravity affects their movement. Beginners have a hard enough time understanding the code they write, why overload them with the task of creating a mental model of what the output should be? Instead, users should be able to see how altering the code directly affects the output. Users should be able to see the state of their program. Before writing Learnable Programming, Bret Victor gave a talk entitled Learning on Principle. Toward the end, Victor posited a code editor that would do exactly that. He made a working prototype, and he blew away the audience. In fact, he blew the minds of anybody who watched that video. Among those people were some affiliated with the Elm programming language. They decided to integrate Victor’s idea into their project. This is what they made.

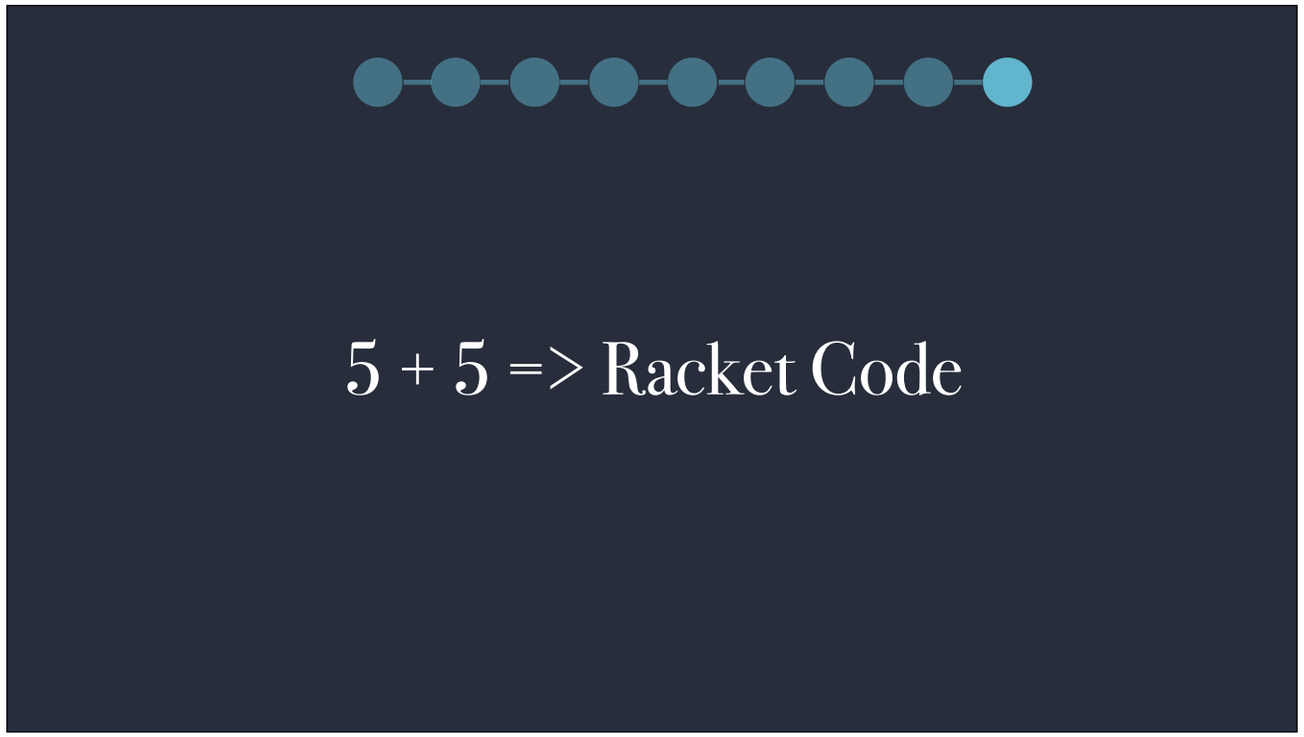
Demo

Elm's time traveling debugger

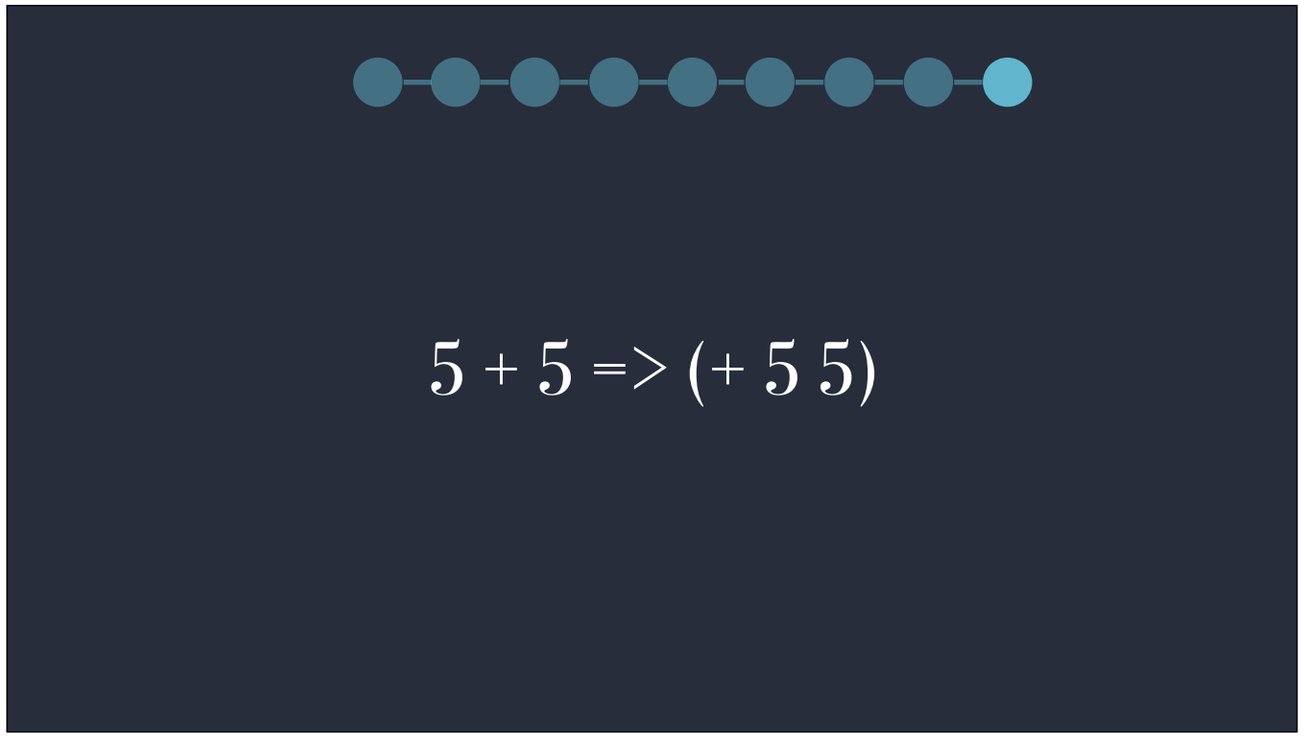
The 8th and final design principle is “Learning Tools.” Students need some way of resolving the problems that arise when programming. These tools are not features, per se, of the language or development environment. Instead they are a methodology.

Story about JS classes slowing to a crawl. Embarrassed. Giving answers does not conform to Constructionism. They need learning tools.

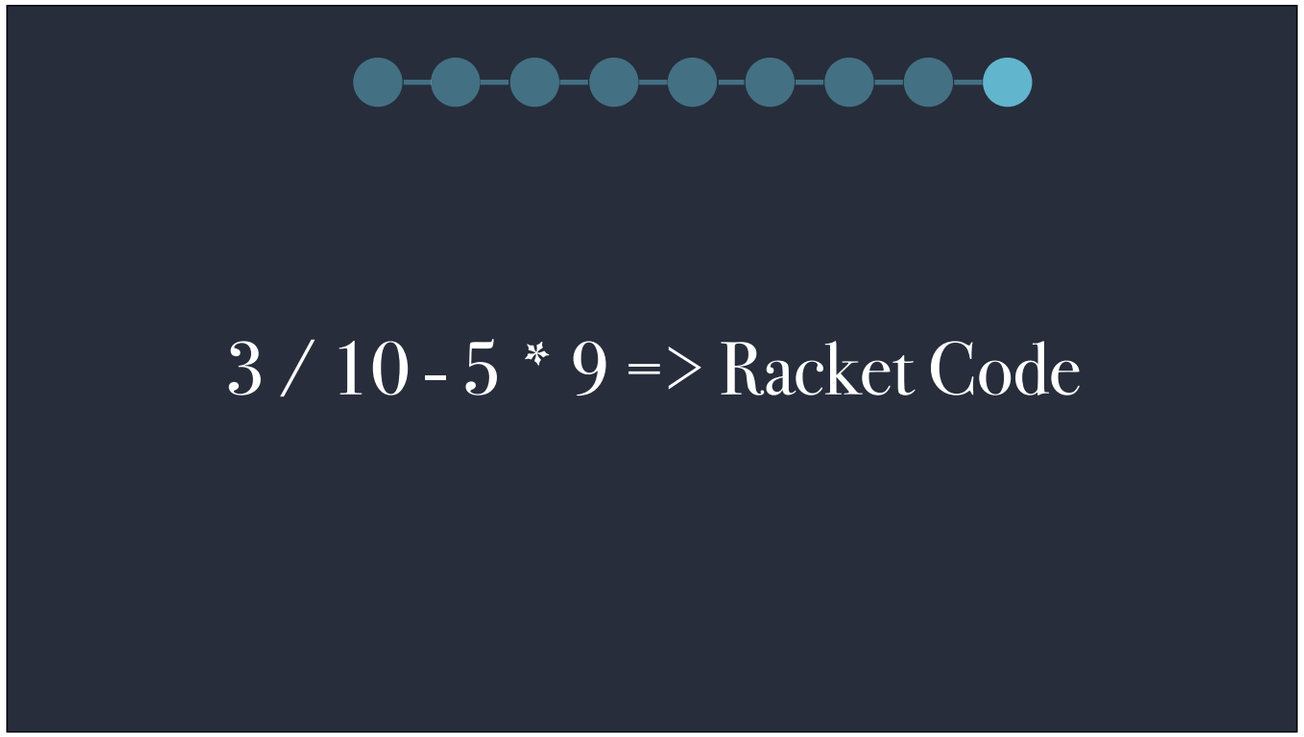
This summer I had the great fortune of attending a professional development session by a program called Bootstrap. They have been teaching coding to high school and middle school students for over ten years. Their curriculum has two wonderful learning tools that I would like to share with you.



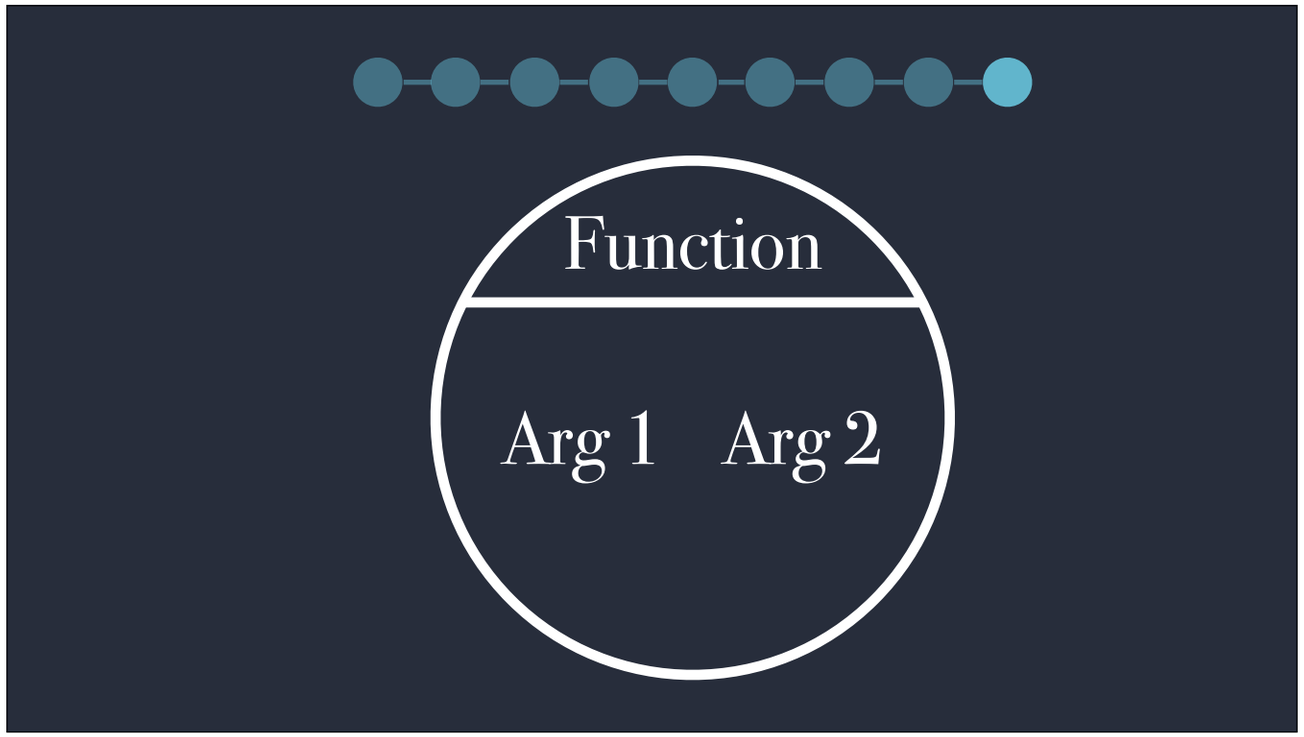
The first tool helps students write code in Racket, the language used in the Bootstrap curriculum. Since Racket is a Lisp, you have prefix operators and lots of parenthesizes. By most accounts, the Lisp family of languages are weird.



With something simple like 5 + 5, you can just explain how you write the problem in Racket. Start with an open parenthesis, put the function (after all, everything even math is a function), next write the two arguments, and close the code with a parenthesis. That’s not so bad.

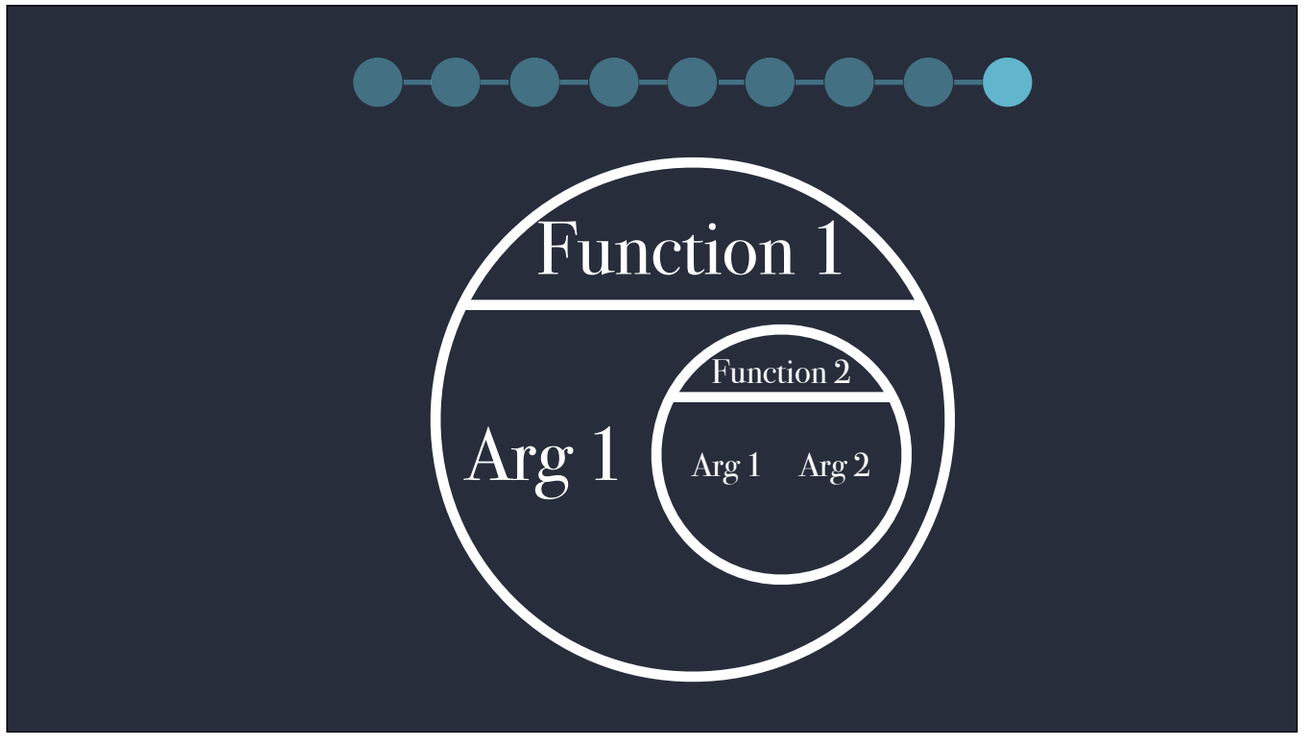


But what happens when the math gets a bit harder. Suddenly there’s no simple explanation. So the Bootstrap team came up with the idea of the circles of evaluation, a visual spatial tool to help kids write code in Racket.

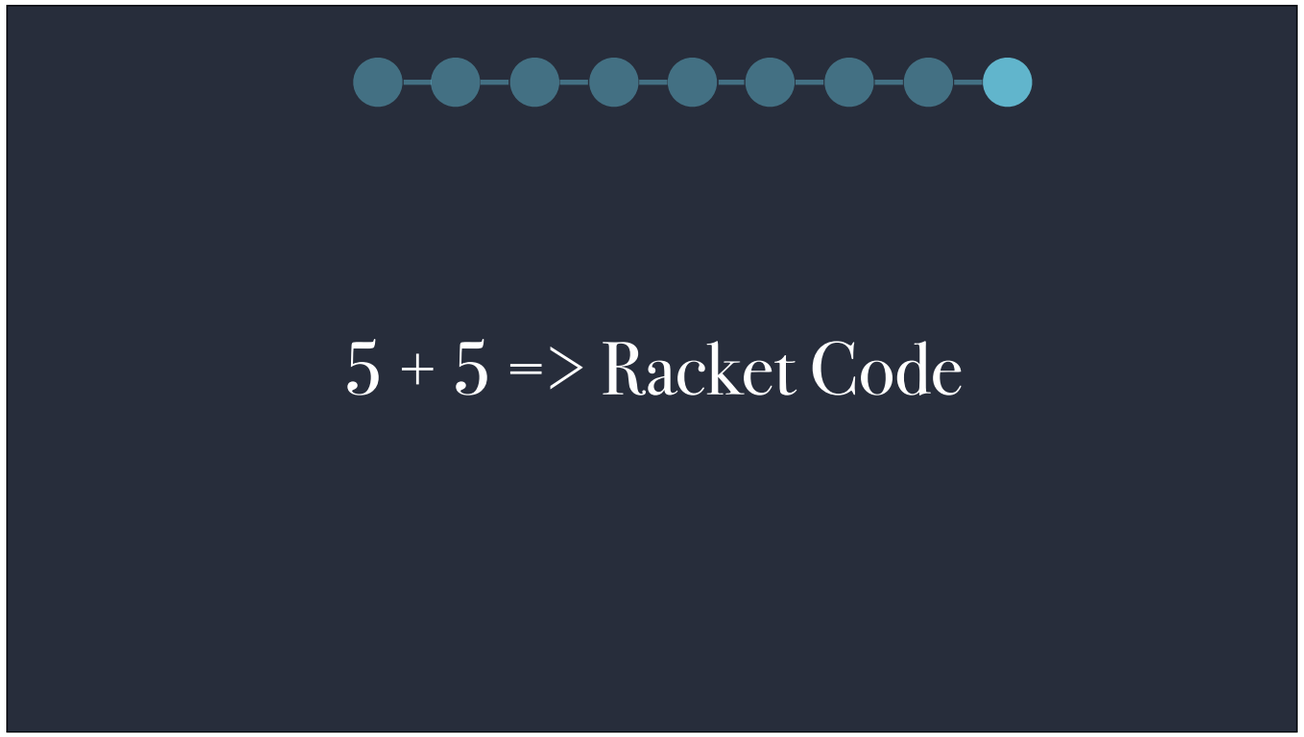


Here’s the model. Draw a circle with a line going across toward the top. In this space goes the function. In our previous examples, the mathematical operators are the functions. Below, from left to right, go the arguments.

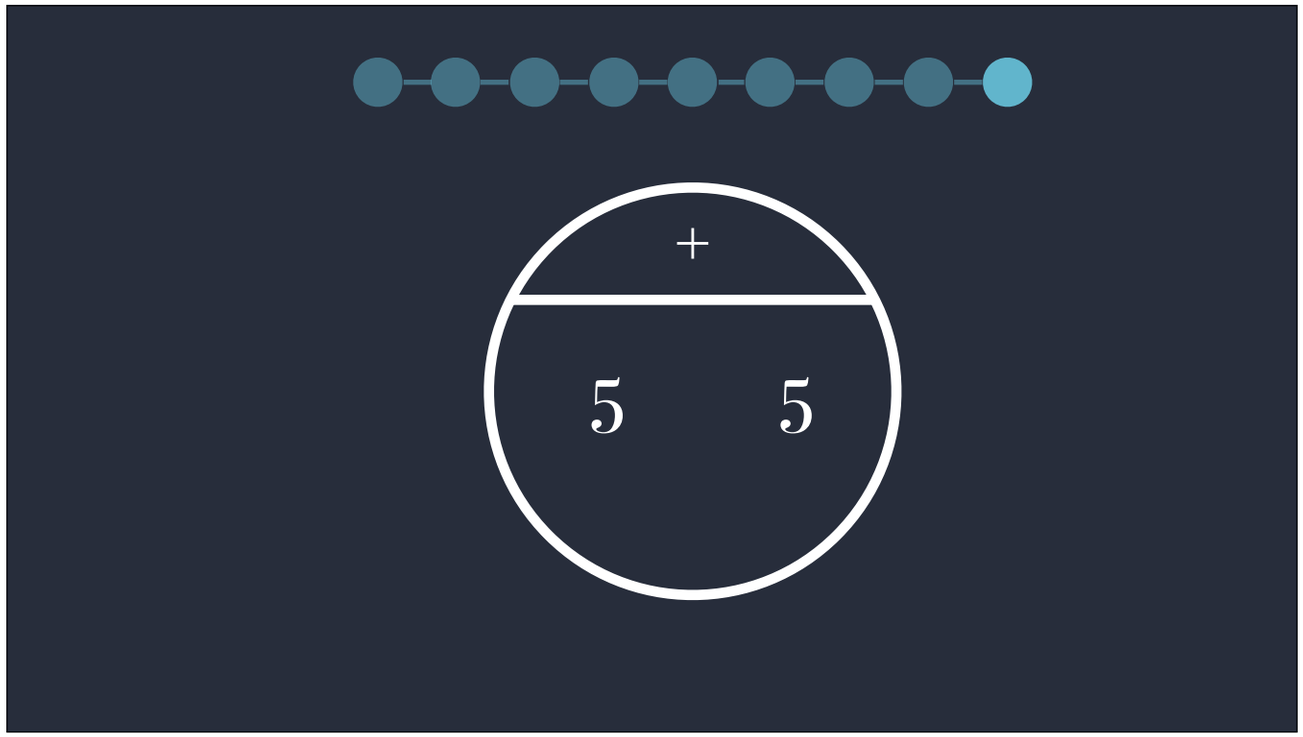
But what if you have those first class functions we talked about in the decomposition/recomposition design principle?



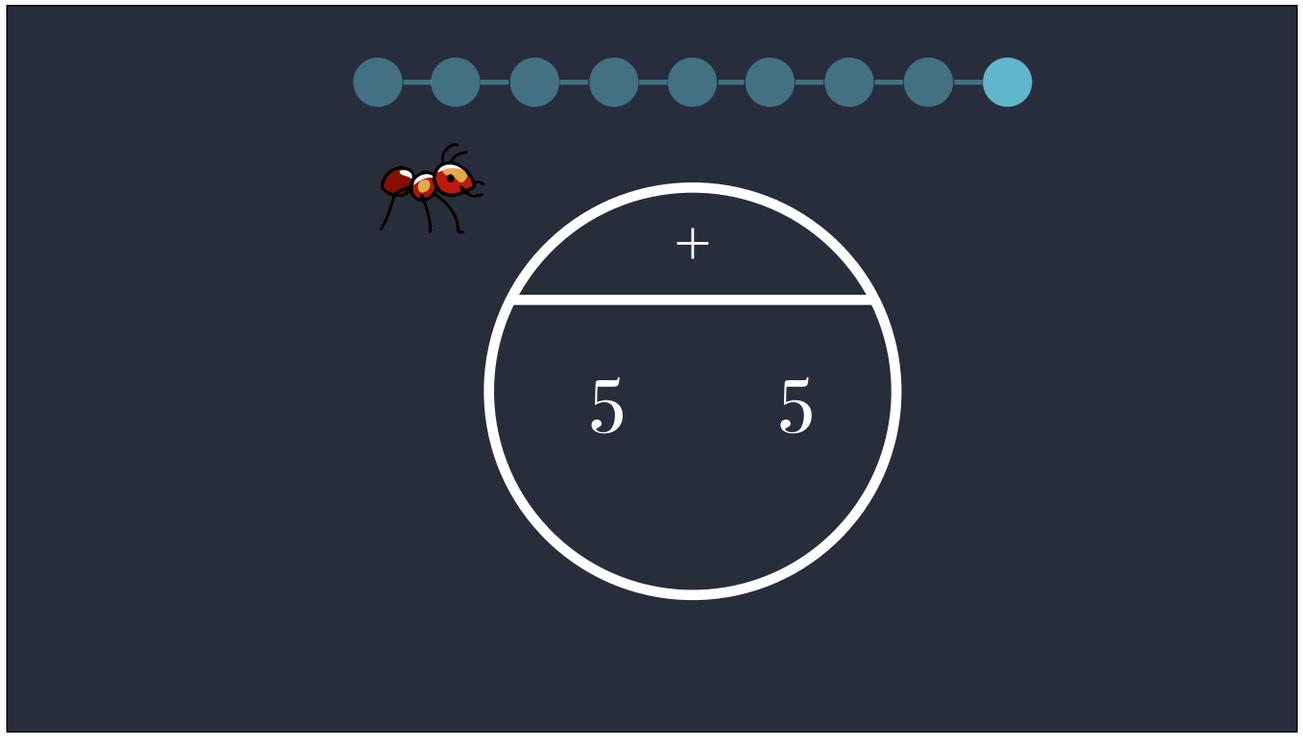
Well, you get something like this. Function 2 is an argument for function 1, and it is drawn up with its own circle of evaluation.



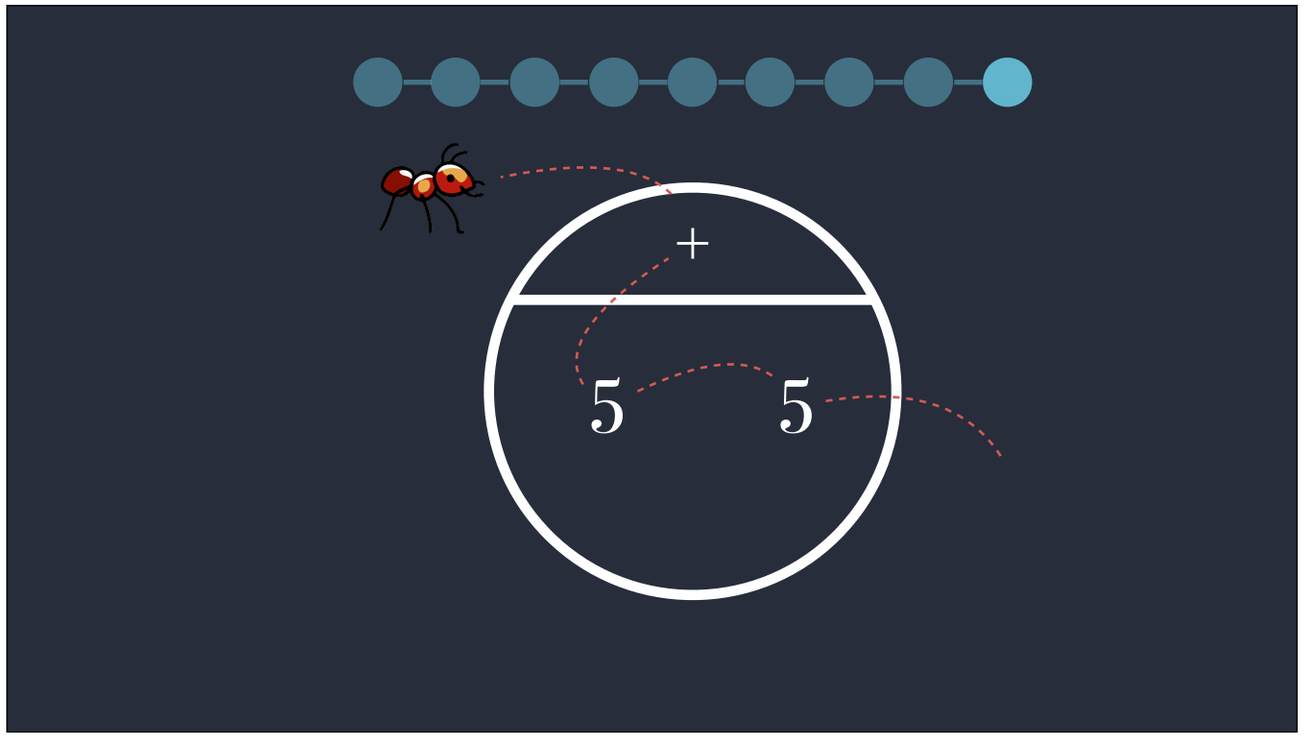
Okay, let’s go back to the easy math problem and rewrite it with a circle of evaluation.



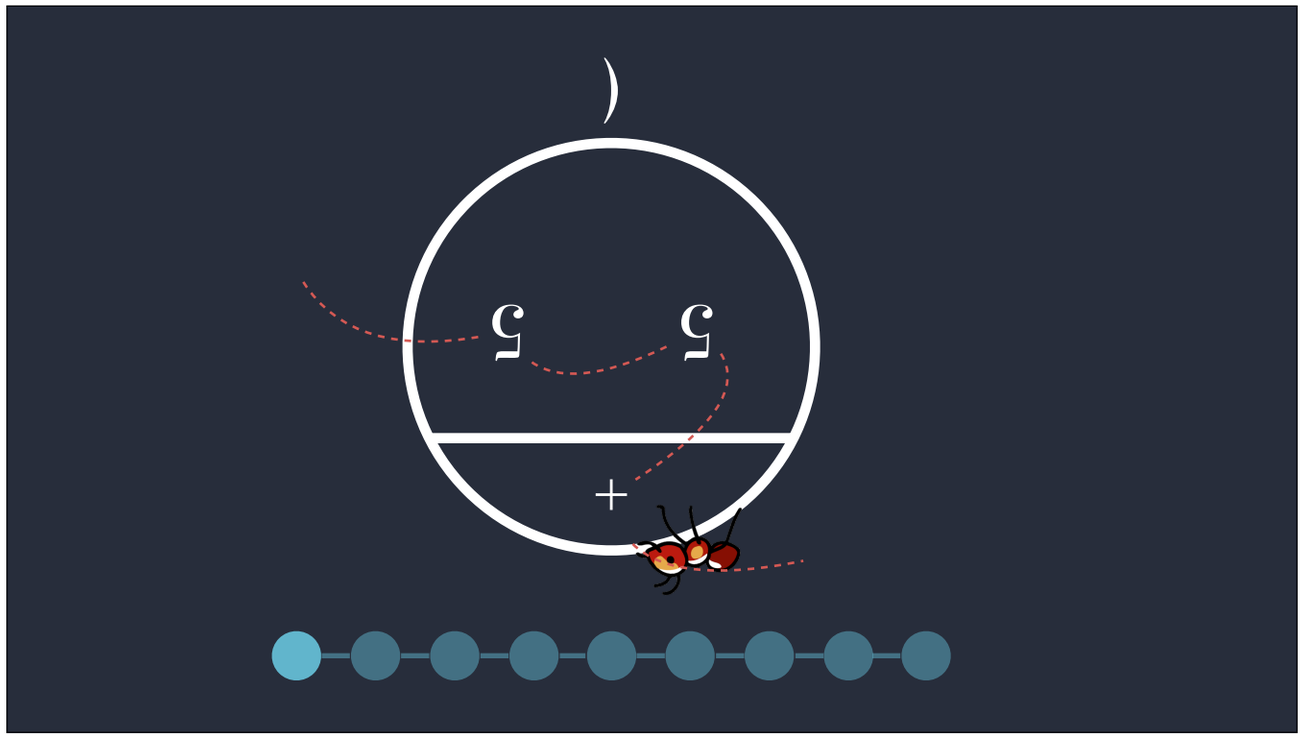
This is what we get. But this still doesn’t look like Racket code.

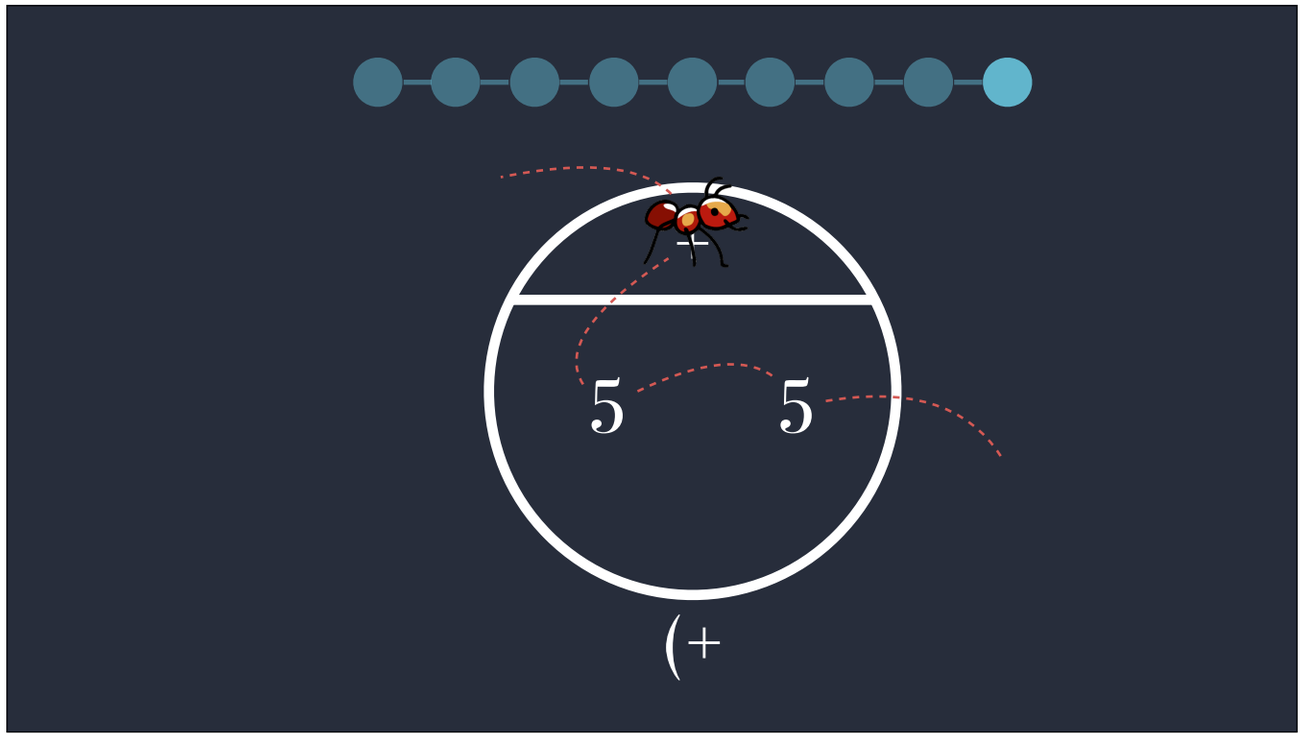


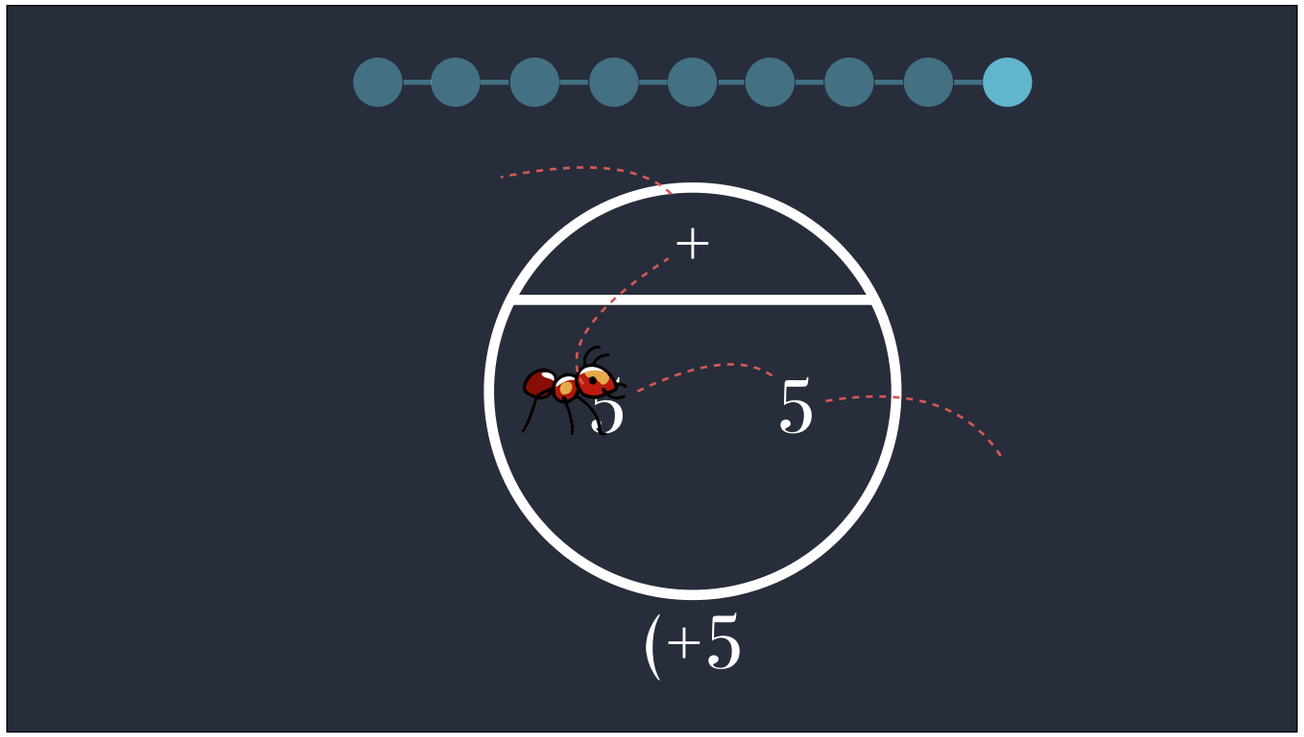
What Bootstrap does is ask the students to pretend there’s an ant.

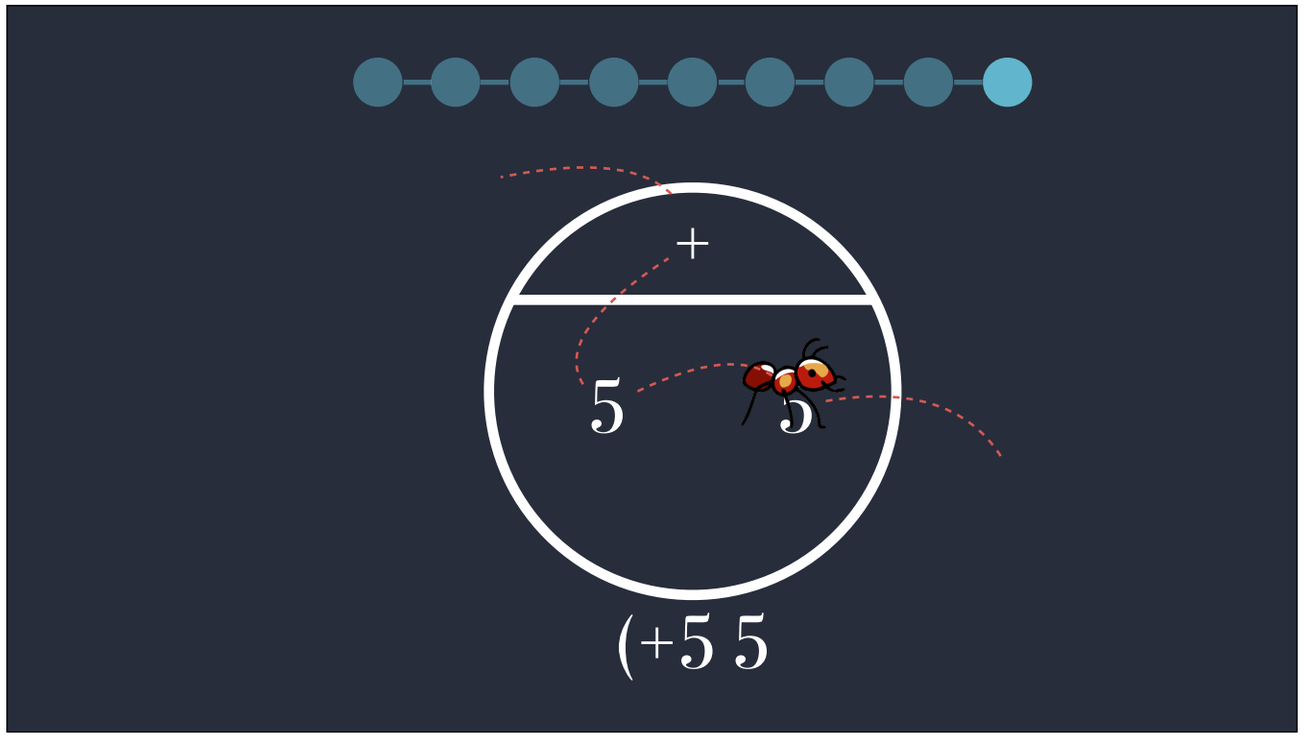


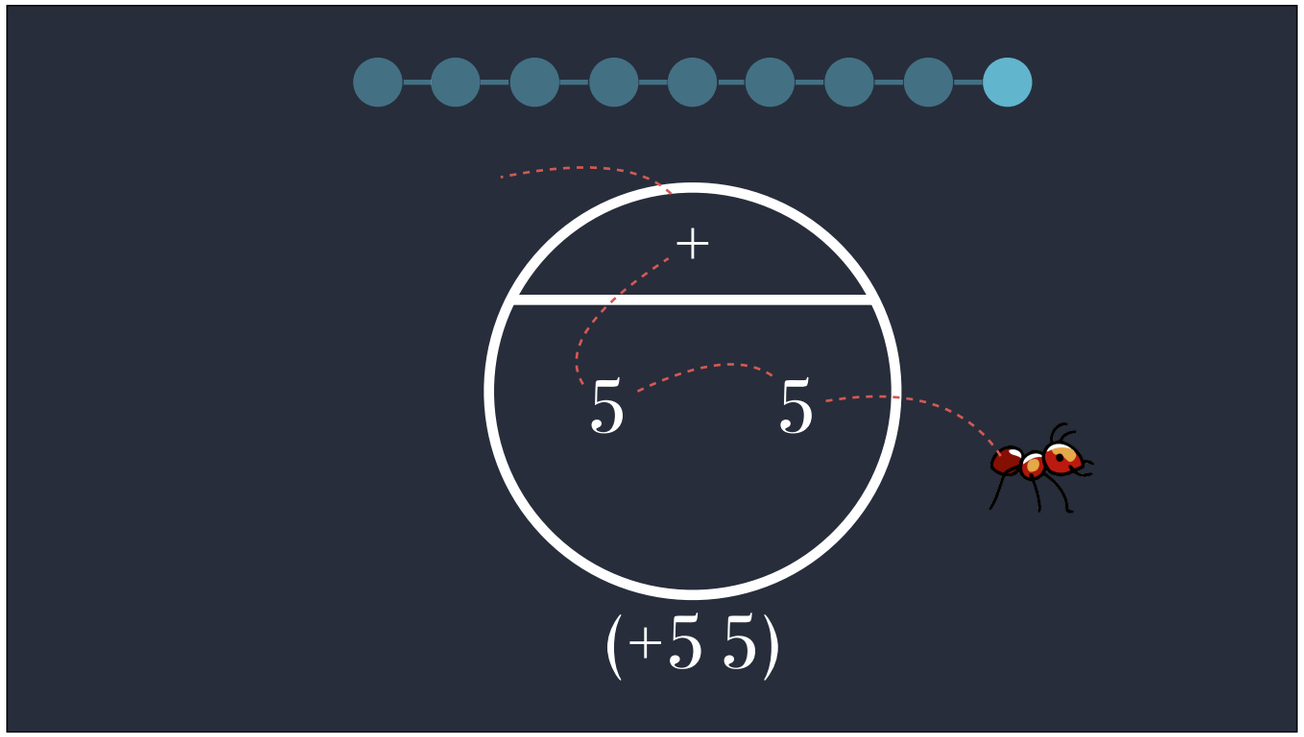
The ant is going to cross into the circle of evaluation, starting with the function up top. When the ant crosses the circle, it breaks the circle into two (use hands). You see that? When you break the circle into two, it kind of looks like parenthesis. That’s your clue to type a parenthesis. After breaking the circle, the ant first encounters the function, so type that out. Then the ant passes down below and goes from left to right. Type out the arguments as you go. When the ant leaves the circle, you close with another parenthesis. Let’s see that in action.

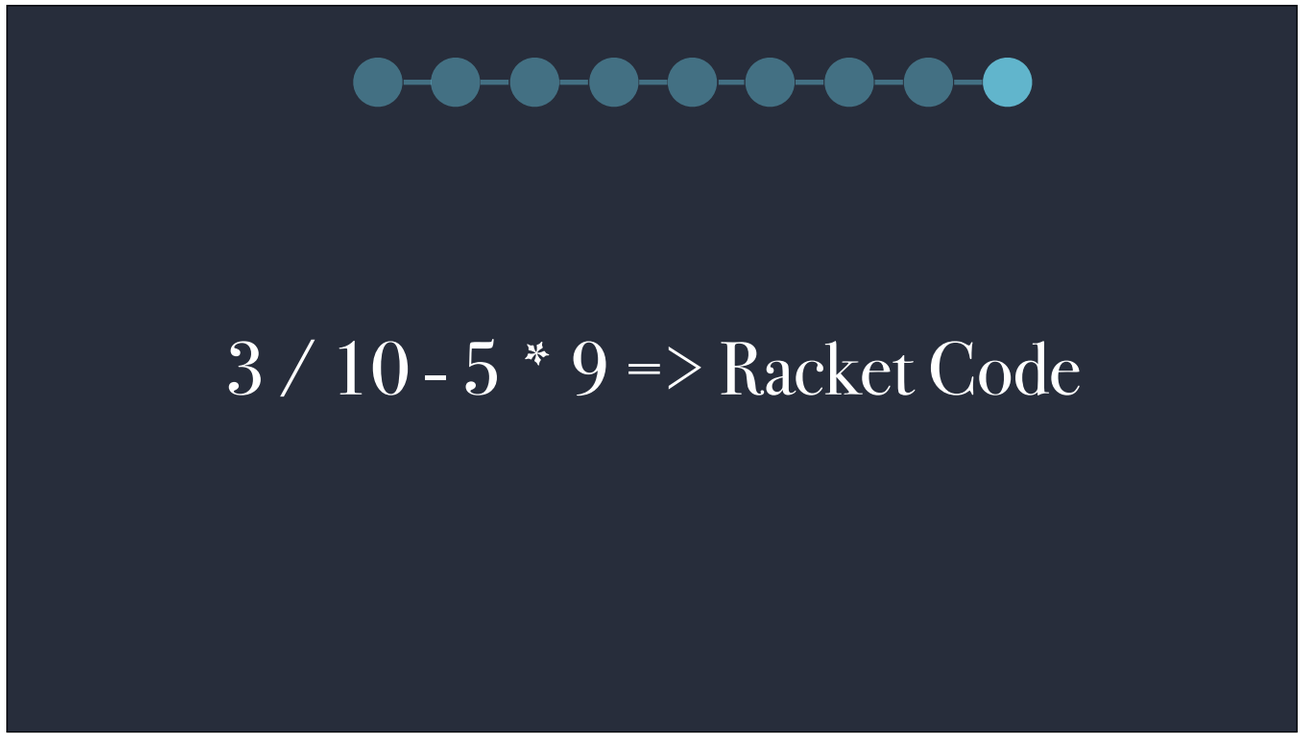

(
5
5
+


+
5
5
(+

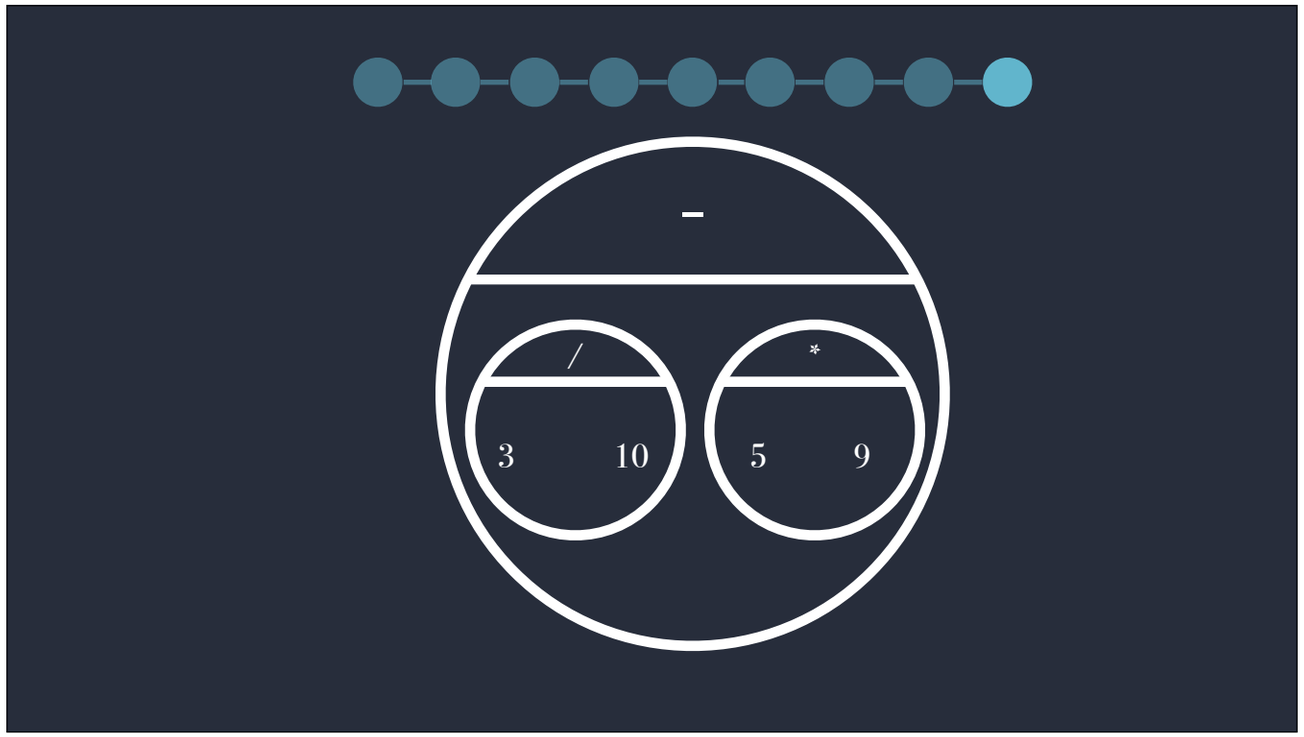

+
5
5
(+5


+
5
5
(+5 5

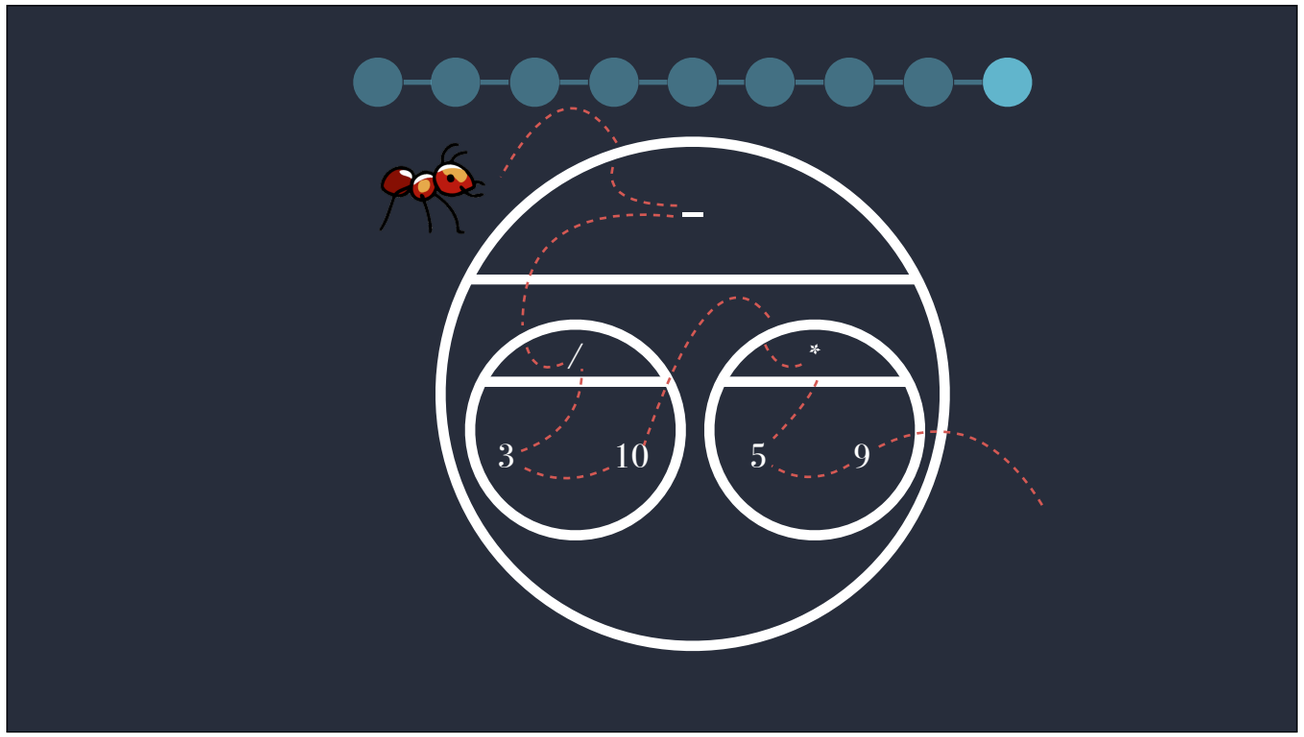

+
5
5
(+5 5)



Now what about the harder example? Let’s transform the math into the circles of evaluation.

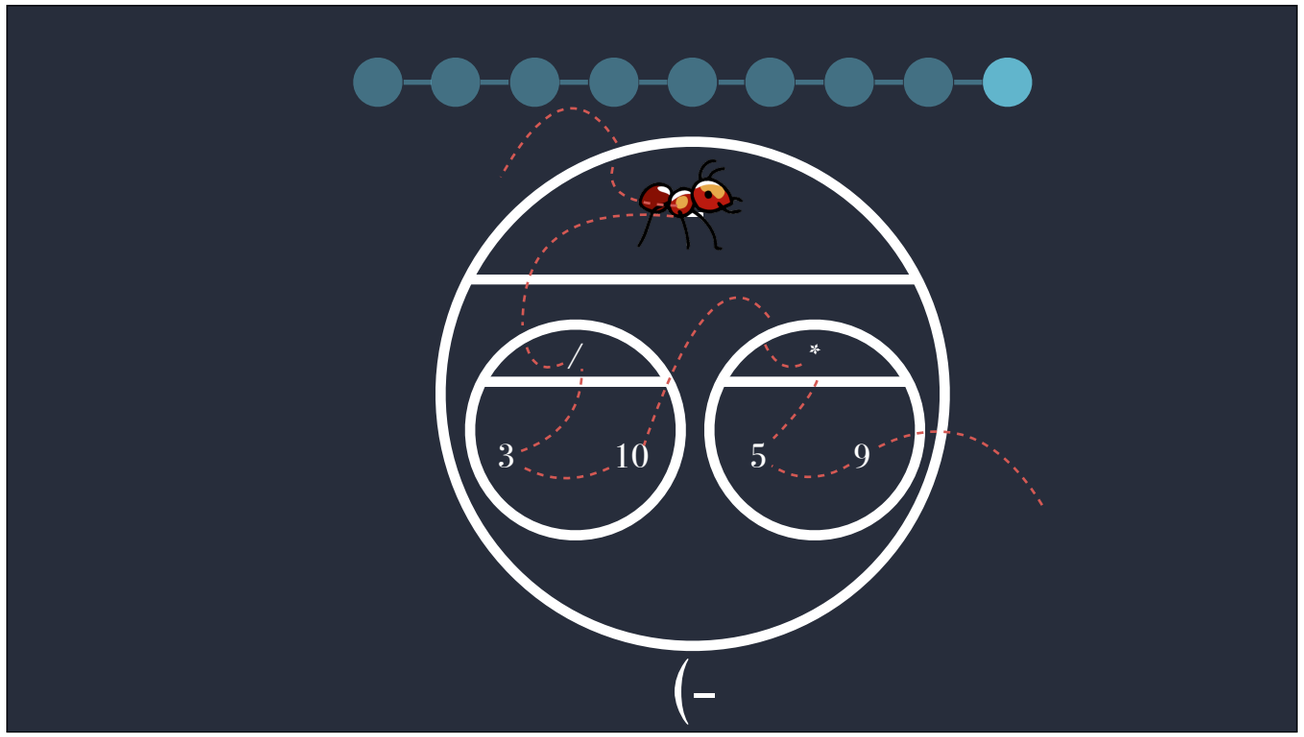


You get something like this. Admittedly, you have to understand order of operations to get to here. But, the Bootstrap team consciencously worked to have huge amount of transfer between programming and algebra.



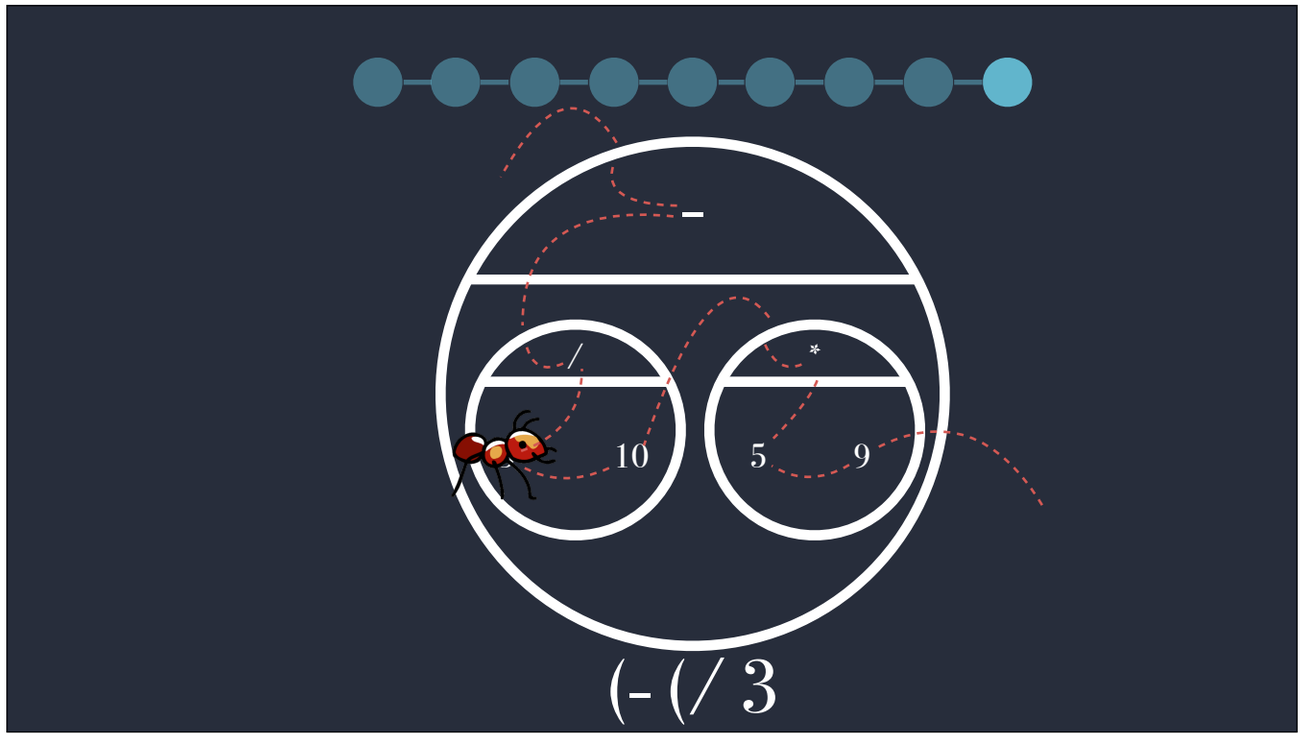
Okay, now we get our ant, and we walk through the circles.

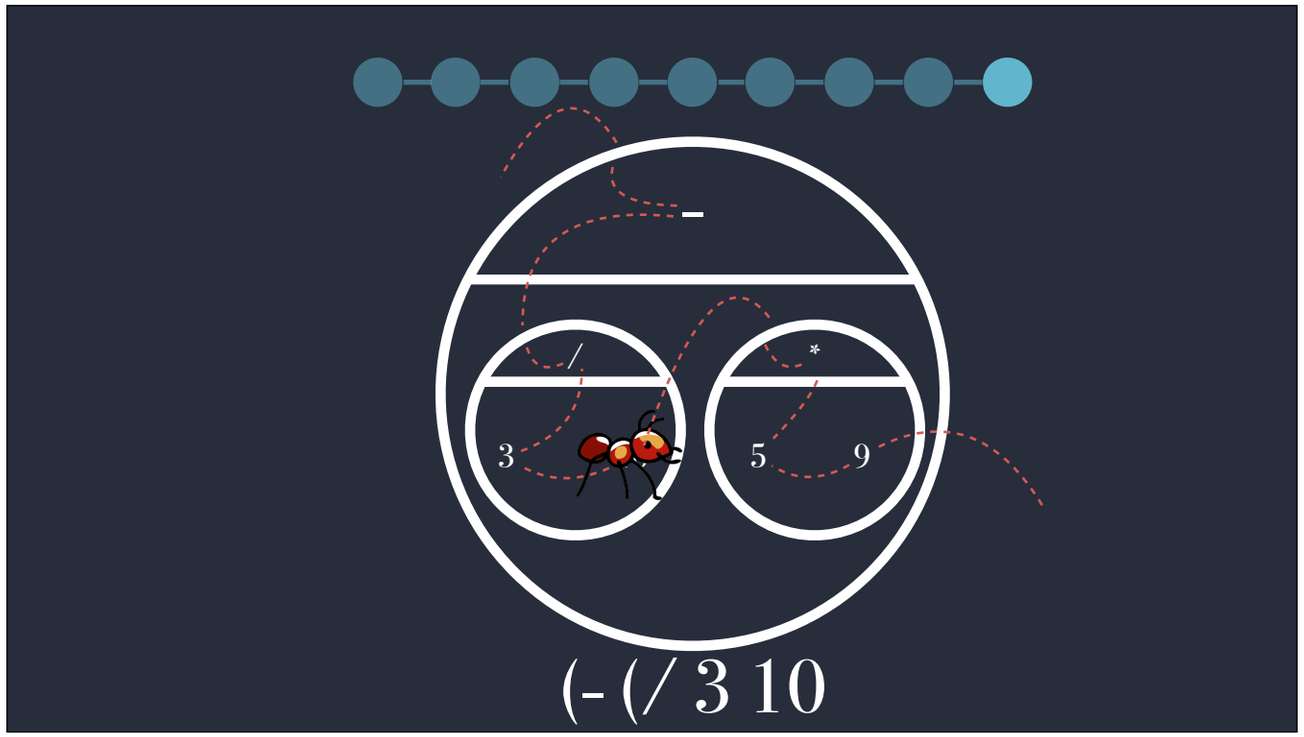

-
/
*
3
10
5
9
(

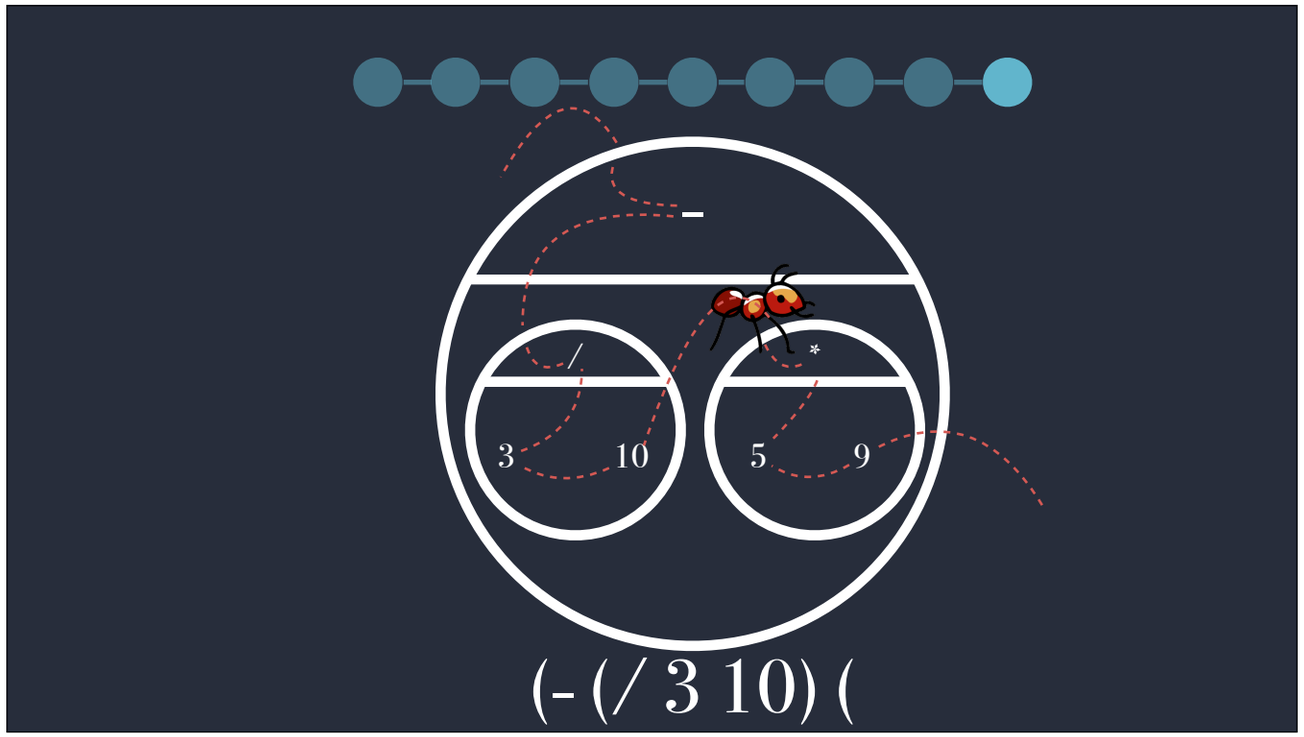

/
*
3
10
5
9
(-

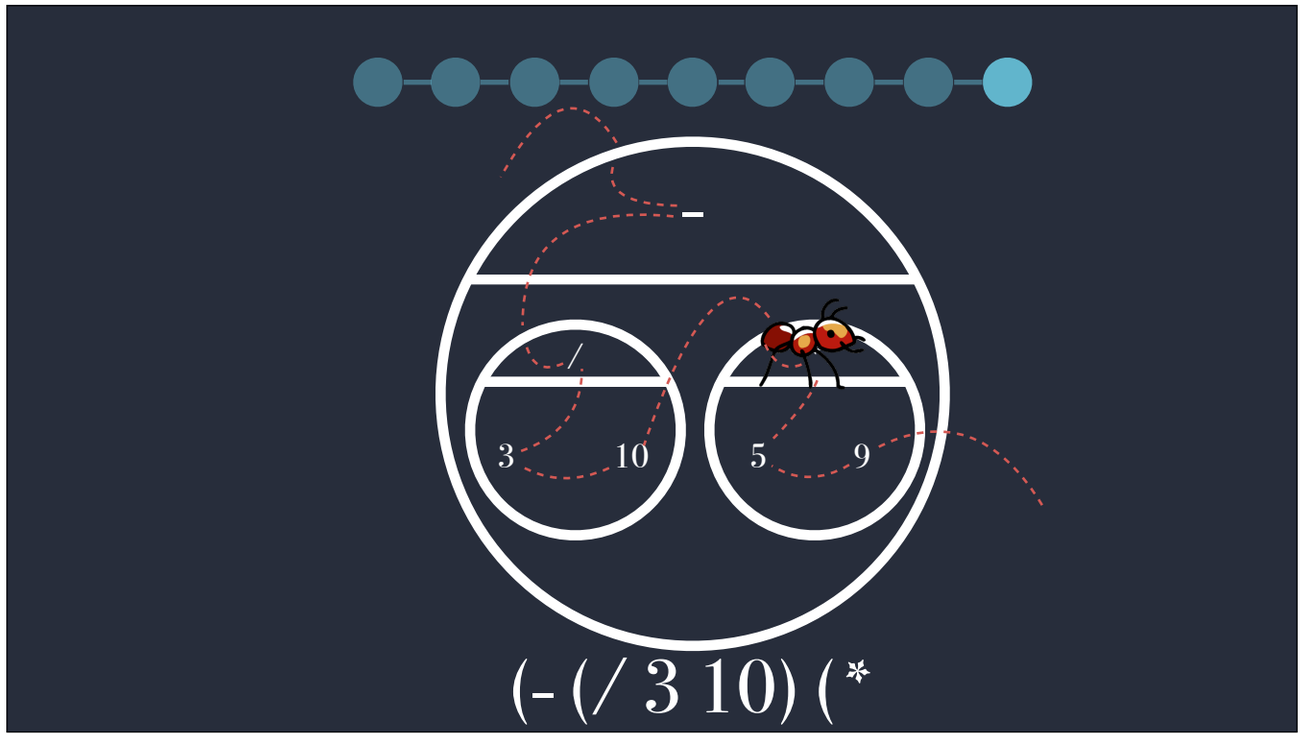

(- (
3
10
/
5
9
*
-

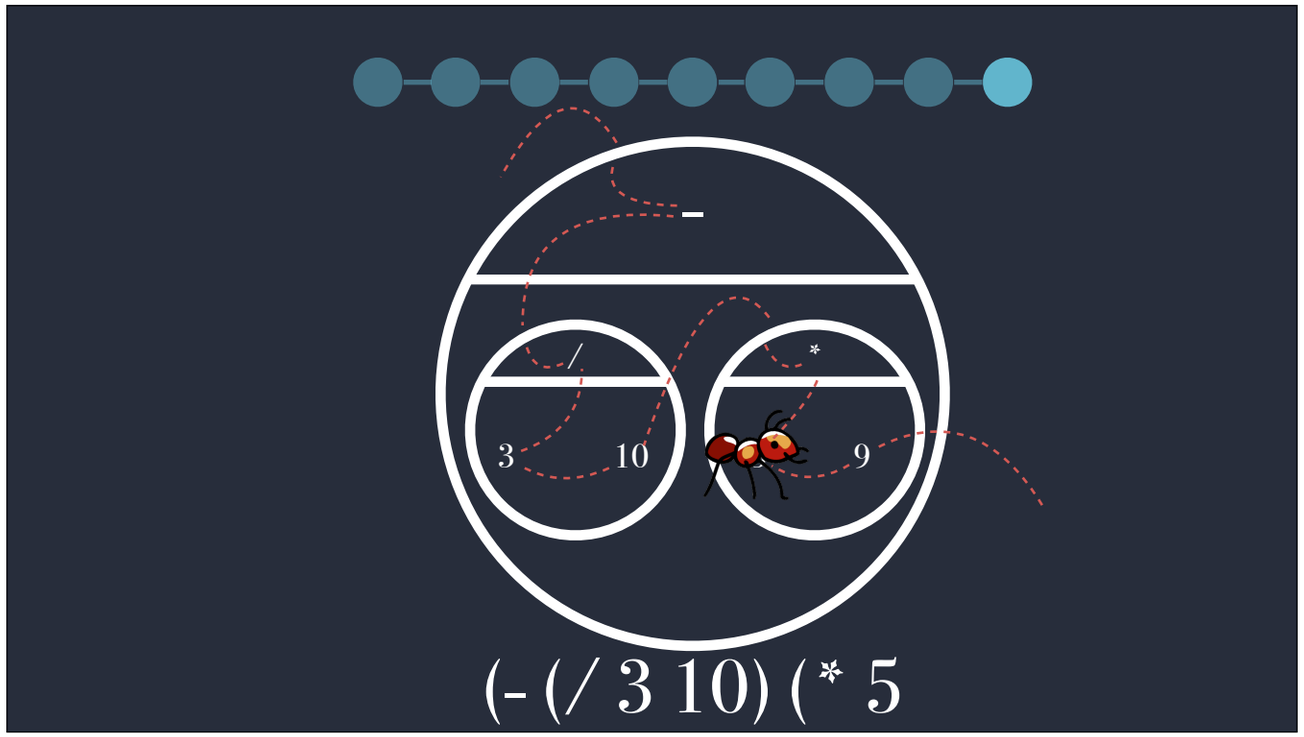

(- (/
3 10
5 9
*
-

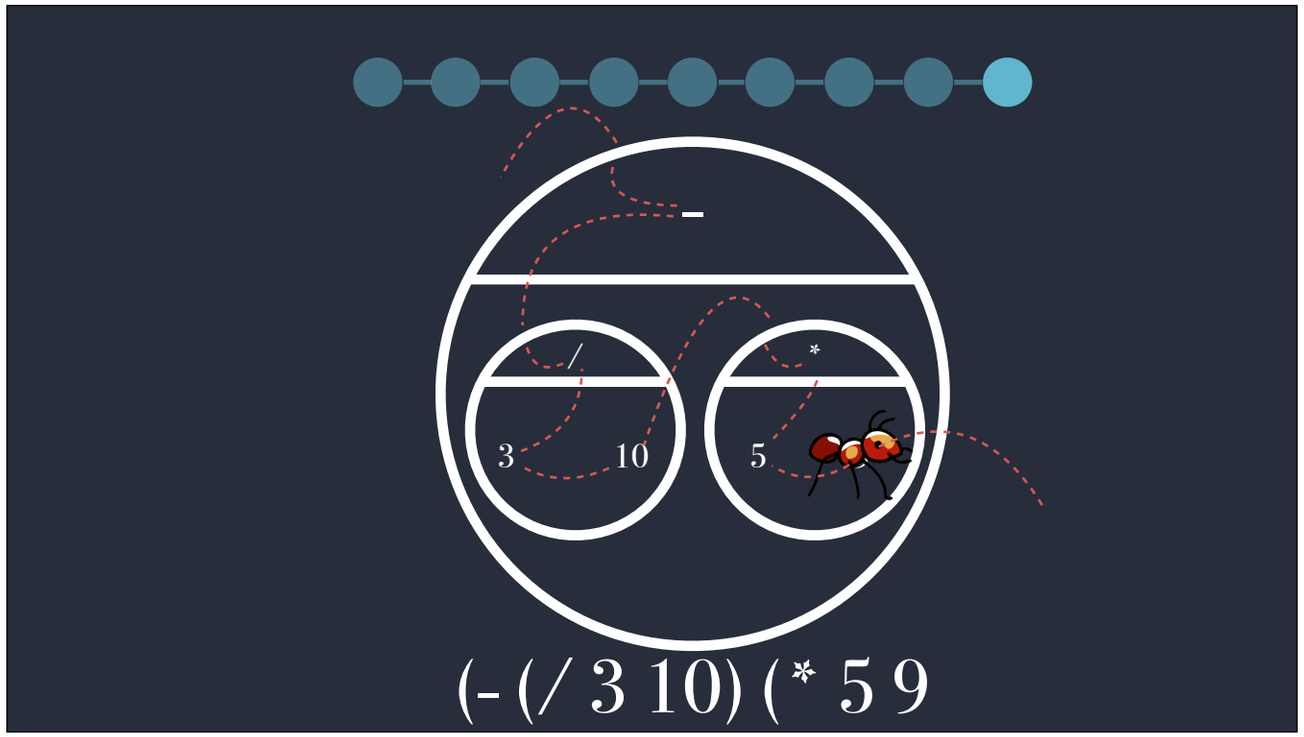

-
/
*
10
5
9
(- (/ 3

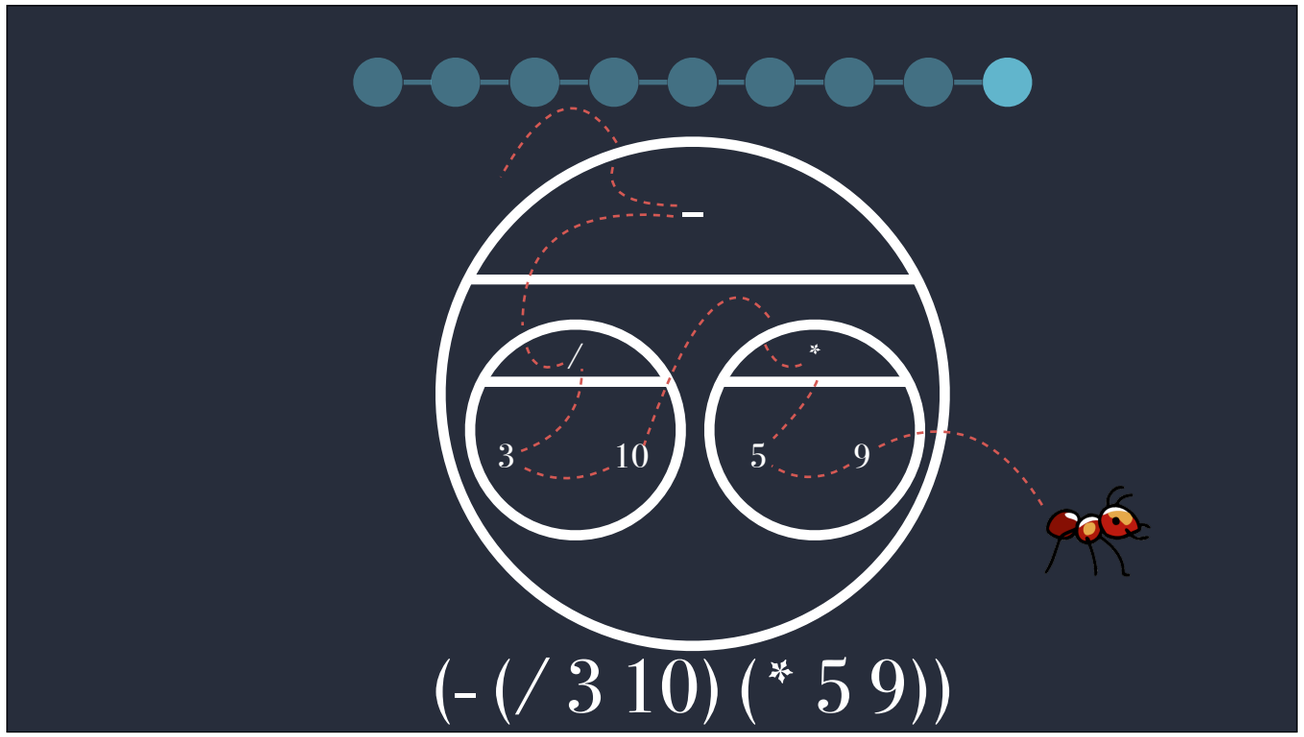

-
/
*
3
5
9
(- (/ 3 10


-
/
*
3
10
5
9
(- (/ 3 10) (


-
/
3
10
5
9
(- (/ 3 10) (*


-
/
*
3
10
9
(- (/ 3 10) (* 5


-
/
*
3
10
5
(- (/ 3 10) (* 5 9



And there’s your Racket code. Circles of evaluation are great tools for learning. They would have been most helpful last year when I was tossing out solutions to students instead of having them work through the issue themselves. Tools like the circles of evaluation allow for Constructionism to take place in the classroom.

However, not all errors are syntactic in nature. Since Racket is a functional language, students may have trouble writing a function from start to finish.

As such, the Bootstrap team implements something called the design recipe. This walks students through writing a good function from start to finish.

Programming Design Principles

Constructionism
Situate the learning
Readability
Recomposition/decomposition
Helpful error messages
Metaphor
See the state
Learning tools

Programming is taking a more central role in the lives of our children. Microsoft recently announced it will be spending $75 million over three years for better programming instruction. NYC wants every public school to offer CS courses over the next ten years. Even my employer is mandating CS classes for students 6th to 9th grades. If we want to be serious about programming, then we need to take a look at how we teach it. These 8 design principles are not a silver bullet, no single language (not even Elm) can claim to be the “best” language for beginners. But I truly believe that these 8 design principles can put us on the path to a more effective way to teach programming to students.

www.patr1ck.net

Here is my website where I have posted some links to the resources that I mentioned in this talk. I have also documented my capstone project, as well as some of the work I have done with Racket and Bootstrap over the summer. Thank you.