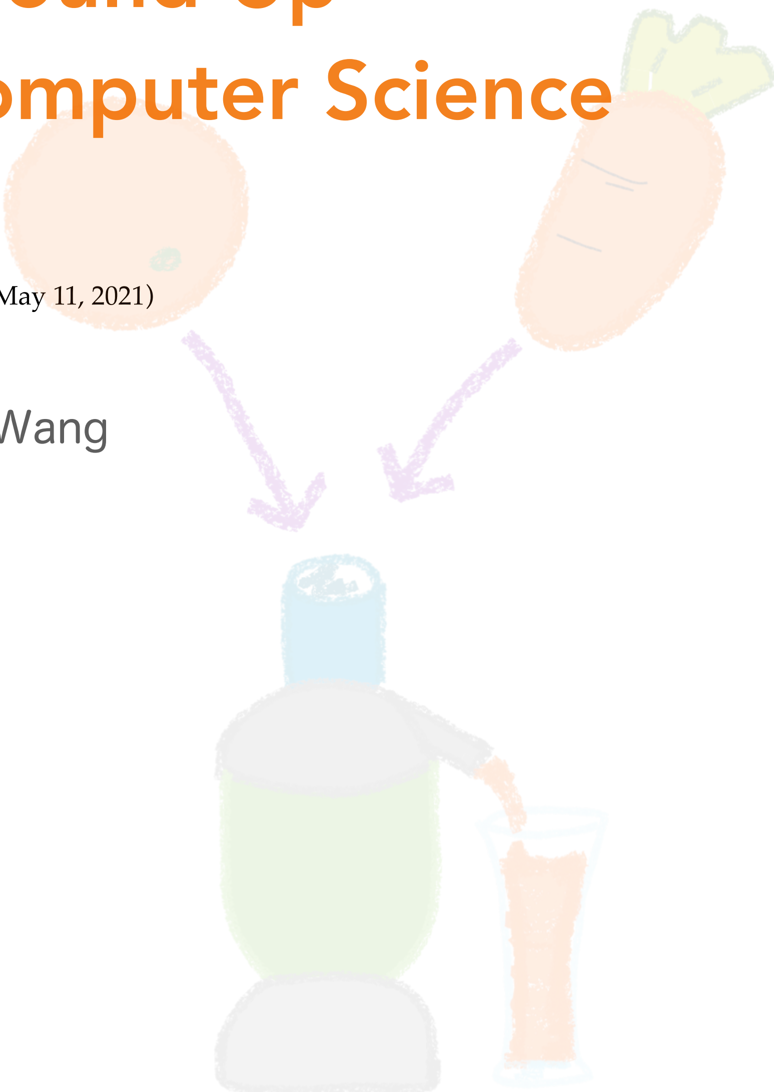
# Ground-Up Computer Science

(Draft. May 11, 2021)

Yin Wang



# Preface

This is an introductory book to computer science. Based on experience of teaching computer science to complete beginners and professionals, it is for people who hope to understand computer science at a deep level but have found no good place to start.

Many computer science introductory books have been written, but they are often not written with complete beginners in mind. Deep books often have hidden assumptions of the reader's prior knowledge, so they are likely to get frustrated and give up. Friendlier books often don't go deep, so readers only get to the surface. It is very hard to balance between depth and progress.

This book is written after extensive experiments on real beginners. Its contents and method are developed and tested with beginners to make sure they can practically grasp the important concepts without much struggle. Effort is needed but no effort is wasted. No prior experience of computer science or math is required. The hope is to guide people through the maze of computer science knowledge, reaching a clear and simple vision of its most important and profound principles.

Yin Wang
yinwang0@gmail.com
May 11, 2021
Shanghai

# 1 Functions

A: Why are you here?

B: I come to you, master, to be introduced to the art and science of computer programming.

A: Are you determined to spend a good part of your life on this art?

B: Yes, I'm sure. Computer programming is the art I hope to study whole-heartedly.

### Prior Knowledge

A: That sounds good, but let me first make sure that you have the prior knowledge and intelligence.

B: What is the prior knowledge needed?

A: Do you know the result of `2 * 3`?

B: Of course. The result is `6`.

A: Good. That is all you need to learn computer programming.

B: That seems too easy. Don't I need to know more math?

A: No, you don't need much math. Many teachers say that math is the foundation of computer science, but it is the opposite—computer science is the foundation of math. You don't really need a lot of math to understand computer science, but computer science can help you understand math.

B: Nobody has told me that before!

A: But your learning will not be an even road. A lot of effort is needed to master any art. The same is true about computer programming.

B: I understand. I will do my best.

A: I'm happy that you are determined, so let us begin tomorrow.

B: Great. Have a good night, master!

A: Good night!

### Expression, Value and Computation Graph

A: Good morning.

B: Good morning, master!

A: Did you have a good rest?

B: Yes. I enjoyed my sleep.

A: Good. Always have a rest after about an hour of learning. This will help the ideas to sink in. Don't rush through the materials. Eat them in small pieces and enjoy. This will help you digest.

B: I understand. I will enjoy my food of thought.

A: Yesterday you answered my question "What is the result of `2 * 3`".

B: That is 6.

A: How about the result of `1 + 2 * 3` ?

B: 7.

A: Right. Let us look deeper into `2 * 3` and `1 + 2 * 3` and see what they really are.

B: What is deeper about them?

A: `2 * 3` as you see here, is an *expression*.

B: What is an expression?

A: Expressions can be multiple kinds of things. We don't need to define the concept right now. Some examples are good enough. Here `2 * 3` is a kind of expression called *arithmetic expression*.

B: So I guess `1 + 2 * 3` is also an arithmetic expression?

A: Yes. `1 + 2 * 3` is also an arithmetic expression. There are many arithmetic expressions.

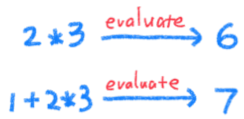
B: They all look similar.

A: An expression can have a *value*. Here the value of `2 * 3` is `6`. Notice that an *expression* and its *value* are two different things. `2 * 3` is an expression, not a value. Its value is `6`.

B: Then the value of the expression `1 + 2 * 3` is `7`.

A: Right. The process with which we get the value of an expression is called *evaluation*. When we *evaluate* `2 * 3`, we get the value 6.

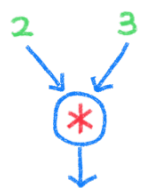
B: If we *evaluate* `1 + 2 * 3`, we get the value 7.




A: Yes. If you draw a picture, it may look like the above. Evaluation is a big topic in which you will spend a lot of time.

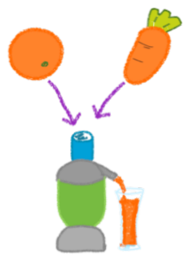
B: I have a long way to go.

A: What you see in `2 * 3`, the characters `2`, `*` and `3`, are *text*. Text is like letters in this book. Everybody can see the text, but not many of them can see its essence. We will now look at the essence of `2 * 3`. It is something like this:



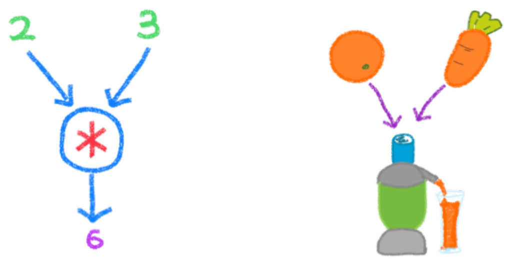

B: This looks like a circuit.

A: Yes. It looks like a circuit, or a pipeline. We may call this a *computation graph*. You may think of the multiplication sign (`*`) as a juicer with *input* and *output* pipes. If you put in orange and carrot, you get orange-carrot juice out.



B: I always enjoy juice.

A: Compare the pictures of `2 * 3` and the juicer, you will see that they are very similar.




B: I can think of `2` as the orange, `3` as the carrot, `*` as the juicer, and this produces output `6`, which is like the juice.

A: Right. The juicer runs, breaks up the orange and the carrot, mixes them into juice. The process of making juice is very similar to the process of *evaluation*.

B: Evaluation is as simple as making juice!

A: It is not always that simple. Juicers can be complex too, but indeed evaluation is very similar to making juice. The computer breaks up the numbers `2` and `3` into pieces, and then makes `6` out of those pieces.

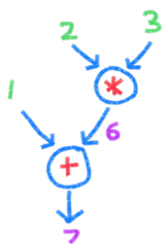
B: I can see how similar they are. What are the *pieces*?

A: The pieces are usually called *bits*, but we don't talk about bits in this course. Bits are low-level details. We don't want low-level details to obscure high-level ideas. For now a rough idea is good enough.

B: Okay.

A: Can you draw a computation graph of the expression `1 + 2 * 3` ?

B: Let me try...




A: Good. You can think of this as two different machines connected by pipes. The result produced by the multiplication machine (`6`) will flow into the addition machine as *input*. Together with another input `1`, the addition machine will produce the final result `7`.

B: You used the term *input.* Can I say `7` is the *output* of the addition?

A: Yes. `7` is the *output* of the addition. You can also say that the *output* of the multiplication is `6`.

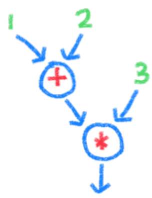
B: I see. Every machine has one or more inputs and an output.

A: Look at the picture. Can you see how the order of multiplication and addition is determined by the computation graph of `1 + 2 * 3` ?

B: Yes. Because the output of the multiplication is an input for the addition, so we have to get the result of the multiplication first.

A: Right, otherwise the addition cannot proceed. Can you draw a computation graph of `(1 + 2) * 3` ?

B: Like this?




A: Correct. Have you noticed that we haven't any parentheses in the computational graphs?

B: Right. There are no parentheses, but we have parentheses in the text `(1 + 2) * 3`. Why is that?

A: Because without parentheses we can't distinguish `(1 + 2) * 3` from `1 + (2 * 3)`. These have different orders of operation, so we have to use parentheses in the text. Why don't we need parentheses in the computation graph?

B: Because the graph itself can express the order of operations `+` and `*`?

A: Right. You can say that the order is specified by the *topology* of the computation graph. The term *topology* means how we connect the pipes.

B: It seems computation graphs are more expressive than text.

A: Yes. Computation graphs are the essence of text representations. `(1 + 2) * 3` and `1 + (2 * 3)` are just text representations of the computation graphs. From now on, when you look at expressions, try to think of their computation graphs. You don't have to draw every graph, but this vision will vastly help you understand expressions.

B: I'll keep that in mind.

### Console

A: Now you have learned the most simple computer programs. The expressions `2 * 3` and `1 + 2 * 3` are also programs.

B: Can I run them on a computer?

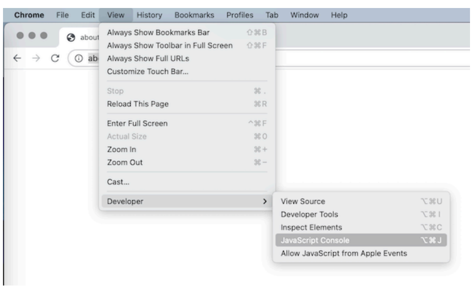
A: Yes, you can. Do you know that there is a programming language in every web browser?

B: I heard of it. It is called JavaScript.

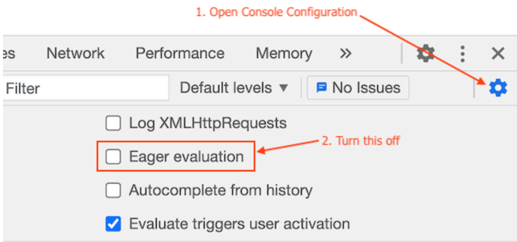
A: Yes. We do experiments and exercises in the JavaScript language. It is a decent language for our purpose, but the knowledge you learn here is not limited to JavaScript. You can apply it to any language.

B: Great. What shall I do now?

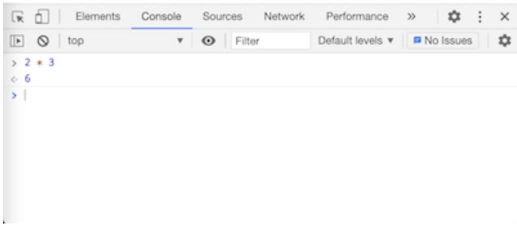
A: First, go to the menu of the Chrome browser. Choose the menu item *`View -> Developer -> JavaScript Console`*. This will open up the JavaScript Console.



Then turn off the *Eager evaluation* option, because this may distract your thoughts.




Then you can enter the above expressions `2 * 3` and `1 + 2 * 3` into the console. Try more expressions if you like.




You may need to find ways to open it in other browsers or operating systems. They are all similar.

B: I got it working.

A: It looks like a calculator, right?

B: Yes, but seems more advanced.

A: Right. It has the power of the JavaScript language in it, which is a lot more powerful than a usual calculator. We will make use of it soon.

B: Nice.

### Variable, Function and Function Call

A: Now we proceed. Today you will learn three most important elements of a programming language: *variable*, *function*, and *function call*.

B: Only three?

A: Yes, but with these three basic elements, you can construct very interesting and powerful programs, as you will do in the exercises.

B: Are they elements of every programming language, or just JavaScript?

A: They are elements of most modern programming languages. You are not tied to any programming language in this course.

B: Good to know that.

### Variable

A: The following code creates a *variable* named `x`. Enter it into the console and see what happens. Make sure you put a semicolon at the end to separate it from code that follows it.

```
var x = 2 * 3;
```

B: Console gave me *undefined*. What does it mean? I'm defining a variable, but it shows me *undefined*. I was expecting something like 6.

A: The name *undefined* is a bit misleading, but let's deal with it. `undefined` is the value of the expression `var x = 2 * 3` which you just entered.

B: Is `var x = 2 * 3` another kind of expression?

A: Yes. `var x = 2 * 3` is a different kind of expression. It is called a *variable definition*. It is not an arithmetic expression although it contains one (`2 * 3`).

B: So it did create the variable `x` for me?

A: Yes. You can check the value of `x` by entering it into the console.

B: Console gave me 6, as expected. But why did console gave me *undefined* for the variable definition `var x = 2 * 3` ?

A: I said that the name *undefined* is misleading. Here undefined is just a special value. It is the value of the variable definition expression `var x = 2 * 3`. It basically means "The action of defining the variable is done."

B: I see. How can I make use of this *undefined* value?

A: No, you never use it. Don't write *undefined* yourself. It is there just to satisfy the concept that "everything you enter is an expression".

B: That is a bit strange, but I'm okay with it.

A: Every language has some strangeness. As long as you grasp the important things, they will not cause much trouble.

B: Good to know that.

A: Now that you have the variable `x`, you can use it in any place where `6` can be used. You can make some examples and try them.

B: I tried `1 + x`, `3 * x`, `4 - x`, `x * x`, `2 * x - 1`. The results are as expected.

A: Very good. Actually everything you just entered is an *expression*, and console gave you its *value*. Again, it is important to distinguish an expression from its value. `2 * 3` is an expression, `var x = 2 * 3` is also an expression.

B: This seems to be a good idea. Everything I enter into console is an expression, so I don't have to think which one is an expression.

A: Is `x` an expression?

B: No, `x` is a variable.

A: It seems you forgot what you just said: "Everything I enter into console is an expression."

B: Oh, my bad.

A: Actually a variable is also an expression, because you can enter it into the console, and you can get its value. `x` is an expression, so are `1 + x`, `3 * x`, ...

B: I see.

A: Is `6` an expression?

B: No, it is a value.

A: Try enter `6` into console and see what happens.

B: It gave me 6.

A: The `6` you entered is an *expression*. The 6 that console gave you is a *value*. Those two 6's are different things.

B: It may take me a while to understand this.

A: This is actually quite deep topic. For now, just repeat this to yourself: Whatever you enter into console is an expression, and whatever console gives you is a value.

(I may be cheating a bit for now, but this is good for you. Let us continue.)

B: Okay. So `2 * 3` is an expression, `x` is an expression, `6` is an expression, `var x = 2 * 3` is also an expression.

A: Right. Expressions can be a variety of things. Now define another variable y and see what happens.

```
var y = x;
```

B: I got *undefined*.

A: Correct. That means the variable `y` is defined. Now check the value of variable `y`.

B: I entered `y` into console and got 6.

A: Excellent. Now `x` and `y` have the same value, 6.

## Function

A: Let us look at what is a function. You can create a function for calculating the square of a number like this:

```
x => x * x
```

B: I entered this into console, but I got the same thing back.

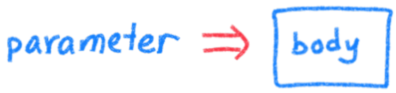
```
> x => x * x
<· x => x * x
```

A: This is as expected. A function is also an expression. Its value looks like itself.

B: I'm surprised.

A: Look at the function `x => x * x`. In the middle there is an arrow `=>`. It is used to separate the two parts of a function. The left hand side of `=>` is a *parameter* or input. The parameter is a name. The right hand side of `=>` is the *function body*, or just *body*. The function body is an expression which describes how the function computes the output.

B: That seems clear.




A: How many parts does the function `x => x * x` have?

B: Three. The parameter `x`, the function body `x * x`, and the arrow `=>`.

A: No, it has only two parts: the parameter `x` and the function body `x * x`. The arrow `=>` is there just to separate the two parts. We say that the arrow is just *syntax*. It is not an essential part of the function. The purpose of `=>` is just to make clear which part is which, otherwise we would have something like `x x * x`, which is confusing.

B: Indeed. Now I have a better understanding of what *syntax* means.

A: Other languages have functions too, and each function also has two parts, but other languages may not have the arrow `=>`. They may use other ways to separate the two parts.

B: I see.

A: The function by itself does nothing, because you haven't given it any input. It is like a juicer without fruits. We now give the function an input and see what happens.

B: How do we do that?

A: Try this

```
(x => x * x)(3)
```

B: The result is 9.

A: You see its syntax? First, we put the input `3` into a pair of parentheses, and then append it to the function `x => x * x`. This way we have given the function `x = > x * x` an input `3`.

B: I noticed that there is a pair of parentheses around the function `(x => x * x)`, why is that?

A: If you omit the parentheses and just write `x => x * x (3)`, JavaScript will think that you have given `3` to a function named `x`, which is not what we meant, so we put the function into parentheses to make it act as a whole.

B: I see. Without the parentheses this will look confusing and ambiguous.

### Function call

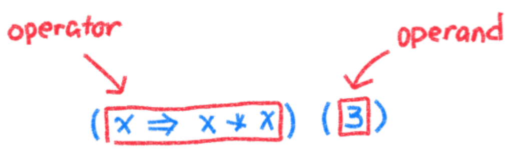
A: This construct `(x => x * x)(3)` is a *function call*, or just *call*. So now you have learned all three basic constructs of programming languages: *variable, function, call*.

B: That is so quick.

A: A function call has two parts. The first is the *operator*, the second is the *operand*. For example in this call `(x => x * x)(3)`, can you tell me which is the operator and which is the operand?

B: The operator is `x => x * x`, and the operand is `3`.

A: Correct. Remember, the function call has two parts.




B: I will remember that.

A: How many parts has a function?

B: Two parts. The parameter and the function body.

### Functions with names

A: Good, you have remembered it. Have you noticed that our functions `x => x * x` doesn't have a name?

B: Isn't its name x?

A: No. x is the name of its parameter, not the name of the function itself. The function doesn't have a name.

B: Indeed. I don't see its name in `x => x * x`.

A: Functions don't really have names, but it may be inconvenient if we use them without names, because then we have to copy them every time we use them.

B: That will make the code very complicated.

A: Now we find a way to give functions names. Actually you can do it with what you have just learned.

B: Let me see. Can I use variables to name functions?

A: That is a good observation. Try that.

B: Something like this?

```
var square = x => x * x;
```

A: Correct. Try it in console.

B: *undefined*.

A: Now enter the name `square` into console.

B: It shows me `x => x * x`.

A: As expected, right?

B: Yes. Does that mean that `x => x * x` is the value of the variable `square`?

A: Exactly. This `var square = x => x * x` is not that different from `var x = 2 * 3`. Both `2 * 3` and `x => x * x` have values. We just use variables `x` and `square` to refer to their values.

B: Got it.

A: Can you give an input `3` to *the function referred to by the variable* `square`? For brevity, we can also say "the function `square`" to mean the exactly same thing.

B: My guess is

```
(square)(3)
```

A: Try it.

B: I got 9.

A: Correct. Here the name `square` has just one part, so this will not confuse JavaScript. So you don't need to put parentheses around the variable `square`. You can just say `square(3)` instead.

B: I see.

```
square(3)
```

A: Let's take a look what actually happens when you enter `square(3)`. First, the variable `square` will be evaluated and the value is `x => x * x`. Also `3` is evaluated and the value is 3. Then `square` inside `square(3)` is replaced with `x => x * x` and we have `(x => x * x)(3)`. Is that easy?

B: That is understandable. Then we get 9 from `(x => x * x)(3)`.

## The parameter's scope

A: Now enter `x` into console, and tell me its value.

B: Hmm... it is 6.

A: Why isn't the value 3, since you entered `(x => x * x)(3)` before, and 3 gets into x?

B: I guess this 6 is from `var x = 2 * 3` which I entered before?

A: Right. This experiment shows that function calls will not change the variables outside of the function, even if they have same names.

B: That is interesting. Why is that?

A: The function's parameter name is only *visible* inside the function body. It is not the same thing as the variable defined outside of the function. This is to ensure that you don't change other variables accidentally when calling a function.

B: This sounds reasonable.

### Substitution

A: Now we take a closer look at how function calls are evaluated.

B: Okay.

A: When `(x => x * x)(3)` is evaluated, we first replace every `x` inside the body `x * x` with `3`, and we get `3 * 3`. This process is called *substitution*.


B: From the substitution we get `3 * 3`, and not `x => 3 * 3` ?

A: Yes. Remember, the function call's value is the value of the *function body* after the substitution. If we get `x => 3 * 3`, then next step we won't get 9. `x => 3 * 3` is another function, not a number.

B: This sounds reasonable. By definition, the function body describes how to compute the output value.

A: That is a very good understanding.

B: I'm happy.

A: Now, do a substitution of `(x => 2 * (x + 3))(5)` and show me the result.

B: *(Write your own result with pen and paper, without using console, and send it to the teacher.)*

A: Good. Always remember the purpose of *function body*.

B: Okay.

A: *Substitution* as described here is a way of thinking about function calls. It will help you understand function calls, but this may not be exactly how the machine evaluates the function call. For performance reasons the machine may be more clever about the evaluation process.

B: Okay. I will try to use it when I do exercises.

A: The first part of this class is good for now. Have a hour of rest and come back later.

B: Thank you master!

### Function of more than one parameter

A: We haven't learned all about functions yet.

B: What is more about them?

A: A function can have more than one input.

B: I was about to ask about that.

A: Here is a function with two parameters. The syntax is to separate the parameters with a comma, and put them into parentheses.

```
(x, y) => x + 2 * y
```

B: That is clear.

A: Can you figure out how we can call this function, with two inputs 1 and 3?

B: `((x, y) => x + 2 * y)(1, 3)`.

A: Correct. Try it in the console.

B: I got 7. After a substitution, the body becomes `1 + 2 * 3`. The value is 7.

A: Nice. This is a good use of substitution.

B: It seems to be a very useful tool.

### Function as output from another function

A: Functions are values, just like numbers are values. You have already defined variables whose values are functions, for example `var square = x => x * x`. Now we will see that functions can also be the output of another function.

B: What does that mean?

A: Try this function:

```
x => (y => x + y)
```

Can you see what it means?

B: I'm a bit confused. It has two arrows in it.

A: Don't be afraid. There is nothing in this that you haven't learned already. Remember that a function has just two parts, parameter and body. What are parameter and body of the first arrow?

B: The parameter is `x`, the body is `y => x + y`.

A: Correct.

B: So this function's output is a function `y => x + y`?

A: Right. This is what I mean by "function as output".

B: I see. What is the use of this function `x => (y => x + y)` ?

A: Give this function an input `2`, and see what you get.

B: I entered `(x => (y => x + y))(2)` and got `y => x + y`.

A: This proves that the function will return a function, right?

B: Yes, but I still don't see how this is useful.

A: Actually `y => x + y` is not the complete output. The console is hiding something from you.

B: The console is hiding things?

A: Try a substitution on `(x => (y => x + y))(2)`.

B: The function body is `y => x + y`. I replace the `x` inside it with `2`, and I get `y => 2 + y`.

A: Correct. But the console gave you `y => x + y` as the value of `(x => (y => x + y))(2)`. The actual output should be `y => 2 + y`. This is what I mean that console is hiding things. The console is hiding the information that it knows that "`x` is 2 inside `y => x + y`".

B: Interesting. Why does it do that?

A: It is a bit early to explain this here. Try to give `y => 2 + y` input 3, and see what happens.

B: I entered `(y => 2 + y)(3)`, and got 5.

A: Now try `(x => (y => x + y))(2)(3)`.

B: I got 5 too.

A: Does this mean the function returned from `(x => (y => x + y))(2)` behaves like `y => 2 + y`?

B: Yes.

A: What does this mean to you?

B: I think it proved that what I got from `(x => (y => x + y))(2)` is `y => 2 + y` and not `y => x + y`.

A: Correct. It might be interesting to see what you can get from `(y => x + y)(3)`.

B: I entered it and got 9. That is strange.

A: Do you remember that you have a variable named `x` outside of the function?

B: I see. I have a variable `x` whose value is 6. `(y => x + y)(3)` is substituted into 6 + 3, thus result 9.

A: Exactly. This example shows you that you can create functions inside another function. The output of `(x => (y => x + y))(2)` is a new function.

B: Nice. I haven't seen functions that creates functions before.

A: Functions that can return functions as output, is called *high-order functions*.

B: I have heard of the term high-order functions before, but I didn't expect to learn this at such early stage.

A: It is possible to understand this, and I believe this is the best time to understand it. Many of my students succeeded in learning functions this way, and you will too.

B: Thank you.

A: `(x => (y => x + y))(2)(3)` is hard to read. You may use variables to break it up.

B: I tried

```
var f = x => (y => x + y);
var g = f(2);
g(3)
```

I got 5 in the end. The same result.

### Function as input for another function

A: That is good. You have seen functions used as output from another function. Now I will show you that functions can also be used as *input* to another function.

B: Output then input. It seems that we will cover every case.

A: That is right. Look at this function `apply`:

```
var apply = (f, x) => f(x);
```

The first parameter `f` here is a function. We usually use the parameter names `f`, `g`, `h` etc for functions.

B: I see. What will this `apply` function do?

A: The function `apply` takes a function `f` and input `x`, then passes `x` to `f`. It just calls the function `f` with input `x`.

B: It doesn't seem to be very useful.

A: It may not be very useful, but it is a very simple example. Now try this:

```
apply(x => x * x, 3)
```

B: I got 9.

A: Can you show me what is going on by using substitution?

B: Yes. In `apply`'s body `f(x)`, replace `f` with `x => x * x` and replace `x` with `3`, I get `(x => x * x)(3)`, whose value is 9.

A: Excellent. Try this also:

```
var h = x => x * x;
apply(h, 3)
```

B: The result is the same, 9.

### Function as input and output

A: Last example. This time we make a function which takes two functions as input, and produces a function as output.

B: A function whose inputs and output are all functions?

A: Right. Here is an example. Its name is `compose`.

```
var compose = (f, g) => x => f(g(x));
```

B: This one is harder to read.

A: When you see more than one arrows, focus on the first arrow and mentally put everything else after it into parentheses, so this is equivalent to

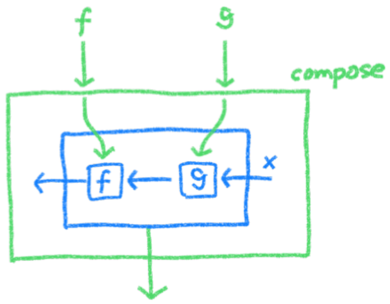
```
var compose = (f, g) => (x => f(g(x)));
```

But using fewer parentheses may make it easier to read when things become complex.

B: I see, but I don't understand what it does.

A: A computation graph may help here. The `compose` function takes two inputs `f` and `g`, both are functions, and produces a function `x => f(g(x))`.

B: It gets better now, but what is the purpose of this?




A: Think about juicers and other machines. You give the compose function two machines and it connects them with pipes. The output of the first machine becomes the input of the second machine. The output of compose is the assembly of the two machines.

B: I see. It creates a new function by connecting two functions.

A: Try use the compose function with this example:

```
compose(x => x * x, x => x + 1)(3)
```

B: I got 16.

A: Can you see why you got 16?

B: Here compose's parameter `f` is `x => x * x` and `g` is `x => x + 1`, so substitution gives me `(x => x * x)((x => x + 1)(3))`. One more substitution, I get `(x => x * x)(3 + 1)` and then `(3 + 1) * (3 + 1)`, and so on, and the result is 16.

A: Excellent. Now can you simplify this expression `compose(x => x * x, x => x + 1)`? Can you rewrite it into a simple function of the form `x => ...` without using `compose`?

B: *(Write your solution, and send it to the teacher)*

### Parameter names don't matter

A: Let's look at another thing. Are these two functions equivalent?

```
x => x * x
y => y * y
```

B: What does *equivalence* mean here?

A: Equivalence of functions means, if you give two functions the same input, they will always produce the same output.

B: That makes sense. I just tried calling those functions with multiple different inputs. They seems to be equivalent.

A: Those two functions both computes the square of the input, no matter what its name is, so of course they are equivalent.

B: It seems that the names of the parameters don't matter. I can change it to any name.

A: Right, but you need to be careful about consistency.

B: For example?

A: Is `(x, y) => x + 2 * y` equivalent to `(u, v) => u + 2 * v`?

B: Yes.

A: Is `(x, y) => x + 2 * y` equivalent to `(y, x) => y + 2 * x`?

B: Yes.

A: Is `(x, y) => x + 2 * y` equivalent to `(y, x) => x + 2 * y`?

B: No. Those are different.

### Alternative syntax of functions

A: We are almost done with this lesson. Before I give you exercises, I need to let you know an extra syntax provided by JavaScript and most other languages.

If we want to give functions names, there is a simpler way to write this.

```
function f(x)
{
  return ...;
}
```

This is equivalent to

```
var f = x => ...;
```

You may see this syntax in the exercises.

B: It seems there is a lot in the exercises.

A: Yes. Don't be afraid or confused. You only need to use things you just learned. Please don't search online for solutions and just use your own head. This will deepen your understanding.

B: Sure. I'll do my best.

A: If you are stuck for longer than an hour, please let me know. I may give you hints.

B: Thank you!

A: Here are the exercises. *(Exercise omitted for the sample)*