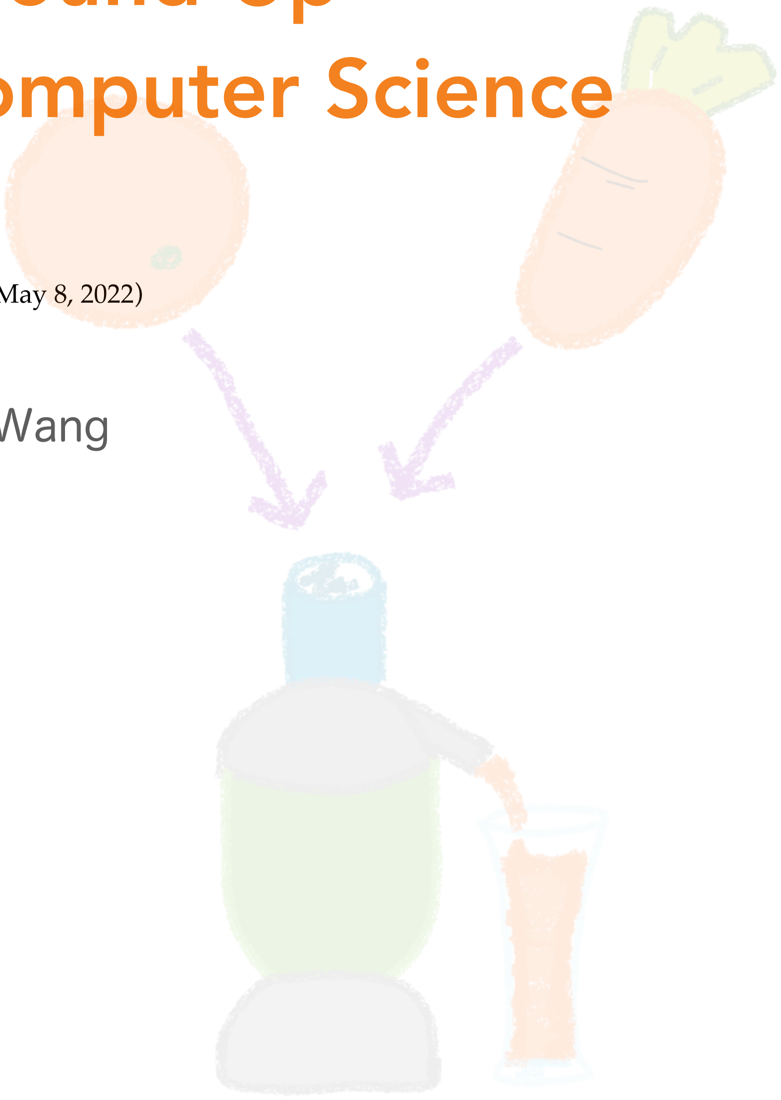
Ground-Up Computer Science

(Draft. May 8, 2022)

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 into a deeper 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. Advanced books often have hidden assumptions about the readers' prior knowledge, so they are likely to get frustrated and give up. Friendlier books often don't go deep enough, so readers only get to the surface. It is very hard to balance between depth and progress.

This book is written after extensive practice of teaching real beginners. The contents and methods are repeated developed and refined 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, achieving a clear and simple vision of its most important principles.

Yin Wang
yinwang0@gmail.com
May 6, 2022
Shanghai

1 Functions

A: Welcome.

B: It's good to be here!

A: How about learning some computer programming today?

B: That seems to be a hard topic. I'm afraid that I don't have any prior knowledge.

A: What is the result of `2 * 3` ?

B: `6`.

A: How about `1 + 2 * 3` ?

B: 7.

A: So you have enough prior knowledge to start with.

B: I guess I'm surprised.

A: Now let's look deeper into `2 * 3` and `1 + 2 * 3` and see what they really are.

B: What's deeper about them?

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

B: What is an expression?

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

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

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

B: I see.

A: An expression can have a *value*. For example, 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: So I can say the value of the expression `1 + 2 * 3` is `7` ?

A: Right. The process with which we compute 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.

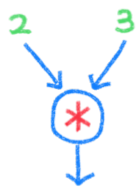
$$2*3 \xrightarrow{\text{evaluate}} 6$$

$$1+2*3 \xrightarrow{\text{evaluate}} 7$$

A: Yes. If you draw a picture, it may look like the above. Evaluation is a big topic.

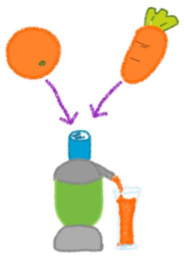
B: That's clear. I like pictures.

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 can see its essence. We will now look at the essence of `2 * 3`. It is something like this:



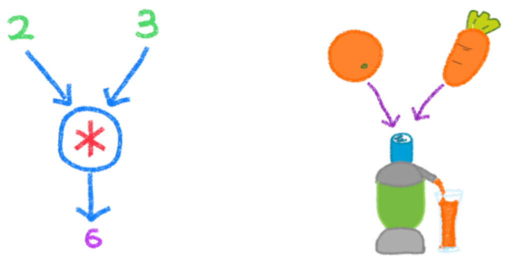
B: This looks like a circuit.

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



B: This is really intuitive.

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 evaluating expressions.

B: That seems really simple.

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.

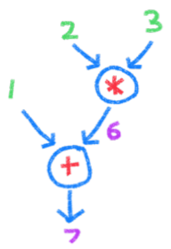
B: I can see how similar they are. What are the *pieces*?

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

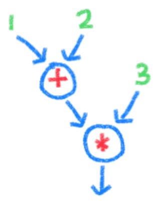
B: I see. Every machine has one or more inputs and an output.

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. The order is specified by how we connect the pipes.

B: So 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 computation graphs. From now on, when you look at expressions, try to think of the corresponding computation graphs. You don't have to draw every graph, but this vision will greatly help you understand expressions.

B: Okay. I'll keep that in mind.

Console

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

B: Can I run them on a computer?

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

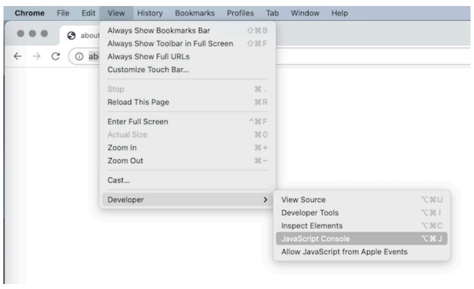
B: I heard of it. It is called JavaScript.

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 does not depend on JavaScript. You can apply it to any language.

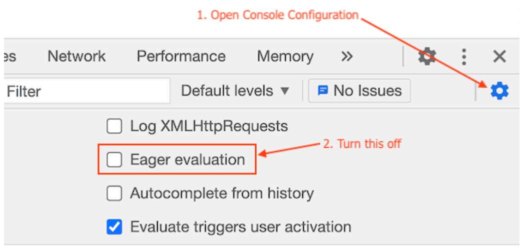
B: Great. What shall I do now?

(From now on, please use your computer and follow every step. Practice and play with it is the best way to learn.)

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

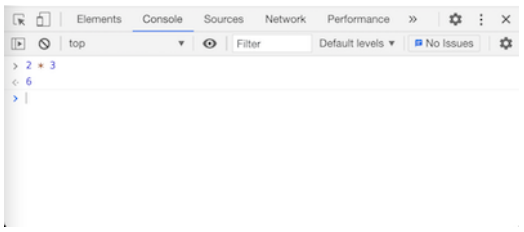


Then turn off the *Eager evaluation* option, because this feature may distract your thoughts.



- [ ] Log XMLHttpRequests
- [ ] Eager evaluation
- [ ] Autocomplete from history
- [x] Evaluate triggers user activation

Then you can type the above expressions `2 * 3` and `1 + 2 * 3` into the console Try some other expressions if you like.



```
> 2 * 3
< 6
>
```

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

B: Okay, 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 calculator. We will make use of it soon.

B: Nice!

Variable, Function and Function Call

A: Here we go. Today you will learn three most important elements of programming languages: *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.

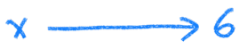
Variable

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 was expecting something like 6.

A: var x => 2 * 3 will associate the name x with the value 6, as in the following picture.



The *action* of creating such an association is not a value itself, so JavaScript gave you the special value *undefined*. It basically means "The action of defining the variable is done, but this expression var x => 2 * 3 has no value."

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

A: Yes. var x = 2 * 3 is a new 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 how can I make use of the *undefined* value I got from var x => 2 * 3?

A: 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 something similar to *undefined*. It is usually called *void* in other languages. They are all of the same nature.

B: Good to know that.

A: This kind of expression with *undefined* value are also called *statements*. They make some action happen but they don't have a value.

B: Okay.

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. 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: Variables are also expressions, because you can enter them into the console, and you can get their values. `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: 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.

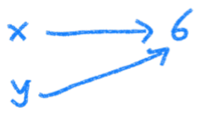
```
var y = x;
```

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. It's like this picture now.



B: So x and y points to the same 6?

A: Right. When you evaluate `var y = x`, JavaScript will evaluate x first and get the value 6. Then it associates y with the value 6. y is never associated with the variable x.

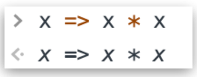
B: So x and y share the same value.

Function

A: Right. Let's 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.



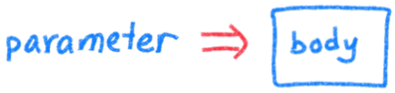
```
> x => x * x
<· x => x * x
```

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

B: I'm surprised. I thought a function is supposed to compute something.

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: I see. 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.

Try this

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

B: The result is 9.

A: This way we have given the function `x = > x * x` an input `3`, just like you put a fruit into a juicer. And then it starts working.

B: Nice.

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

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 this clear.

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

Function call

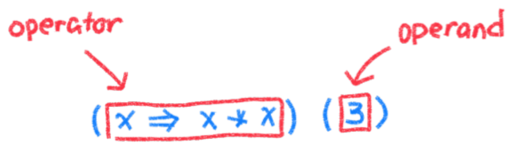
A: This construct `(x => x * x)(3)` is a *function call*, or just *call*. So now you have learned all three basic constructs of programming languages: *variable*, *function* and *function call*.

B: That is so soon.

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.



B: So a function call has two parts?

A: Right. How many parts has a function?

B: Also two parts. The parameter and the function body.

A: It might be easy to confuse functions with function calls. They are two different kinds of constructs. Make sure you can distinguish them.

B: That's easy. A *function* won't do anything without input. If we give it input, it is a *function call*.

Functions with names

A: Very good. Have you noticed that the function `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.

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

B: Right. 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: Good observation. Try that.

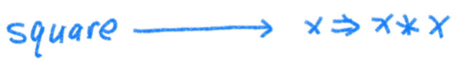
B: Something like this.

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

A: That's right. Try it in the console.

B: *undefined*.

A: The situation is now like this picture. The variable `square` is associated with the function `x => x * x`.



B: It's just like the picture with `var x = 2 * 3`, except that this time the value is a function.

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

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

A: This variable definition `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. There is nothing new in this one.

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

B: `(square)(3)`.

A: Try it.

B: I got 9.

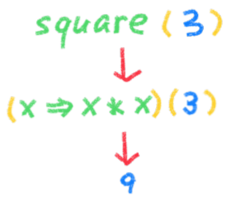
A: Correct. Here the name `square` has just one part, so there is no confusion in syntax. You don't need to put parentheses around the variable `square`. You can just say `square(3)` instead.

B: I see. `square(3)`.

A: What is the operator and operand in `square(3)`?

B: The operator is `square`. The operand is `3`.

A: Good. Let's take a look at what happens in small steps when you enter `square(3)` into console. The operator `square` will be evaluated and the value is `x => x * x`. The operand `3` is also evaluated and the value is 3. So essentially `square(3)` is `(x => x * x)(3)`. And then we get `9` from `(x => x * x)(3)`.

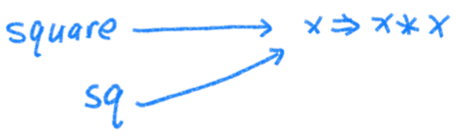


B: I see. That is the same process as other variables. We can use the name `square` wherever we need the function `x => x * x`. We just substitute its value in there.

A: You can also give the function another name, for example

```
var sq = square;
```

B: This is much like `var y = x` we have done previously, so I think `sq` will be pointing to the same function as `square`. Like this picture.



A: Try `sq(3)`?

B: That's the same thing as `(x => x * x)(3)`, so I get 9.

The parameter's scope

A: Good. There is nothing new here. Now type `x` into the console, and let me know its value.

B: Hmm... it is still `6`.

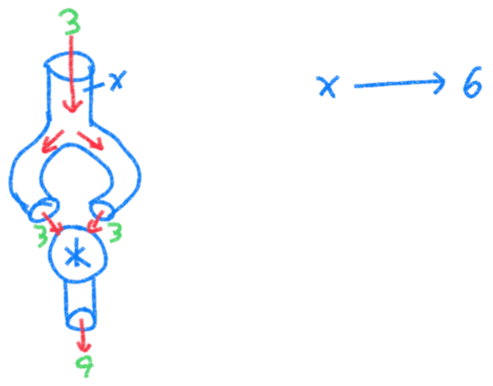
A: Why hasn't the value changed to `3`, since you entered `(x => x * x)(3)` before, and `3` gets into `x`?

B: I guess this `6` is from our first variable definition `var x = 2 * 3`?

A: Right. This means 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 like a label for the input pipe. It is only visible inside the function body. It is not the same thing as the variable defined outside of the function. So you won't be able to change outside variables accidentally when calling a function.



B: The picture is very clear. That sounds a reasonable design.

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*.

$$(x \Rightarrow x*x)(3) \xrightarrow{\text{substitution}} 3*3$$

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

A: Yes. 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: *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!

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. 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: Did you notice 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.

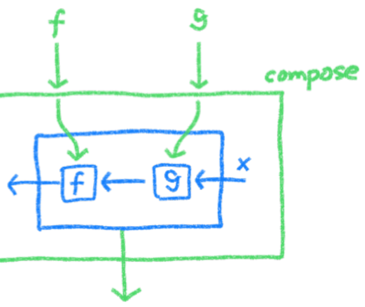
A: When you see more than one arrows, focus on the first arrow and mentally put everything else after it into parentheses, so this is equivalent to

```
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)`, 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. If you give the following two functions the same input, do they always produce the same output?

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

B: Those two functions both computes the square of the input, so of course they always produce the same output, no matter what the parameter name is.

A: Excellent. It seems that the parameter names don't matter. Can we change it to any name?

B: Yes, I think so.

A: Let's see some other examples. 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.

A: Is `x => y => x + y` equivalent to `x => x => x + x`?

B: *(Think about your answer and discuss with your teacher.)*

Equivalence of x => e(x) and e

A: We have seen the first kind of equivalence, where functions remain the same when only parameter names are changed.

B: Do we have other kinds of equivalence?

A: Yes. We have another equivalence relation. If `e` is any expression, then `x => e(x)` is equivalent to `e`.

B: That surprised me. Why is that?

A: Let's consider a simple example. Let `e` be the function `y => y * y` here, then we have `x => (y => y * y)(x)` is equivalent to `y => y * y`.

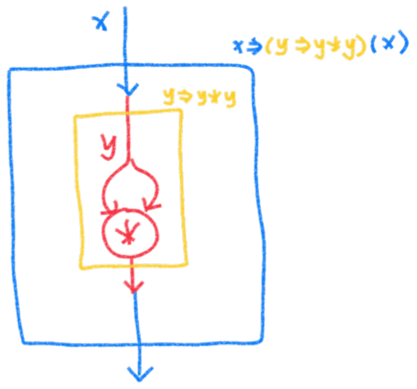
B: Are they really equivalent?

A: You may give them some inputs and see if they give you the same outputs.

B: I tried some numbers, 2, 3, 7, ... Indeed they gave me the same outputs 4, 9, 49, ... So they are both computing the square of the input.

A: Yes. Now you may draw a computation graph. It may help you understand why they are equivalent.

B: Here is my computation graph.



I can see that the input of the `x => (y => y * y)(x)` function is directly passed to `y => y * y` and the output of `(y => y * y)(x)` is the output of `x => (y => y * y)(x)`. So they always have the same inputs and outputs.

A: Right. You may consider this as just juicers with extension pipes attached. It's still the same juicer.

B: Now it's easy to see.

A: The `e` in `x => e(x)` can be any expression we have learned so far. It can be functions, variables, or even calls.

B: Is it very useful?

A: I can't say "very", but it is useful sometimes. It may appear when you need to simplify expressions containing functions. You will see examples in the exercises.

B: Great!

Alternative syntax of functions

A: We are almost done with this lesson. Before I give you exercises, I need to let you know an alternative syntax for functions. It is provided by JavaScript and many other languages.

There is a simpler way to write named functions.

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

This is equivalent to

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

B: Let me try to see the relationship between the two. I see the function names, parameters and body correspondingly, only the syntax is different.

A: Right. Also please notice that inside the function body's curly braces, you may have more than one statements. You may introduce new local variables inside the braces. For example

```
function f(x)
{
  var y = x * x;
  return y + 1;
}
```

This equivalent to

```
var f = x =>
{
  var y = x * x;
  return y + 1;
}
```

B: I didn't know that I can write curly braces in the "arrow notation" too.

A: Yes you can, although it's not often written that way. If the function has a name and has multiple statements in the body, it is usually written in "function notation".

Another thing about this "function notation" is that you must write the keyword `return` for the output, otherwise you will get an unexpected *undefined* value, which means you haven't returned anything.

B: That seems a little different from the "arrow notation".

A: Yes, but only a few nonessential differences. You will see both kinds of 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. Think independently. This will deepen your understanding.

B: Okay.

A: Do the exercises one by one and send the solution to me as soon as you finish each one. Don't wait until all is done. 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. *(Exercises omitted for the sample)*

Ground-Up Computer Science

Chapter 2

(December 20, 2021)

Yin Wang

2 Recursion

A: How was the exercises?

B: They are hard, but interesting!

A: Don't feel frustrated. They are just warm-ups. The following exercises may be easier.

B: They are like fun puzzles. I sweated, but not frustrated.

A: Good warm-ups. Today we will proceed towards the concept of *recursion*, which is very important to computer science and math.

B: I heard of recursion, but never really learned it.

A: Recursion is a deep topic. Few programmers learned recursion in its full strength. Having been using it for many years, I still occasionally have new discoveries of recursion.

B: That sounds profound.

A: Don't worry. We will digest it in small pieces.

B: Good.

Boolean and string data types

A: To introduce recursion, we will first introduce a new concept called *conditional branch*, or just *branch*. But conditional branch depends on a new data type called *boolean*, so we will look at *boolean* first.

B: It seems that there is a dependency *boolean* -> *conditional branch* -> *recursion*. But first may I ask, what is a *data type*?

A: Let me explain this with examples. For example, *number* is a data type. We used numbers in the first lesson. The expressions `2`, `3`, `2 * 3`, `1 + 2 * 3`, ... Their values are all numbers.

B: Right.

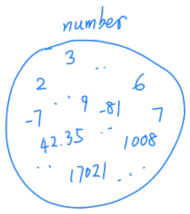
A: We call those expressions `2`, `3`, `2 * 3`, `1 + 2 * 3`, ... *number typed* expressions.

B: Because their values are numbers?

A: Right. Is `square(3)` is also a number typed expression.

B: Yes, because the value of `square(3)` is 9, which is a number.

A: Excellent. You may think of the data type *number* as a bag containing all numbers.



B: I saw this kind of picture before. It looks like a *set* in math.

A: It seems that you have learned the concept *set*. Yes, you may think of a data type as a set.

B: But I didn't learn much about sets. Actually I forgot most of it.

A: Don't worry. Think about the concept "clothing", which includes all kinds of clothes. You may think of *clothing* as a set. Similarly, when we talk about "humans", we mean the set containing you, me and all other people. *Human* is also a set.

B: A set feels like a bag of things.

A: Yes, but this "bag" is not physical. It does not physically contain things. It is something abstract. It lives in our head.

B: Now I have a better understanding of the word *abstract*.

A: To get a feel of the data type *boolean*, try this expression

```
2 > 3
```

B: I got `false` from console.

A: `2 > 3` is false, right?

B: Right. `2 > 3` is not true.

A: Try `2 < 3` then.

B: It says `true`, because `2 < 3` is truth.

A: `2 > 3` and `2 < 3` are called *comparison expressions* because they compare two values. The value of comparison expressions are either `true` or `false`.

B: I noticed that `2 > 3` and `2 < 3` look very much like `2 * 3`. There are `2` and `3`, and an operator (`>`, `<` or `*`) in the middle.

A: Good observation. Actually there is not much difference between `>`, `<` and `*`. We call `>`, `<` and `*` *binary operators* because they have two inputs. The difference is that the outputs of `*` are numbers, but the outputs of `>` and `<` are booleans. We also have `+`, `-`, `/` etc. They are all *binary operators*.

B: Are there other comparison expressions?

A: Yes. You may try these:

```
2 <= 3          // 2 is less or equal to 3
2 >= 2          // 2 is greater or equal to 2
3 == 3          // 3 is equal to 3
2 != 3          // 2 is not equal to 3
2 * 3 == 6      // 2 * 3 is equal to 6
3 == 4 - 1      // 3 is equal to 4 - 1

var y = 2 * 3;
y < 8           // y is less than 8
```

B: What are those double slashes `//` ?

A: The characters after the double slash `//` are called *comments*. JavaScript will ignore whatever you write after `//` until end of the line, so you can write additional information there to explain the code.

B: For in-class experiments, do I just type the parts that are not comments?

A: Yes. You may read the comments, but you don't need to type them into console. Now try them and let me know what are the values of those *comparison expressions.*

B: They are either `true` or `false`.

A: Here `true` and `false` are called *boolean* values. They are values of the *boolean* data type. We call `true`, `false`, `2 < 3`, `3 == 3` etc. *boolean typed* expressions.

B: Why do we write `==` for equality? Why not just `=`?

A: This is because the single equal sign `=` has already been used for variable definitions, for example `var x = 2 * 3`, so we have to use some other symbol for equality. JavaScript chose to use `==`, so did many other languages.

B: I see. `=` and `==` look quite similar.

A: Yes, so you must be very careful. Never write `=` when you mean `==`, otherwise you may cause dangerous *bugs* (computer term for mistakes).

B: What bugs can this cause?

A: It is better to think about this yourself and let me know your thoughts.

B: *(Discuss with your teacher about this.)*

A: There are no other values in the *boolean* data type, except `true` and `false`.

B: That sounds reasonable, because a comparison is either `true` or `false`.

A: Actually we can say "boolean type" for short of boolean data type. Can you draw a picture of the *boolean* type, similar to the picture of the *number* type?

B: Here it is, a bag containing just `true` and `false`.

A: Very good. Boolean is essential for conditional branches. That is why I introduce them first.

B: What is a *conditional branch*?

A: Let me show you by example. The following `abs` function can compute the *absolute value* of the input `x`. Its function body contains a *conditional branch* statement, marked in orange.

```
function abs(x)
{
  if (x < 0)
  {
    return -x;
  }
  else
  {
    return x;
  }
}
```

B: I can see that.

A: You may think of the orange part in English: "**If** (`x` is less than 0), **then** {the value of `abs(x)` is `-x`}, **otherwise** {the value of `abs(x)` is `x`}."

B: That makes more sense now. It is not very different from English.

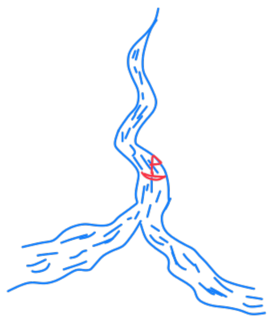
A: Constructs in programming languages are very similar to natural languages. Programming languages are just more precise in syntax.

B: All languages are similar. Some are more precise than others.

A: Right. Depending on whether `x < 0` is true or false, this conditional branch statement may execute either `return -x` or `return x`, but not both.

B: Only one branch is executed?

A: Yes. It is like branches of a river. A boat can go down only one of the branches, never both.



B: That is clear. The condition is how we decide which branch to take?

A: Correct. The general form of a conditional branch is like this:

```
if (condition)
{
  // code to be executed when condition is true
}
else
{
  // code to be executed when condition is false
}
```

Here *condition* must be a boolean typed expression.

B: Like `x < 0`, `2 >= 3`, `t == 2 * 3` etc?

A: Right. This is why I told you about the *boolean* type.

B: Now these are connected.

A: We use the value of the condition to decide which way to go, so the condition must be evaluated before we take one of the branches.

B: There seems to be an order of evaluation. Some expressions must be evaluated before others.

A: Yes. We have a similar evaluation order in the variable definition `var x = 2 * 3;`, where `2 * 3` must be evaluated before we create the variable `x`.

B: I also remember that we must first evaluate the operands of a function call.

A: Right. We will reinforce your understanding of evaluation order later, so don't worry if you can't remember them now.

B: Okay.

A: You can create functions that return boolean values too. For example, you can define a function which will return `true` if the input temperature (`temp`) is over 30 celsius.

```
function hot(temp)
{
  return temp > 30;
}
```

B: That is understandable, because `temp > 30` is similar to `temp * 2`, which is just a normal value, so we can use it as output. Can I write something like `if (hot(42)) { ... } else { ... }` ?

A: Yes. The condition can be any boolean typed expression, including `hot(42)`.

B: This is like standardized machines. I put smaller components together to make bigger components.

A: Yes. This way of building things by combining smaller pieces is called *modular design*. You don't take apart the components after they are built. You just put them together.

B: This is like building cars or airplanes.

A: Exactly the same idea, very clever! Now we go on. We will write a bigger function with three branches in it. This function `sign` will return the sign of its input number. For example:

sign(-4) returns -1

sign(4) returns 1

sign(19) returns 1

sign(-28) returns -1

sign(0) returns 0

Can you see the pattern?

B: `sign` returns -1 for negative numbers, 0 for zero, and 1 for positive numbers.

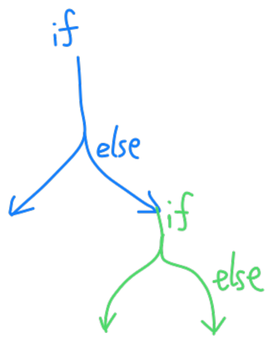
A: Yes. Now write this function using conditional branches.

B: I have some trouble here

```
function sign(x)
{
  if (x < 0)
  {
    return -1;
  }
  else
  {
    // ...
  }
}
```

The conditional branch statement can have only *two* branches, but here we have *three* cases: negative, zero and positive.

A: Here is a hint: you may have another conditional statement inside the branches of a conditional statement. For example, you may put it into the else branch.



B: Nice. A branch of a river can have branches too. This means I can write this?

```
function sign(x)
{
  if (x < 0)
  {
    return -1;
  }
  else
  {
    if (x == 0)
    {
      return 0;
    }
    else
    {
      return 1
    }
  }
}
```

A: Excellent. When you write a conditional statement inside a branch, you get three branches.

B: Yes. One of the outer branch is split into two, thus totally three branches.

A: But this way of writing three branches looks complicated. When we have many branches the code will be hard to read.

B: Are there clearer ways?

A: There is a special syntax rule which we may use here. The rule is that whenever you have an *if* immediately inside the *else* branch, you may omit the braces for the else branch.

For example, in the above code you may delete these braces (marked).

```
  if (x < 0)
  {
    return -1;
  }
  else
  {
    if (x == 0)       // branch immediately inside else
    {
      return 0;
    }
    else
    {
      return 1
    }
  }
```

After that, you may put the second `if` next to the `else`, so it looks like "`else if`".

```
  if (x < 0)
  {
    return -1;
  }
  else if (x == 0)
  {
    return 0;
  }
  else
  {
    return 1
  }
```

B: Indeed that looks prettier, and it is clear that we have three branches.

A: But remember that this simplification requires that no other code goes between `else` and `if`, otherwise you have to write the braces.

B: Got it.

A: So we have finished learning *conditional branches*.

B: There are four basic building blocks now. *Variable*, *function*, *call* and *conditional branch*.

A: Yes. Just four of them. Now we proceed and use them to write programs that were not possible before.

B: Exciting!

fact: a recursive function

A: Remember that our main topic today is recursion, so we proceed to write our first recursive function.

B: Okay.

A: Do you know the *factorial* function?

B: I learned factorial in math class, sort of.

A: Don't worry. I will show you by example:

The factorial of 5 is `5 * 4 * 3 * 2 * 1`.

The factorial of 4 is `4 * 3 * 2 * 1`.

The factorial of 3 is `3 * 2 * 1`.

And so on...

B: I see it now. The *factorial of n* is the product of every number from 1 to *n*. It is written as *n!* in math, so *5!* means 5 * 4 * 3 * 2 * 1.

A: Notice that *n!* is in math language. In JavaScript *n!* is written as `fact(n)`, which is a function call.

B: Can I think of math's factorial operator *!* as a function with one parameter.

A: That is exactly what it is. If `!` were a variable name in JavaScript, then we would write *n!* as `!(n)`. Unfortunately we can't use `!` as a variable name in JavaScript, so we write `fact(n)` instead.

B: I think `fact(n)` looks better than `n!` or `!(n)` because I can immediately know that it is a function call.

A: I think so too. With this preparation, now we will write a function which computes the factorial of `n`. It should be something like this:

```
function fact(n)
{
  // TODO
}
```

Don't write it yet. I will first give you some guidance.

B: Okay.

A: Written as math, we know that

5! = 5 * 4 * 3 * 2 * 1

4! =   4 * 3 * 2 * 1

Is it true that *5! = 5 * 4! ?*

B: Yes.

A: Is it true that *4! = 4 * 3! ?*

B: Yes.

A: Is it true that *n! = n * (n - 1)!* for any natural number n?

B: Yes.

A: Notice that when we say *natural numbers*, we include zero. How about zero?

B: Ah, I was wrong. For zero we can't have *n! = n * (n - 1)!* because then *(n - 1)!* would be *(-1)!*. Factorial is not meaningful for negative numbers.

A: What is the value of *0!* then?

B: Zero?

A: Actually *0!* is defined to be 1 in math.

B: Oh? How can *0!* be 1?

A: This is for simplicity. If we define *0!* as 1, then we can have *1! = 1 * 0!*, which is in the form *n! = n * (n - 1)!*. This can simplify our reasoning.

B: I don't see how it is simpler.

A: Think about it this way:

1. For zero, we have *0! = 1*.
2. For any number except zero, we have *n! = n * (n - 1)!*.

If we defined *0!* as 0, then we need one more special case for 1 here.

B: I see. If we define *0!* as 1, then we have just two cases to think about.

A: We can use a conditional statement to represent the above two cases.

B: Two branches, two cases.

A: Right. Can you translate the above math definition of factorial into JavaScript?

B: Here it is

```
function fact(n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return n! = n * (n - 1)!;
  }
}
```

A: That is not right. "*We have n! = n * (n - 1)!*" doesn't mean that you just write `return n! = n * (n - 1)!`. Do you see the problem?

B: Oh, I forgot. *n!* is math's language. In JavaScript we write `fact(n)` for *n!, so (n - 1)!* should be written as `fact(n - 1)`. The last line should be

```
return fact(n) = n * fact(n - 1);
```

A: This is still not correct. Remember that after the `return`, you should write an expression that is the *output value* of the function, but `fact(n) = fact(n - 1)` is not even a valid expression in our language. It is a syntax error.

Think about this carefully. What should be `fact(n)`'s output in the recursive case?

B: `return n * fact(n - 1)`.

A: Right. Don't be misled by math's language. Now write the function in full.

B: Here it is.

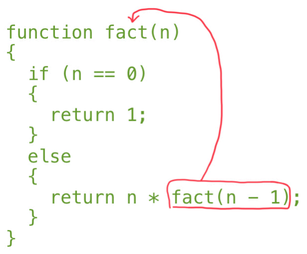
```
function fact(n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return n * fact(n - 1);
  }
}
```

A: Correct.

Anatomy of a recursive function

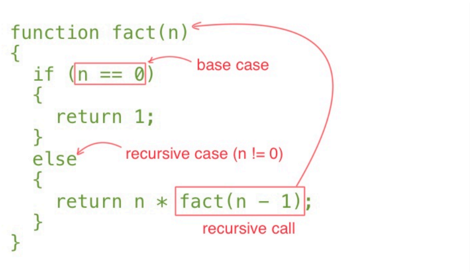
A: Take a careful look at `fact`. How is it different from the functions in Lesson 1?

B: It calls itself inside its own function body, like a cat chasing its own tail.



```
function fact(n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return n * fact(n - 1);
  }
}
```

A: Nice analogy. The concept "calling oneself" is what we call *recursion*. We call `fact` a *recursive functions*, and the call `fact(n - 1)` a *recursive call*. Take a look at this picture.



```
function fact(n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return n * fact(n - 1);
  }
}
```

B: One picture is worth more than a thousand words.

A: The branch without recursive call is called a *base case*. Here `fact` has one base case (`n == 0`), but other functions may have more than one base cases.

B: What do you call the branches with recursive calls?

A: They are called *recursive cases*.

B: I can see a recursive case in the picture. Must we have both a *base case* and a *recursive case* in a recursive function?

A: Yes. A base case is where the function stops calling itself, so you must have at least one base case, otherwise the function will keep calling itself and go into *infinite loops*.

B: Can we demonstrate how this could happen?

A: Yes, you can. Try deleting the base case of `fact`, leaving only the recursive case. Then do substitutions repeatedly on `fact(5)`.

B: The erroneous `fact` function looks like this:

```
function fact(n)
{
  return n * fact(n - 1);
}
```

(Write your substitution of `fact(5)` using the erroneous definition, and send it to the teacher.)

A: Now you can clearly see that you must have a base case.

B: Yes.

The evaluation process of fact(5)

A: Let's take a closer look at the process when we evaluate a call to fact. We use a small example `fact(3)`. Always use small examples just enough to show the ideas, because complex examples often confuse us.

B: Also they are more work to do.

A: You may use our old friend substitution for this. First do a substitution of `fact(3)`. Of course this time we use the correct definition of fact.

B: Like this?

```
if (3 == 0)
{
  return 1;
}
else
{
  return 3 * fact(3 - 1);
}
```

A: For our purpose, you may further *reduce* this. For example, here `3 == 0` must be false, so you can just go to the else branch and have `3 * fact(3 - 1)`. Because you know `3 - 1` is 2, you can reduce the whole thing to

```
3 * fact(2)
```

This simple form will aid our thinking.

B: I see. I don't need to show every tiny step. I see you used the word *reduce*. Does that mean the expression gets smaller?

A: Yes, usually smaller. You can imagine something shrinking. Now you can write out the substitutions step by step.

B:

```
fact(3)
3 * fact(2)
3 * 2 * fact(1)
3 * 2 * 1 * fact(0)
3 * 2 * 1 * 1
6
```

A: Don't go directly to 6 in the end. There are three multiplications in `3 * 2 * 1 * 1`. Which one should we compute first?

B: `3 * 2` ?

A: No. Think about it this way. Why did we get `3 * 2 * 1 * 1`? Because we wanted to compute `fact(3)`, which is substituted to `3 * fact(2)`. But until we have the value of `fact(2)`, we can't compute the multiplication `3 * fact(2)`.

B: This means we must compute `3 * fact(2)` last?

A: Actually, if you put parentheses on multiplications in your substitutions, you will see the order clearly.

B: I just put every multiplication into parentheses. Indeed this is a lot more clear.

```
fact(3)
(3 * fact(2))
```

```
(3 * (2 * fact(1)))
(3 * (2 * (1 * fact(0))))
(3 * (2 * (1 * 1)))
```

A: Right. Is that interesting?

B: Very interesting. I didn't know that parentheses can help so much. It is now all clear without much effort.

A: Remember this trick. Sometimes writing out every parentheses explicitly can help. Take a look at the evaluation process. Do you see any problems?

B: As I can see, the intermediate expressions can get very long.

A: Yes. Consider computing `fact(1000)`. The intermediate steps will grow to 1000 in length, `1000 * 999 * 998 * ... * 3 * 2 * 1 * 1`.

B: Does it take space to store this expression?

A: Yes. Nothing comes for free. Every number has to be stored somewhere, otherwise we would not know what to multiply.

B: That seems to be a big problem.

A: This way of writing `fact` function is not *efficient* regarding to storage space, but our purpose is just to show the anatomy of recursive functions, and `fact` serves as a very good first example. In practical programs, you don't want to write it this way.

B: I will remember `fact`, our first etude of recursive functions.

Another recursive function (fib)

A: Now we take a look at another recursive function `fib`, where `fib(n)` will compute the nth fibonacci number.

B: Fibonacci number. I heard of it too, but...

A: Let me remind you. The first fibonacci number is 0, the second fibonacci number is 1, then every next fibonacci number is the sum of the previous two fibonacci numbers.

B: Can you give me an example?

A: Here is the fibonacci sequence:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

B: I see, in math language,

fib(0) is 0

fib(1) is 1

fib(2) = fib(0) + fib(1), which is 1

fib(3) = fib(1) + fib(2), which is 2

fib(4) = fib(2) + fib(3), which is 3

fib(5) is fib(3) + fib(4), which is 5

fib(6) is fib(4) + fib(5), which is 8

fib(7) is fib(5) + fib(6), which is 13

fib(8) is fib(6) + fib(7), which is 21

... ...

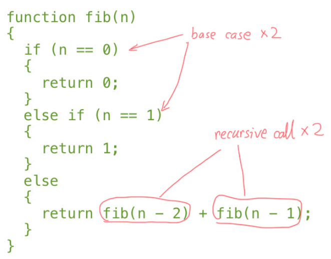
A: Yes. Using experience of `fact`, can you write a recursive function `fib`, which computes the nth fibonacci number?

B: Here it is.

```
function fib(n)
{
  if (n == 0)
  {
    return 0;
  }
  else if (n == 1)
  {
    return 1;
  }
  else
  {
    return fib(n - 2) + fib(n - 1);
  }
}
```

A: Perfect. Can you tell me how many *base cases* and *recursive calls* are there?

B: Two bases. Two recursive calls.



```
function fib(n)
{
  if (n == 0)
  {
    return 0;
  }
  else if (n == 1)
  {
    return 1;
  }
  else
  {
    return fib(n - 2) + fib(n - 1);
  }
}
```

A: Nice picture. Why do we have two base cases?

B: I'm not sure. We are given the first two numbers. They are not computed but just given, so I used them as base cases.

A: Think about this. We need to compute the other numbers from the previous *two* numbers. The recursion will end up needing the values of `fib(0)` and `fib(1)`, so you can't have just one base case.

B: Is there a systematic way in which I can know how many base cases to write?

A: The trick is to look at the recursive calls. See how the recursive call's input parameters are different from the function's parameters, and then figure out the values in which they end up.

B: I don't understand this.

A: It is a bit early to explain this in full. We will come up with a systematic way of thinking about recursion in the next class after we have more ways to write recursive function. For exercises of this class, you may just mimic `fact` and `fib`.

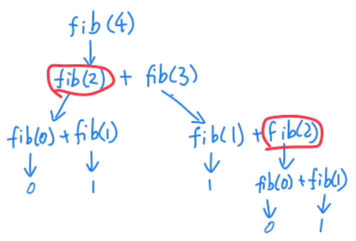
B: Okay.

A: Can you see that our way of writing the `fib` function has a big problem?

B: Does it also take a lot of storage space, like `fact`?

A: Not exactly the same problem. You can use `fib(4)` as an example, try drawing a graph showing the substitutions of the recursive calls. Draw an arrow connecting *each* recursive call to its substitution.

B: I can draw it like this. Because there are two recursive calls, the graph seems to be "branching" into a tree.

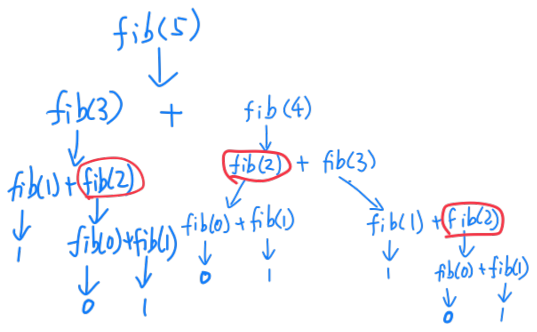


A: Good observation. Can you see how many times `fib(2)` is evaluated?

B: Twice. I have put two circles on them.

A: Right. Try expanding this graph into a graph for `fib(5)`, and again see how many times `fib(2)` is evaluated.

B: Three times. The number of times we evaluate `fib(2)` seems to be growing with bigger `fib(n)`.



A: Can you see the problem?

B: Yes. We are wasting time by computing some expressions repeatedly.

A: Have you noticed other such repeated computations?

B: Yes. `fib(3)`, `fib(1)`, `fib(0)`, ...

A: Now that we see the examples of `fib(4)` and `fib(5)`, try to figure out a general formula in which the number of repeated evaluation of `fib(2)` grows with `n`. For example, is it $2n$, n^2, or 2^n ? If you have no clue, try extending our previous examples. Draw a graph of `fib(6)`, `fib(7)`, and so on.

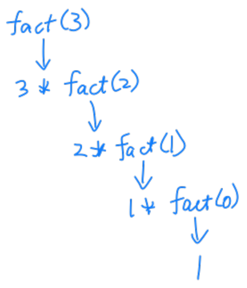
B: *(Write your answer and send it to the teacher.)*

A: So `fib` is a very slow way of implementing fibonacci numbers, but it demonstrates this type of recursion very well.

B: What type of recursion? How is it different from `fact`?

A: Try redraw a picture for `fact(3)`, in the style you just did for `fib(4)`.

B: Here it is. I can see that the graph for `fact(3)` has no branching. It is more like a chain, not a tree.



A: Yes. The recursion of `fact` is called *linear recursion*, while the recursion of `fib` is called *tree recursion*, because it looks like a tree with branches.

B: I can see a downward growing tree in `fib`'s recursion process.

A: Why is `fib` a tree recursion?

B: The branches come from `fib(n - 2) + fib(n - 1)`, so the reason of tree recursion seems to be in the number of recursive calls.

A: Good observation. Whenever you see two recursive calls in the same recursive case, that means a tree recursion.

B: That is useful observation, because tree recursions may be costly.

A: Tree recursion may not always be costly, and it is often the only way to write certain programs.

B: Some teachers told me that recursion is slow and should be avoided or converted to other ways, for example *loops*.

A: I don't want to teach "what is a loop" for now, but it is a usual misunderstanding of recursion. Recursion is usually not any slower than loops. If you can't write a fast program with recursion, then you can't write a fast program with loops or any other way either.

But recursion is usually a lot simpler than other ways, so it is not something you should avoid. You just have to master the idea. It is fundamental to computer science, math and so many natural phenomenon in the universe.

B: What a terrible misunderstanding. I can see how much those people have missed. I will do my best to master recursion.

A: Good. We have more recursive functions in later classes. Actually almost every function we will write from now on will be recursive functions!

B: Wow!

A: Okay. That's all for today's class. Here are the exercises.

B: Thank you. Have a good night!

Ground-Up Computer Science

Chapter 3

(March 11, 2023)

Yin Wang

3 Lists

A: Do you remember *pair*?

B: Yes. It is an exercise in Lesson 1.

A: Today we are going to use *pair* to create something very useful.

B: I was puzzled by *pair* although I finished the exercise. Would you give me a review?

A: Yes, I will explain it first. A solid understanding of *pair* is very important, because *pair* is the most fundamental *data structure*.

B: I should pay close attention then. But may I ask what is a *data structure*?

A: In the physical world we have *structures* like houses, roads and bridges. In the virtual world inside the computer, we have *data structures*. We use data structures to build up a virtual world inside the computer.

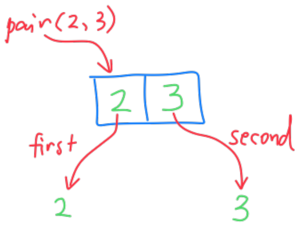
B: What is the role of pairs in this virtual world?

A: Pairs are basic building blocks, much like bricks in the physical world. Basic building blocks are very important because everything inside the computer is built with them. We should have a good understanding of pairs.

B: Okay!

A definition of pair

A: A *pair* is like a box with two compartments. We create a pair by putting two values into the compartments. We can take the two values out individually.



B: This reminds me of my contact lens case.

A: Good analogy, but there is a difference. We reuse the same contact lens case many times, but we never reuse pairs. Using pairs is like using a new disposable case every time.

B: That seems wasteful. Where do those used pairs go?

A: They will be recycled diligently by the computer, so there is no waste. There is time and energy cost for this, but the logic will be very clear this way. In this course we almost never reuse pairs or other data structures.

B: Are all programs this way, without reusing anything?

A: No. Practical programs often reuse data structures, but for pedagogical purposes we don't reuse them here. We think more clearly when nothing is reused.

B: I'm afraid this way will not fit into my future everyday programming.

A: The ideas you learn here is very important. They can be used directly, and they can be adapted easily when you need to reuse data structures, so don't worry.

B: That sounds good.

A: Let's be concrete. In the pair exercise, we defined these three functions:

```
function pair(a, b)
{
  return select => select(a, b);
}

function first(p)
{
  return p((a, b) => a);
}

function second(p)
{
  return p((a, b) => b);
}
```

B: Yes. I wrote `first` and `second` for the exercise, but I need to understand them again.

A: Let us try to understand how they work with the help of our friend substitution. Now, try a substitution of

```
var p1 = pair(2, 3);
```

B: Okay. The function body of `pair` is `select => select(a, b)`. I replace `a` with 2, and `b` with 3, and it becomes `select => select(2, 3)`. Place this back into place, I got

```
var p1 = select => select(2, 3)
```

A: Good. Tell me, what is `p1`'s type?

B: `p1` is a function, but I don't understand the purpose of `select => select(2, 3)`.

A: Can you see that this function `select => select(2, 3)` somehow has 2 and 3 inside its body?

B: Yes. 2 and 3 were inserted by substitution.

A: By inserting 2 and 3 into the function body, substitution effectively puts 2 and 3 together into a "box". Now do a substitution of `first(p1)`, step by step.

B: Okay.

```
1. first(p1)
2. first(select => select(2, 3))
3. (select => select(2, 3))((a, b) => a)
4. ((a, b) => a)(2, 3)
5. 2
```

(Please follow this and understand each step.)

A: Do you understand the meaning of this process?

B: Somehow `first` takes out the first thing inside `p1`, which is 2.

A: That may look like magic, but think about step 3 above. It looks like this pattern:

```
(select => select(2, 3))((a, b) => ____)
```

In the blank, we may fill in any code that refers to `a` and `b`, and this code will operate on 2 and 3.

B: I see, because this `((a, b) => ___)` will be called with 2 and 3 as inputs.

A: Very good. Actually `first` and `second`'s definitions are just special cases of this pattern, where the blank is filled with `a` and `b` respectively.

B: I can see that, `(a, b) => a` and `(a, b) => b`.

A: To enhance your understanding, can you fill in some code in `(select => select(2, 3))((a, b) => ____)` and compute the sum of 2 and 3?

B: I can just fill in `a + b`.

```
(select => select(2, 3))((a, b) => a + b)
```

Then substitution will give me

```
((a, b) => a + b)(2, 3)
2 + 3
5
```

A: Using the same process, you can understand `second`.

B: Yes. `first` and `second` can get the two values out of the pair.

Abstraction

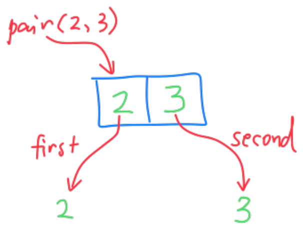
A: From now on, we need to forget about the details inside `pair`, `first` and `second`.

B: Forget about them? But we just spent time understanding them.

A: It is very important not to think about the implementation details after we convinced ourselves that they work properly. This is called *abstract thinking*. This is like you have examined the internals of your juicer or TV, then you just use them without thinking about the pipes, motors or circuits inside.

B: I see. Otherwise I will be overwhelmed by details.

A: We usually think about pairs in an *abstract* way. We stop thinking about `select => select(a, b)`. We just think about the three functions `pair`, `first` and `second`, as in this picture.



B: Indeed, I don't see `select => select(a, b)` in this picture. I only see `pair`, `first` and `second`.

A: Those functions can be called *interfaces* of pair.

B: Interface, the word seems intuitive.

A: Although `select => select(a, b)` is how our pair is implemented, pairs are not always implemented as functions. Actually we have better ways to implement pairs.

B: Why do we use functions here then?

A: Just to show how powerful functions can be. This definition of pair doesn't even depend on computers. Actually this definition dates back to the 1920s when there are no computers.

B: This is math?

A: Or call it *logic*. Computer science can be probably called *applied logic* because it originated from logicians.

B: Wow, I didn't know that. Good to know.

A: Let's get back. A call to `pair(2, 3)` will create a pair containing 2 and 3, and `first` and `second` can take them out. From now on, we can just use `pair`, `first`, `second` and forget about implementation.

B: Are these three functions all we need for pairs?

A: Oh, actually there is one more.

B: What is it?

A: After we create the pairs, we need a way to know whether something is a pair or not.

B: But I know it is a pair.

A: For example, the input to a function can be sometimes a pair, and sometimes not a pair, so we have to ask questions about it.

B: We can ask questions by using `if (...)`.

A: Yes. So we need a function `isPair` which can tell us if something is a pair.

B: How can we tell pairs from other functions since our pairs are just `select => select(a, b)`, which is just a function?

A: As long as we don't put other functions into pairs, they won't get mixed, so we may just ask if something is a function. If yes, then we think it is a pair. This is not accurate, but good for our purposes.

B: How do we know if something is a function?

A: The `typeof` operator of JavaScript can tell the type of a value, so we can write `isPair` this way.

```
function isPair(x)
{
  return typeof(x) == "function";
}
```

If the input `x` is a function, then `isPair(x)` returns `true`. otherwise it returns `false`.

B: This is a fuzzy, but it can distinguish pairs from other types such as *number* and *boolean*.

A: Yes it is fuzzy, but this way of making things all by ourselves proved to be very successful in understanding them.

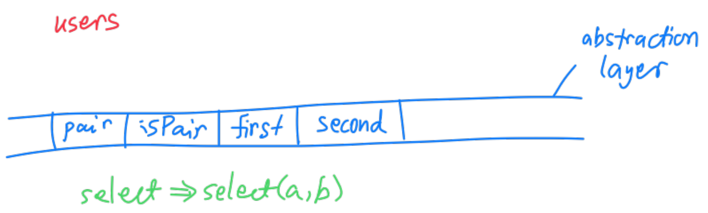
B: Nice.

A: To enhance your understanding of `typeof`, you may try these examples of `typeof` in console.

```
typeof(2)
typeof(2 * 3)
typeof(2 < 3)
typeof(false)
typeof(x => x * x)
typeof(pair(2, 3))
```

B: I have tried them all. The results are type names as strings.

A: Good. Now we may just use the four functions `pair`, `isPair`, `first`, `second` and can forget about pair's implementation. We call `pair` the *constructor* of pair, `isPair` the *type predicate*, `first` and `second` the *visitor* of its members. The set of these four functions are called *abstraction layer* or *interface* of pair.



B: I see there are three kinds of interface for pair:

1. constructor
2. type predicate
3. visitor

A: Yes. Some of the categories have more than one function, but it is good to know there are three kinds of them.

B: I can see that.

A: Please note that above are just our own terminologies for this course. Other people may not call them this way. I'm not into jargons, but these terms will make things easier to talk about.

B: I like this way of creating terminologies on the fly.

A: So much for abstraction. Let's move on.

B: Okay.

Lists

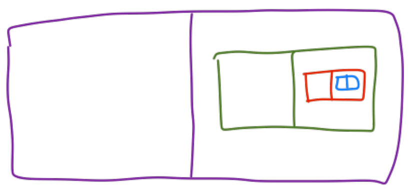
A: I guess you have now understood the definition of pair. Pair is a fundamental *data structure*. It is very important. We can use pairs to create complex and interesting structures.

B: What is a more complex data structure?

A: Actually every data structure in this course will be built upon pairs, as you will see. Next we will create lists.

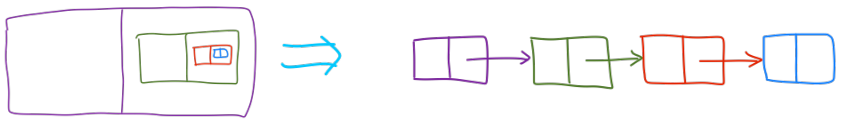
B: Great. Let's start!

A: What is interesting about a box is that it can contain another box, and this inner box can contain yet another box, and so on.



B: Box inside box inside box... This reminds me of some kind of Russian dolls.

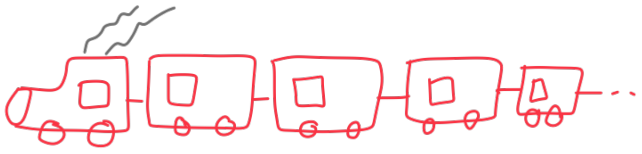
A: It is hard to look at boxes this way because they get small quickly, so we draw the contents on the outside and point to them with arrows.



B: This looks like a chain. Why do we put the inner boxes in the second parts only?

A: Because we want to store useful things in the first parts, thus we can create a train containing lots of things.

B: This can be a very long train.



A: How do we know it is the end of the train?

B: When the second part doesn't point to another pair?

A: Yes. If the second part points to another pair, then the train will continue.

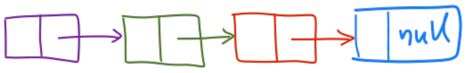
B: What do we put into the second part of the last pair then?

A: We don't put useful things in the second part of the last pair. We put in a special value which means "nothing" or "empty", much like a notice "END of TRAIN".

B: How do we make the notice?

A: There is a special value called `null` in JavaScript and many other languages. When we put `null` into the `second` part of the last pair, it means *THE END*.

B: Like this?



A: Yes. This structure is called a *list* in computer science.

B: It does look like a list of things.

A: Whenever we see the `second` part of a pair is `null`, we know this is the end of the list.

B: Yes, `null` means *THE END*.

A: Actually we think of `null` itself as a list too. `null` is the *empty list*.

B: It is lists inside lists, until we get to the *empty list*.

A: We can have a mathematical definition of lists. It has the following rules:

1. `null` is a list.
2. If the `second` member of a pair is a *list*, then this pair is also a list.

B: I noticed that you used the word *list* inside list's own definition. This feels like recursive functions.

A: Indeed this is very much related to recursive functions. List's definition is a *recursive definition*. List is a *recursive data type*.

B: What is the relationship between *recursive definitions* and *recursive functions*?

A: Recursive data types need recursive functions to process them.

B: No doubt both names have *recursive* in it. This seems to be a profound connection.

A: Yes, there is a deep connection. You will see that the recursive functions have similar structures to the recursive data types they process.

B: I'm amused by this thought.

A function for lists: length

A: List is a very useful data structure. Now we write some functions to process lists. The first function is called `length`. It calculates a list's length.

B: What does *length* mean for lists?

A: Let's think about this. What is the length of `null`?

B: 0, because `null` contains nothing.

A: Correct. What is the length of `pair(2, null)`?

B: 1, because it contains one piece of data.

A: Right. We can call the data inside the list *members*. What is the length of `pair(2, pair(3, null))`?

B: 2.

A: And the length of `pair(2, pair(pair(3, 4), null))`?

B: 3.

A: Wrong.

B: What?

A: `pair(3, 4)` is considered to be *one* member (not two) of the list, because it takes only one place in the list.

B: It seems that we count the number of cars in the train, no matter how many things each car may contain.

A: Exactly. Last question, what is the length of `pair(2, pair(3, pair(4, null)))`? Notice this is a different list.

B: 3.

A: Good. Now we write this function. It should be a recursive function. Do you see why?

```
function length(ls)
{
  //...
}
```

B: Because list is a recursive data type, and recursive data types need recursive functions to process.

***Step 1**: write the recursive call.*

A: Yes. We need a *systematic* way of creating recursive functions. First, let's think about how the *recursive call* should be written. Do you remember what is a recursive call?

B: A *recursive call* is a call to the function itself inside its own definition.

A: Right. Because a recursive call is a call to the function itself, so it should take the *same type of input* as the function.

B: You mean it should take lists as input?

A: Yes. The recursive call to `length` should take a list as input, because the `length` function takes a list as input.

B: This fact seems obvious, but easy to forget.

A: Yes, so pay more attention to this idea. Many of people's trouble with recursive functions is that they neglect this obvious fact.

B: What is the consequence of the fact that we should give `length` a list as input?

A: The consequence is that we need to find some other list from its original input `ls`. Now find inside `ls` something that is also a list, but slightly smaller.

B: I can naturally think of `second(ls)`.

A: Bingo. This is our input for the recursive call.

B: So the recursive call can be written as `length(second(ls))`?

A: Right. Can you see that is a legitimate call to `length`?

B: Yes, because `second(ls)` is also a list.

A: Our partial result is like this

```
function length(ls)
{
  if (...)
  {
    // base case
  }
  else
  {
    return length(second(ls));
  }
}
```

B: Are we finished with the recursive case?

A: Actually no, because `length(second(ls))` is not a correct result of `length(ls)`. Can you see that?

B: Yes, `length(second(ls))` is less than `length(ls)`. Their difference is 1.

***Step 2**: construct the recursive case result from the recursive call.*

A: Starting from the recursive call, we need to think how we can get the correct result of the function. `length(second(ls))` is certainly not a correct result of `length(ls)`.

B: How about `1 + length(second(ls))`? Addition of 1 would make up the difference.

A: Good. Now you can finish the recursive case.

B:

```
function length(ls)
{
  if (...)
  {
    // base case
  }
  else
  {
    return 1 + length(second(ls));
  }
}
```

***Step 3**: derive the base case from the recursive call.*

A: The recursive case is done. Next we need to think about the base case.

B: That seems to be last step?

A: Yes. The base case can be derived from the recursive call. Take a look at the recursive call `length(second(ls))`. If the recursive call happens multiple times, what will its input be?

B: We will use a shorter list as input each time. Finally we will see `null`.

A: Right. We will reach the base case when the input is `null`. Now can you write the base case?

B: When `ls` is `null`, the function should return 0, because the length of `null` is 0.

```
function length(ls)
{
  if (ls == null)
  {
    return 0;
  }
  else
  {
    return 1 + length(second(ls));
  }
}
```

A: The function is all done. As a side note, to think clearly about the base case, sometimes I find it useful not to think about a sequence of recursive calls, but simply a call which takes the base case input. In this case, you may just think about `length(null)`.

B: If I think about `length(null)`, I can immediately see that it should return 0. This is the case even when `length` is not recursive.

A: When writing lists, usually we want to use the name `head` and `tail` instead of `first` and `second`, so we can be clear that we are operating on lists. So our code can be changed this way:

```
var head = first;
var tail = second;

function length(ls)
{
  if (ls == null)
  {
    return 0;
  }
  else
  {
    return 1 + length(tail(ls));
  }
}
```
B: I see. I will use `head` and `tail` for lists.

A: Now we should make some small examples to test our `length` function.

```
var list1 = pair(2, pair(3, null));
var list2 = pair(4, pair(5, pair(6, null)));
var list3 = pair(2, pair(pair(3, 4), null));

length(null);   // should be 0
length(list1);  // should be 2
length(list2);  // should be 3
length(list3);  // should be 2
```

B: I got all of them correct. I also tried some other examples.

A: Good. It is a good habit of make small tests. It is better that the tests execute *every branch* of the code. For example, always remember to test both the base case and the recursive case. Here we have `length(null)` which executes the base case.

B: I see. This will catch possible errors.

A: This function is all done now. Go have a good rest.

B: Will do. Thank you!

Display contents of a list

A: Before we write the next function, we should have a way to display the contents of a list.

B: Can't we just use `console.log`?

A: You may try it first.

B: Oh, `console.log(pair(2, null))` just printed `select => select(a, b)`.

A: You see the problem?

B: I guess this is because our pairs are implemented as functions, so JavaScript has no idea how they are different from other functions and just displays them as functions.

A: Right. Programming languages normally provide functions for displaying simple values, but they can't display complex data structures that the programmer defined.

B: I see, so we have to do this ourselves?

A: We need a function to convert a list into a string, and then we can use `console.log` to display this string.

B: I see.

A: We can call this function `pairToString` because it converts pairs to string. It can be written in the usual recursive way, but it is not a good place to explain it. You will understand how it works in the next lesson. Here is the code, you can just copy and use it.

```
function pairToString(x)
{
  if (!isPair(x))
  {
    return String(x);
  }
  else
  {
    return "("
      + pairToString(first(x))
      + ", "
      + pairToString(second(x))
      + ")";
  }
}
```

B: `show(pairToString(pair(2, null)))` displays *(2, null)*, without the *pair*?

A: Yes, we may choose whichever style to display the data structure. Showing the word *pair* every time is hard to read, so we may omit it. But remember, *(2, null) i*s not something you can type into console. This is not valid JavaScript code.

B: I see. This is just our own way of showing data.

Another list function: append

A: Next, we will write another function `append`. A call to `append(ls1, ls2)` should return a new list whose contents should be `ls1` and `ls2` concatenated together, in that order.

```
function append(ls1, ls2)
{
  // TODO
}
```

B: Can you give me some examples?

A: Okay. First we create two lists.

```
var list1 = pair(2, pair(3, null));
var list2 = pair(4, pair(5, pair(6, null)));
```

Then `append(list1, list2)` should give us `pair(2, pair(3, pair(4, pair(5, pair(6, null)))))`. The content is just the two lists concatenated.

B: That is clear now.

A: Now we start to write this function, using the three-step thinking process.

B: Step 1 is to *think about the recursive call*.

A: Yes. From the two parameters `ls1` and `ls2`, try to find two other parameters that are lists.

B: Both `tail(ls1)` and `tail(ls2)` are lists. I'm not sure if I need both of them.

A: Actually, as long as one of them is different from `ls1` and `ls2`, we may make progress.

B: Then there are three ways we can write the recursive call.

```
append(tail(ls1), ls2)
append(ls1, tail(ls2))
append(tail(ls1), tail(ls2))
```

I don't know how to choose.

A: You may just try them one by one, and see which one works.

B: Okay. I'll start with the first one `append(tail(ls1), ls2)` then.

A: Do you remember Step 2?

B: Step 2, from the recursive call, try to construct a correct return value of the function, then we have a recursive case. But I have no idea how to proceed.

A: Here is the trick. *Pretend that the function is already written*. Try to use the small example we just made, and think about what you get from the recursive call. Just think or draw it on paper. Do not try it on computer.

B: Okay. If `ls1` is `pair(2, pair(3, null))` and `ls2` is `pair(4, pair(5, pair(6, null)))`, then `append(tail(ls1), ls2)` will give me `pair(3, pair(4, pair(5, pair(6, null))))`.

A: How far is `pair(3, pair(4, pair(5, pair(6, null))))` from the the desired result of `append(ls1, ls2)`?

B: The desired result of `append(ls1, ls2)` is `pair(2, pair(3, pair(4, pair(5, pair(6, null)))))`. We need a 2 in the front of it.

A: How can we put 2 in the front of a list?

B: `pair(2, ...)`.

A: Right. Then you can put 2 in the front of the result of the recursive call.

B: That would be `pair(2, append(tail(ls1), ls2))`.

A: What is 2 there? We may not have this as our input.

B: I see, 2 is the head of `ls1`. I should use `head(ls1)` instead, so the recursive case result should be `pair(head(ls1), append(tail(ls1), ls2))`.

A: Very good. That is it.

B: I feel too lucky that this first try worked.

A: To make sure that you don't depend on luck, you should also try other two options for the recursive call. This may make your thinking more reliable.

B: Okay. Let me try the second way, `append(ls1, tail(ls2))`.

A: Use the example lists again.

B: `append(ls1, tail(ls2))` would give me `pair(2, pair(3, pair(5, pair(6, null))))`.

A: For simplicity, when we talk about list's contents, we may just write it in a mathematical way. For example, the above `pair(2, pair(3, pair(5, pair(6, null))))` may be written as (2 3 5 6).

B: Okay. That will make it easier to see.

A: Just make sure that you understand (2 3 5 6) is not actual code that you can write. It is just a representation of the data inside the list.

B: Okay.

A: How can you get our desired result (2 3 4 5 6) from (2 3 5 6)?

B: I have no idea.

A: For lists, we don't really have an efficient way of putting a member in the middle of it.

B: Does this mean `append(ls1, tail(ls2))` is not good?

A: Right. It is not a good way to write this recursive call.

B: I understand now. How about `append(tail(ls1), tail(ls2))`?

A: You can use the examples again.

B: `append(tail(ls1), tail(ls2))` would give me (3 5 6).

A: How can you make it (2 3 4 5 6)?

B: We can use pair to put 2 in the front and get (2 3 5 6), but then this is just `append(ls1, tail(ls2))`, which is no good.

A: Right. So this one is not an option either.

B: Okay. It seems that `append(tail(ls1), ls2)` is the only way to go.

A: Then we may go ahead to Step 3.

B: Okay. The base case.

A: Look at the recursive call `append(tail(ls1), ls2)`, and observe how `ls1` and `ls2` is progressing.

B: Only `ls1` is different from call to call, and we will see `null` at some point, so the base case would be `ls1 == null`.

A: Right. Think about what you should return for the base case?

B: Using our thought trick. I think about `append(null, ls2)`. We should get `ls2` because `ls1` is empty.

A: Correct. Write the function now.

B: Here it is.

```
function append(ls1, ls2)
{
  if (ls1 == null)
  {
    return ls2;
  }
  else
  {
    return pair(head(ls1), append(tail(ls1), ls2));
  }
}
```

A: Very good. Now try it with some examples.

B: Okay. They look correct.

A: Now try to understand `append` again, with our friend substitution. Do a step-by-step substitution of `append(list1, list2)` where list1 is (2 3) and list2 is (4 5 6).

B: *(Write down the steps of substitution and send it to the teacher.)*

A: Now, from the substitution process, tell me what is the number of new pairs created from `append(list1, list2)`.

B: *(Write down the answer and send it to the teacher.)*

A: Very good. Now we write our last function for this lesson. Its name is `nth`. A call to `nth(ls, n)` will give us the `n`th member of `ls`.

B: I guess the function starts with

```
function nth(ls, n)
{
  // TODO
}
```

A: Yes. First, tell me what is the second member of (1 2 3 4)?

B: 2, of course.

A: No. I should tell you that in computer science, we usually count not from one, but zero.

B: Then the second member of (1 2 3 4) should be 3. Counting from zero is strange to me though.

A: There are many benefits of counting from zero, as you will see in your computer science career.

B: I'll keep that in mind.

A: Now, let's start writing this function. The first step is to write a recursive call, remember?

B: This time there are also two recursive types, so I have the options

```
nth(tail(ls), n)
nth(ls, n - 1)
nth(tail(ls), n - 1)
```

A: Good. Try them one by one.

B: I can use the previous example, `nth(list1, 2)` where `list1` is (1 2 3 4). Pretending that we have already implemented `nth` correctly, `nth(tail(list1), 2)` would give me the second member of (2 3 4), which is 4. This is not correct.

A: Is there any way to resolve this?

B: I don't think so. There is no way we can get the correct member starting from 4, which is already an end answer.

A: Right. There is no way to fix this. What's next?

B: `nth(list1, n - 1)` would give me `nth(list1, 1)` which is 2. This is also wrong and there is no way to fix it. So the only choice would be `nth(tail(ls), n - 1)`.

A: Correct. So we are comfortable with our choice `nth(tail(ls), n - 1)`. Next step, to think about how we can get correct answer for the function from the recursive call.

B: `nth(tail(list1), n - 1)` would give me the *1st* member of (2 3 4), which is 3, and this is exactly the answer to `nth(ls, n)`. I feel strange to say "1st", but this is the correct answer.

A: Then what do we do?

B: It seems that I can just use `nth(tail(ls), n - 1)` as the recursive branch's value. It counts on the tail of `ls`, but the index is smaller by one, so this will produces exactly the same answer to `nth(ls, n)`, always.

A: Yes, so we have the recursive case.

B: The function looks like this. We only have to find the base case.

```
function nth(ls, n)
{
  if (...)
  {
    // TODO: base case
  }
  else
  {
    return nth(tail(ls), n - 1);
  }
}
```

A: Yes. Go on.

B: The repeated recursion of `nth(tail(ls), n - 1)` leads us to two situations.

1) `tail(ls)` will eventually get to `null`.
2) `n - 1` will eventually get to `0`.

This seems to be a new situation we haven't seen before.

A: Right. Think about what to do this time?

B: I would use both of them as base cases. Something like this:

```
function nth(ls, n)
{
  if (n == 0)
  {
    // base case 1
    return head(ls);
  }
  else if (ls == null)
  {
    // base case 2
    // Don't know what to do...
  }
  else
  {
    return nth(tail(ls), n - 1);
  }
}
```

A: For the case `ls == null`, the function does not make sense for any `n`, right?

B: Yes. `nth(null, n)` is meaningless even when n is 0.

A: When the function does not make sense for certain cases, we should not return a value, but need to report an error.

B: How do we report an error?

A: There is another construct in JavaScript we can use for errors. You may use throw. For example,

```
throw "something wrong";
```

This will print a message on the screen and terminate the program immediately.

B: The structure of this `throw` statement looks just like a `return` statement. Why can't I write this

```
return "something wrong";
```

?

A: Remember that this function may have a caller, if you were to use return, then the caller would get the string `"something wrong"`, and the execution will continue with this string, as if this is a normal value it gets from `nth`.

B: So the program will not terminate immediately?

A: Right. It may continue with this string, but this is wrong. The string may go somewhere else and cause deeper troubles.

B: I see. This is why we need to terminate the function and not return the value to the caller.

A: Yes. Think about the difference, and you will appreciate this thought later in your career. You will make better decisions in the programs you engineer.

B: Thank you! Here is the finished code.

```
function nth(ls, n)
{
  if (n == 0)
  {
    // base case 1
    return head(ls);
  }
  else if (ls == null)
  {
    // base case 2
    throw "The input list is empty";
  }
  else
  {
    return nth(tail(ls), n - 1);
  }
}
```

A: Quite good, but your message is a bit misleading here.

B: What is wrong with it? It just says what the condition says: `ls == null`.

A: Remember that this is a recursive function. It may have reached `null` after several recursive calls to itself, so the original input may not be `null`.

B: I see.

A: Can you see how it can get to this branch when the original input is not `null`?

B: When `n` is large enough so the list `ls` gets to the end.

A: Yes. In this case, what would be a meaningful error message?

B: "Index out of bound"?

A: Perfect. But we are not done yet.

B: Is there anything more I need to change?

A: There is still a bug in the code.

B: Where is it?

A: Think about the call to `nth(null, 0)`. What will your function do?

B: Because `n` is 0, it goes into the first branch, and it tries to get head(null), which is meaningless.

A: Right. Your program will crash because `head` is not meaningful for `null`. This is a serious bug.

B: I see. I should test whether `ls` is `null` in the first base case.

A: You could test that in the first base case, but have you noticed that you tested it in the second base case?

B: Yes. What does this mean then?

A: This means you may just need to think about the order of the two base cases. If you test `ls == null` as first base case, would that make things easier?

B: This is interesting. If I test `ls == null` in the first base case, then in the second base case, `ls` cannot be null. This is because the later branches can be reached only when the previous conditions are all false. And this means I can safely say `head(ls)` there.

A: This is good. Try that.

B: Here it is

```
function nth(ls, n)
{
  if (ls == null)
  {
    throw "Index out of bound";
  }
  else if (n == 0)
  {
    return head(ls);
  }
 else
  {
    return nth(tail(ls), n - 1);
  }
}
```

A: Very good. Now make some examples to test it.

B: Okay. Here they are

```
nth(pair(2, null), 0)  // should be 2
nth(pair(1, pair(2, pair(3, pair(4, null)))), 2)  // should be 3
nth(pair(1, pair(2, pair(3, pair(4, null)))), 4)  // should
throw error
nth(null, 0)  // should throw error
nth(pair(2, null), 1)  // should throw error
```

A: Good. I see you are quite careful about the tests, and they covered all the possible conditions.

B: Thank you.

(Write more of your own tests. If you found some of them interesting, please send them to the teacher.)

A: This is all for this lesson. Here are the exercises.

(Exercises omitted for sample version.)

Ground-Up Computer Science

Chapter 4

(January 22, 2022)

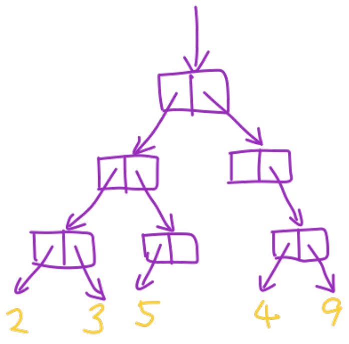
Yin Wang

4 Trees

A: Today we make a small extension on *lists*, and we can have *trees*.

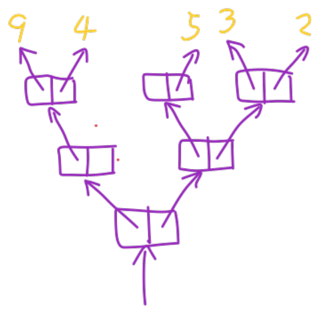
B: What are *trees*?

A: For example, here is a tree. It is built with pairs.



B: Pairs again. They are really powerful. This does look like a tree, except that it grows downwards.

A: We could draw the tree structure in the other direction, then it will look exactly like a tree.



B: Indeed, then why do we draw it upside-down?

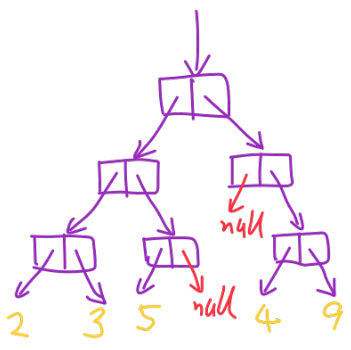
A: Computer science is different from art. We often don't know how tall the tree is. If we draw it as a usual tree, we won't know where to put the root. So we usually draw the root first and let the tree "grow" downwards. This is more convenient.

B: I see.

A: Can you see how similar they are, the tree data structure and the real tree? They both have *branches* and *leaves*. The pairs are the branching points, and the numbers are leaves. We call the pairs *internal nodes* because they are in the middle of the tree.

B: It seems that some pairs have only one branch growing from them?

A: Yes. Where I haven't drawn branches, you may think there is a `null`. We use `null` to mean that there is no left or right branch.



B: I have seen `null` in the previous lesson when we learned lists. We use `null` to mean the empty list. It seems that `null` has a similar meaning in trees too.

A: Yes, they are very similar. In lists, `null` means an *empty list*. In trees, `null` means an *empty tree*. `null` signifies the end of a list or a tree.

B: So the two `null`s have different meanings. Is it possible that we get confused because both data structures have `null` in them?

A: It is possible. We often use the same `null` to mean different things in computer programming. Some people have complained about this.

B: Can we do something different?

A: The important thing of this lesson is the tree data structure, so I will try not to use too much abstraction. But to be clear about the meaning, we can use a variable `emptyTree` instead of using `null` directly. We just define `emptyTree` to be `null`.

```
var emptyTree = null;
```

B: This makes me feel better, although for the computer they are the same.

A: Yes, the purpose is on the human side. Let's see how lists and trees are related. Lists are constructed with pairs whose `second` parts are also lists. The `first` parts are considered to be data (members). When we do recursion on lists, the recursive calls only go into the `second` parts. This is why lists have a *linear* shape. If we relax these restrictions, we have trees.

B: Can we be more clear about the restrictions?

A: Let me rephrase that. There are two restrictions on lists.

1. The `second` parts can only be lists.
2. Recursive calls only apply on `second` parts and don't go into the `first` parts.

B: If we remove these restrictions, does that mean

1. The `second` parts can also contain members.
2. Recursive calls may go into `first` parts. If `first` parts are pairs, we look deeper into them.

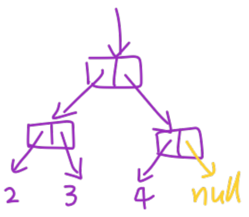
A: Correct. The difference between a list and a tree is partly about the structure, and partly about how we process them.

B: I sort of see this second point. Even some lists contain pairs in their `first` parts, they are still lists, just because we don't treat them as trees. For example, `pair(pair(2, 3), pair(4, null))`.

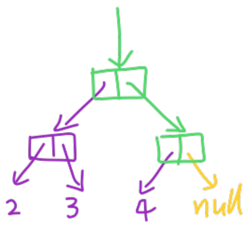
A: Right. Members of lists can be pairs, but if we don't consider those pairs as the structure of the data, then they are still lists.

B: So the difference between lists and trees somewhat lies in how we *look* at a structure, a subjective matter.

A: Indeed this may depend on how you look at the structure. For example, the following tree can also be thought of as a list.



B: That is interesting. If I just follow the `second` parts, I will reach `null`. This would make this tree also a list whose first member is `pair(2, 3)` and the second member `4`.



A: Yes, as long as we don't have recursive calls into the `first` part, we have a list.

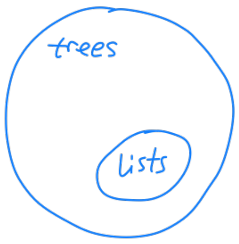
B: So some trees can be thought of as lists. Can lists be thought of as trees too?

A: Actually every list can be thought of as a tree.

B: Every one of them? Let me think... Indeed, lists are just trees where the left branches are all leaves, and the last pair has no right branch.

(Think about this and discuss with the teach if you have questions.)

A: Every list is also a tree, but not every tree is a list. So we can have picture like this:



B: In math, this means that lists are a *subset* of trees.

A: The trees presented in this lesson are quite restricted, because the *internal nodes* don't contain data members. Have you noticed that?

B: Indeed, there aren't any number (data) in the middle of the trees that we have seen so far. All numbers are at the leaves. The internal nodes only points to internal nodes or leaves. Can we also have data in the internal nodes?

A: Yes we can, but we will wait until the next lesson. This simpler tree structure can help you understand the recursion pattern on them. We only need a small extension to our way of doing recursion on lists.

B: Nice. I like baby steps.

A: I'm glad you understand this. Don't go too fast. Now we can start writing our first recursive function on trees.

B: Good.

A: Actually you have already seen such a function. The `pairToString` function you used in last lesson is a recursive function on trees. We will take a look at it first.

B: Okay. I have the code here.

```
function pairToString(x)
{
  if (!isPair(x))
  {
    return String(x);
  }
  else
  {
    return "("
      + pairToString(first(x))
      + ", "
      + pairToString(second(x))
      + ")";
  }
}
```

A: Take a look at the code. How many *recursive calls* are there?

B: Two of them. One of them is recursion on `first(x)`, the other on `second(x)`.

A: Good. Take a look at the list functions you wrote for the previous lesson, have you ever did recursion on the `first` parts?

B: No. All the recursive calls are on the `second` parts.

A: This is because the `first` parts are considered to be data and not structure, so you don't do recursion on them.

B: I see.

A: Now you may understand `pairToString` using this difference.

B: This is easy. I can see that it does recursive calls on `first` and `second` parts, converts them into strings, inserts a comma in between, and put parentheses around the whole thing.

A: How about the base case? Can you figure out how we arrive at the base case, using our three-step method of thinking about recursion?

B: I figure out the base case by looking at the recursive calls. There are two recursive calls `pairToString(first(x))` and `pairToString(second(x))`. Both of them eventually reach a leaf node.

A: Right. Leaf nodes are not pairs, so we use the condition `!isPair(x)` to distinguish them. This case also includes `emptyTree`.

```
if (!isPair(x))
{
  return String(x);
}
```

B: For the base case's return value, I guess `String(x)` is JavaScript's way of converting values into a strings?

A: Yes, but `String(x)` can only meaningfully convert basic data recognizable by JavaScript, that is, numbers, booleans, strings etc.

B: Can't I use `String(x)` to convert pairs into strings?

A: If you use `String(x)` on a pair, you will get something like "[function]" or "select => select(a, b)", just as if you use `console.log` on the pair. Pair is our own *custom defined* data structure, so JavaScript has no idea what is in there. It only knows that it is a function. So we have to convert pairs to strings ourselves.

B: Got it. We have to write a function like `pairToString`, which is a recursive function on trees.

A: Right. This is why we need `pairToString`. We will need to write these "toString" functions for any custom defined data structure, if we ever want to display them.

B: I see.

A: This is pretty much all you need to do the exercises for trees, because trees are just simple extension to lists.

B: Nice. Are there a lot of exercises?

A: Almost the same amount as lists. For almost every list function, there is a tree function which does a very similar thing.

B: That is good amount of exercise. Can I relate the functions together?

A: Yes. I gave similar names to them. For example, when there is `listEqual`, you have `treeEqual`.

B: Nice.

A: Here are the exercises. Exercise Set 4. (Exercise omitted for the sample)

B: Thank you!

Ground-Up Computer Science

Chapter 5

(Dec 23, 2022)

Yin Wang

5 Calculator

A: How was the tree exercises?

B: They are just slightly different from the list exercises.

A: That is good. Their close relationship can help you understand the ideas.

B: In the trees of previous lesson, we can't have data members in the internal nodes. Can we have them in this lesson?

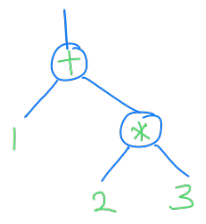
A: Yes, we will put some data members in the internal nodes. We will make something interesting out of this.

B: What is that?

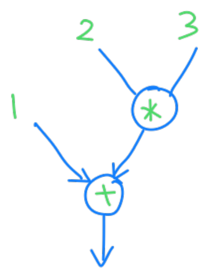
A: Actually, you have already seen trees with data members in the internal nodes before.

B: Really? I don't recall anything like that.

A: How about this one?



B: Oh, that looks like the *computation graph* we learned in the first lesson, except that it is upside-down.



A: It is the computation graph. Does the orientation matter at all?

B: No. We have drawn trees upside-down in the previous lesson. It doesn't matter in what direction we draw it. It is still the same thing.

A: This is a tree with two data members "+" and "*" in the internal nodes.

B: Are "+" and "*" strings?

A: Yes. The operators `"+"` and `"*"` are represented as strings here. It is not necessary to represent operators as strings, but for simplicity we just use strings here.

B: We just put operators in the internal nodes, and the tree becomes a computation graph. Baby steps can go a long way.

A: Yes. Can you see how we can construction this tree, the computation graph?

B: Previously we used pairs as internal nodes. If we use pairs, we have no place to store the data members, so I guess we need something larger.

A: Instead of simple pairs, we can use lists as internal nodes, then we can have data members in the internal nodes.

B: That makes sense.

A: The internal nodes can be as simple as `pair("*", pair(2, pair(3, null)))`.

B: Then the whole computation graph of `1 + 2 * 3` can be written as `pair("+", pair(1, pair(pair("*", pair(2, pair(3, null))), null)))`?

A: Right. But this is hard to read.

B: Yes, it will be even harder with complex expressions.

A: We can apply the idea of *abstract data type* again. We call this data type `binop`, which means "binary operation". A `binop` contains three members, an operator and two operands. The two operands can be `binop` themselves.

B: Why didn't we use abstract data type for the trees of last lesson?

A: That is a good question. Because our pairs happen to create trees, we didn't bother to create an abstract data type for those trees. But this time it is a bit different. We can no longer use pairs as internal nodes.

B: I see. Because `pair("+", pair(1, pair(pair("*", pair(2, pair(3, null))), null)))` is too complex and error-prone. We need to abstract those details out.

A: Right. Let's look at the abstract interfaces of `binop` one by one then. First, the constructor. Think about this, what should the constructor of `binop` create?

B: It should return a list containing three members, an operator and two operands. The operands can be `binop`'s themselves.

A: Correct. Our first attempt is something like this:

```
function binop(op, e1, e2)
{
  return pair(op, pair(e1, pair(e2, null))):
}
```

B: We just put the three members into a list.

A: But if we just write the constructor this way, then we would have trouble distinguishing `binop` from other kinds of lists.

B: Why do we need to distinguish them?

A: For example, when you write a recursive function on a computation graph, you will need to ask whether the input is an internal node (`binop`) or not.

B: How can we tell `binop` from other kinds of lists?

A: For this purpose, we can put a special string member `"binop"` into the list, like this.

```
function binop(op, e1, e2)
{
  return pair("binop", pair(op, pair(e1, pair(e2, null))));
}
```

Whenever the first member is the string `"binop"`, we think this list is a `binop`. This is like putting tags on things. We call this string a *type tag*.

B: What if there are people who happen to create a list with such a tag, but they don't mean the same thing?

A: This method is not for reliable prevention of accidents. It is just demonstrating a general idea how we distinguish data types. We just put some special tag in there, just like we put tags on products in a store.

B: That sounds like a good idea, inspired by everyday life.

A: Actually *type tags* are not necessarily strings. They can be any value as long as it is unlikely some other people come up with the same value.

B: I could think of a better type tag, such as `"binop$62A4#E91"`.

A: Good. Actually you may generate a random number which is very unlikely to be used by other people. That will be much more reliable. But for demo purposes we don't use complex schemes.

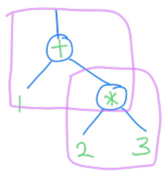
B: Okay. It is good to have ideas first, and refinements can come later.

A: So we have our constructor for `binop`. Can you come up with the *type predicate* and *visitors* yourself? You can call the type predicate `isBinOp` and the visitors `binopOp`, `binopE1` and `binopE2`.

B: *(Write your answers and send them to the teacher.)*

A: Well done. After having the `binop` node structure, we can use it to construct our previous examples for `2 * 3` and `1 + 2 * 3`. You may find how much clearer they are. If you are not sure, take a look at the following picture and see where are the `binop` structures.

B: *(Write your answers and send them to the teacher.)*



A: Good. Now we can think about what we can do with the computation graphs constructed with `binop`.

B: From the first lesson, we know that we can compute their values. That is, to *evaluate* them.

A: Evaluation of computation graphs is our final goal, but first let's do something simpler than that. We build some small utility functions which can make `binop` more convenient to use.

B: What are the utility functions?

A: First of all, computation graphs constructed by `binop` is still not easy to read. It is better if we can display them as usual math expressions such as `2 * 3` and `1 + 2 * 3`. We call these *infix notations* because the operator `*` and `+` are placed in the middle of the two operands.

B: That seems to be a nice thing to have.

A: We call this function `toInfix`. Using our abstract data type, you can proceed to Exercise Set 5 and work out `toInfix`. To avoid the problem of *operator precedence*, we put parentheses around all subexpressions, so we have `(2 * 3)` and `(1 + (2 * 3))`.

B: *(Write your answer to* `toInfix` *and send them to the teacher.)*

A: Now we have a function that can display the `binop` structure as an usual math expression. With this example, I hope you see the difference between *computation graphs* and *text expressions*.

B: We have talked about this in the first lesson. Now I can see that they are really different things. `binop("*", 2, 3)` constructs the *computation graph*, and `toInfix` function converts it into a *text expression* `"2 * 3"`.

A: The computation graph is *structural*, which has a clear structure. It is easy to extract its parts with visitor functions (`binopOp`, `binopE1` and `binopE2`). After it is transformed into text by `toInfix`, it is much harder to get the parts out in a meaningful way.

B: I can see that the string is just a series of characters with no clear division of structure. For the expression `(1 + (2 * 3))`, it is quite hard to extract the two operands `1` and `(2 * 3)` even with the parentheses. The computer only sees the sequence of characters `(`, `1`, `+`, `(`, `2`, `*`, `3`, `)`, and `)`.

A: We use `toInfix` only for displaying the computation graph to humans. Inside computer programs we almost never use the text format. So be sure not to use `toInfix` unless you need to display on the screen.

B: Got it.

A: There is a meaningful (but difficult) way to extract parts from text expressions. You can transform the string back into a graph by using a *parser*. A parser is a function which transforms a text expression into a computation graph.

B: So a parser is like the inverse function of `toInfix`? Can we write a parser?

A: Yes, the parser and `toInfix` are the mutual inverse functions. We don't write a parser now. Parsers are complex and difficult functions to write. It is better you finish all the exercises of this lesson before attempting to write a parser. I will give you a parser exercise if you do well in this lesson.

B: That is nice!

A: Next we write a function similar to `toInfix`, except that it produces *prefix notations*. If the infix notation is `(2 * 3)`, then the prefix notation is `(* 2 3)`. If the infix notation is `(1 + (2 * 3))`, then the prefix notation is `(+ 1 (* 2 3))`. You see what is going on?

B: Yes. The prefix notation just puts the operator before the two operands.

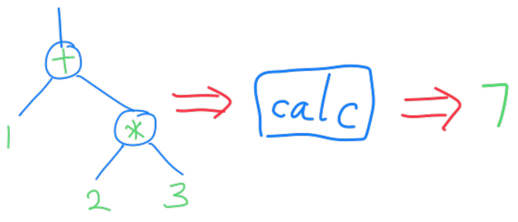
A: We call this function `toPrefix`. This is also in Exercise Set 5. You can do it now.

B: *(Write your answer to* `toPrefix` *and send them to the teacher.)*

A: Okay. Two warm-up exercises done, we can proceed to write the calculator.

B: Great!

A: The *calculator* (`calc`) is a function from computation graphs to values. It belongs to a general category of functions called *interpreters*. You can think of the calculator as an interpreter for a simple language with only arithmetic expressions, like `2 * 3` and `1 + 2 * 3`.



B: I have heard of *interpreters* before, such as the "JavaScript interpreter".

A: Interpreters are functions that execute your programs. A JavaScript interpreter can execute programs written in the JavaScript language. There are multiple implementations of JavaScript interpreter. For example, the JavaScript interpreter we use for this course is called *node.js*.

B: I see. So `calc` is also an interpreter, except that it runs programs written in a much smaller language, just arithmetic expressions.

A: Right. We will extend this small interpreter to something bigger and more interesting in the next few lessons.

B: Nice.

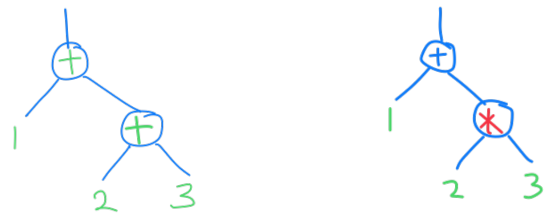
A: Now we start to write the `calc` function. It is quite easy. It is just a little different from a function you wrote in the last lesson.

B: Which function?

A: `treeSum`. The `calc` function is very much like `treeSum`.

B: How are they similar?

A: Think about this, the `treeSum` function just sums up all the numbers in a tree. This is as if you have a computation graph where all the internal nodes have the operator `"+"`. Compare the following two computation graphs.



B: I see, `treeSum` only do additions, so we don't need operators in the internal nodes. For general arithmetic expressions, every internal node may have a different operator, so we need to do different computations depending on the operator.

A: That is a crucial observation. I believe you can figure out what to do given this similarity.

B: *(Write your answer to* `calc` *and send them to the teacher.)*

(Exercise omitted for the sample.)

Ground-Up Computer Science

Chapter 6

(May 1, 2021)

Yin Wang

6 Lookup Tables

A: We are done with the calculator.

B: Can we start writing the interpreter now?

A: Not yet. We have an important data structure missing. Once we have it, we will be ready to write the interpreter.

B: What is the data structure?

A: It is called a *lookup table*. Actually we have two such structures. One of them is a list, the other is a tree. They fulfill the same purpose, so they are exchangeable.

B: What does a *lookup table* do?

A: A *lookup table* is like a restaurant menu. You look it up for the price of an item. What is the price of steak?

pizza	128
cake	46
pasta	68
steak	258
salad	45
beer	35

B: 258.

A: Have you got the idea? We call `"steak"` the *key*, and `258` its *value*. You lookup the key for its value.

B: I see. What has this to do with interpreters? The keys feel like variables.

A: You are ahead of me. In the interpreter we use lookup tables to find values of variables. But for demo purposes we just use them to make restaurant menus here.

B: Foods are good examples.

A: Lookup table is a very useful data structure. It is the core idea behind *databases*. Database people usually call it *key-value store*.

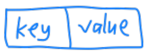
B: I always thought databases are complicated monsters.

A: If the product looks simple, they can't ask for a huge price for it.

B: That is funny!

A: Let's start building our first lookup table. Look at the menu. It has multiple rows. Each row has two columns, the key and its value. We can use known data structures to represent the rows and columns.

B: I guess each *key-value pair* can be represented as a *pair*?

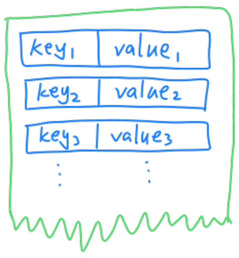


A: That is reasonable. You just need another structure to contain the pairs.

B: I can use a list.

A: Why do you use a list?

B: Because the table has a linear shape in the vertical direction. It looks like a list.



A: Yes, you can use a list. Another reason to use the list is because the list is a *recursive data structure*, so it can contain arbitrary number of rows.

B: I haven't thought of this. What does "arbitrary number" mean?

A: "Arbitrary number" means that the number is not a predetermined constant, for example 6. You may have different menus and they may have different lengths, but we have only one definition for the lookup table data structure.

B: I see. One definition for multiple lengths. That is why we have the `length` function for lists, because each list may have a different length.

A: There is one more thing that may help our understanding here. We can define a variable `emptyTable`, whose value is `null`. It is much like `emptyTree`. Use `emptyTable` for the end of the table instead of `null`. This may improve understandability of code.

B: Okay. I'll keep that in mind.

A: Now you may construct the menu with the "pairs inside a list" structure. Write the code for constructing the menu. Run it and display the table with `pairToString`. You should be able to see the menu.

B: Don't we define an abstract type for the lookup table?

A: This structure is so simple, so we skip that part this time. We just need to understand the idea. For actual engineering projects, it is usually better to define an abstract data type even for the simplest structures.

B: *(Write your answer for constructing the menu and send it to the teacher.)*

A: Let's call this `menu1`. Create a variable for it. Now, write a function `lookupTable` which finds the value for a given key. It should be a recursive function on lists.

```
function lookupTable(key, table)
{
  // TODO
}
```

After you have this function, you may test it with `menu1`. Just lookup some keys and make sure you get correct values.

B: *(Write your answer and send it to the teacher.)*

A: Very good. We need another function `addTable`, which will construct a *new* table based on an existing table. It will put a new key-value pair on top of the old table, thus creating a new table.

```
function addTable(key, value, oldTable)
{
  // TODO
}
```

You can now open Exercise Set 6. There are some more tests in there. Make sure they all pass and understand their implications.

B: *(Write your answer and send it to the teacher.)*

A: Well done. Do you have any questions?

B: I see that you emphasized that `addTable` creates a *new* table. Why is that?

A: Because `addTable` creates a new table. It doesn't modify `oldTable`. This is like all other list functions you have written so far. They never change their input lists but just create new lists. The new lists may reference old lists, so we don't need to copy a lot of data.

B: Since the old table is not modified, can we still find the old value of the key in the old table?

A: Yes, you can find the old value because the old key-value pair is never deleted. There is a test for this. Please study the test carefully and understand why it behaves that way.

B: I have studied it. Indeed, when I lookup the same key in new and old tables, `lookupTable` gives me different values. How is this useful?

A: This can be quite useful. Have you noticed that the old data is automatically "backed up" whenever you update it?

B: Yes, that is interesting. It behaves like a *version control system* such as Git.

(You may ignore the following conversations if you haven't heard of Git or blockchains.)

A: Actually Git uses the same idea, except that Git uses a tree, not a list. We have already built similar trees. The tree functions in this course don't change their inputs either.

B: Good to know. Are there other things built with this idea?

A: Have you heard of *blockchains*?

B: Yes. That is a hot topic recent years, but I have never understood how blockchains work.

A: After this lesson you may have ideas how to build those things. They are just slightly more complex.

B: That is so nice. New doors opened for me.

(Ignorable parts end here.)

A: Now we are done with the linear lookup table, built as lists. Let's proceed to something more advanced. It is called a *binary search tree* (*BST* for short). It serves the same purpose as a lookup table.

B: How is the *BST* more advanced?

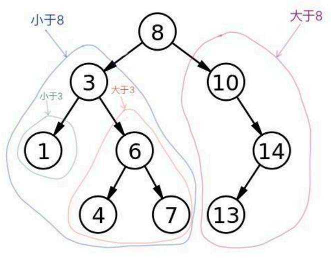
A: It is usually more efficient, so you can find the values quicker using less energy.

B: Nice. How do we use less energy?

A: Do less work.

B: Okay.

A: A BST looks like this. The numbers in the circles are keys. I haven't shown the values but they are also in the nodes.



B: How is this different from a usual tree with numbers in internal nodes?

A: Look at the tree. It has this property.

For every key k in the internal node, every key in the left subtree is less than k, and every key in the right subtree is greater than k.

B: I can see that. This is true for every internal node. 8, 3, 10, 6, 14.

A: Yes, every internal node must have this property, otherwise it will not work.

B: I didn't know that keys can be numbers too. We used strings for the menu.

A: Keys can be any value that can be *compared*. Do you remember that you used the `==` operator in `lookupTable` function for the lookup table? `==` is a *comparison operator*.

B: Yes. I didn't think about this carefully though.

A: Keys can't be something that you can't compare.

B: For example? What can't be compared?

A: For example, the beauty in two paintings, the value in two persons.

B: Oh, that makes sense.

A: For BST, we need to not only use `==`, but also `<` and `>`.

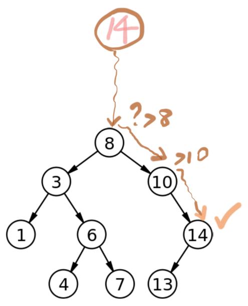
B: How do we use them?

A: Here is how you look for the key *k*. If k is equal to the key of the current node, then you have found it. Otherwise if *k* is less than the current key, then you look for *k* in the left subtree, otherwise you look for *k* in the right subtree. Is that clear?

B: I'm not sure. Maybe we can do an exercise?

A: Okay. Try looking for the key 14 in the previous BST.

B: Since 14 is greater than 8, I look for 14 in the right subtree, and I see 10. Since 14 is greater than 10, I go right again. Then I have found it. The process looks like this picture.



A: Good. Drawing pictures is a good habit. One picture is worth a thousand words.

B: It seems we only visited three nodes of the tree. We always go down from the root, never switch to the other branch.

A: Right. This is why BST is more efficient than linear lookup table.

B: But for the linear lookup table, we may also skip some nodes. If we find the key before reaching the end, we just return it without looking further.

A: That is the case. But for BSTs, the number of visits can be much smaller. Can you see why?

B: I have no idea right now.

A: If the tree is balanced, that is, every internal node has two subtrees, what is the *depth* of a tree with 3 nodes?

B: *(Draw a picture showing the answer, and send it to your teacher.)*

A: What is the depth of a balanced tree with 7 nodes?

B: *(Draw a picture showing the answer, and send it to your teacher.)*

A: What is the depth of a balanced tree with 15 nodes?

B: *(Draw a picture showing the answer, and send it to your teacher.)*

A: Do you see the pattern? What is the depth of a balanced tree with *N* nodes?

B: (Answer the question with a formula containing N, and send it to your teacher.)

A: Very good. What is the average number of visits if you look for a key in a linear lookup table with *N* keys in it?

B: (Answer the question with a formula containing N, and send it to your teacher.)

A: Compare those two formulas, you can see a big difference. The logarithm function is very slow growing, so the BST has the potential of skipping a lot more nodes because you just need to travel the tree's depth.

B: I can see that from the picture. I just go from the root of the tree to a leaf. Sometimes I don't even need to go all the way to a leaf because the key appears in an internal node.

A: This is why we often use trees in computer science. It is probably the most used data structure.

B: Really?

A: There are more complex data structures, but they are seldom used, so trees are considered most useful.

B: Good to know that.

A: Balanced trees are very efficient, but for learning purpose, we will not write balanced trees because they have quite some complexity. Our simple BSTs may not be balanced, but they are often quite efficient too.

B: Okay. I will be happy enough with the basic BST.

A: Similar to `emptyTable`, we have a variable `emptyBST`. Its value is also `null`. Instead of using `null` directly, we use this variable.

```
var emptyBST = null;
```

B: I see. This is just to avoid confusion.

A: Yes. Now we can start to build BSTs. First we write a constructor function for the internal nodes. This node structure is very much like `binop`, except that it contains a key and a value, instead of an operator.

B: So the internal node has four parts -- key, value, left subtree and right subtree?

A: Correct. Four parts. Now write this constructor function. It is called `bst`. It should be easy because you have already written so many constructors.

```
function bst(key, value, left, right)
{
  // TODO
}
```

B: *(Write your answer and send it to the teacher.)*

A: Now we need another function `addBST` which constructs a new BST with a new key-value pair. `addBST` is very much like `addTable`. In fact they have exactly the same parameter pattern, except the last parameter means a different structure.

```
function addBST(key, value, node)
{
  // TODO
}
```

B: I see. `addTable` extends a table, and `addBST` extends a BST, but they serve the same purpose, that is, extending the structure to have one more key-value pair.

A: Because of this similarity, we may even exchange them freely with the use of abstraction. You may see this in the interpreter exercises.

B: Looking forward to the interpreter lesson.

A: Using `addBST`, you should be able to conveniently construct the previous restaurant menu, starting from `emptyBST`. You may construct it now, call it `bstMenu1`. Remember to use `emptyBST` instead of `null`.

pizza	128
cake	46
pasta	68
steak	258
salad	45
beer	35

B: *(Write your answer and send it to the teacher.)*

A: Next function to write is `lookupBST`. It just implements the lookup process that you have described before. `lookupBST` should be a recursive function.

```
function lookupBST(key, node)
{
  // TODO
}
```

B: *(Write your answer and send it to the teacher.)*

A: Now we are done with lookup tables and BSTs. Next lesson we can write an interpreter.

B: Great!

(Exercise omitted for the sample.)

Ground-Up Computer Science

Chapter 7

(Feb 27, 2022)

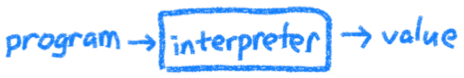
Yin Wang

7 Interpreter

A: Today we will write an interpreter.

B: I often heard of interpreters, for example *JavaScript interpreter*, but I don't really know what they are.

A: An *interpreter* is what executes your programs. All programs in this course are executed by a *JavaScript interpreter*.



B: This looks like the console. I enter an expression and it gives me its value.

A: The console contains an interpreter inside, so it is basically an interpreter.

B: Is JavaScript an interpreter?

A: No, JavaScript is a programming language. You need to distinguish a *language* and its *interpreter*. Those are two different things.

B: So JavaScript is a language as English is a language. Can you give me an example of an interpreter?

A: For example, *node.js* is an interpreter. Node.js is a *JavaScript interpreter*. This means that it can execute programs written in the JavaScript language.

B: I see. JavaScript is the language and node.js is its interpreter. Are there other interpreters of JavaScript?

A: There are many JavaScript interpreters. Anybody can write a JavaScript interpreter. Every web browser has a built-in JavaScript interpreter, for example Chrome, Safari, Firefox, Internet Explorer.

B: Why do we have so many interpreters for one language?

A: This is like there are many singers of the same song. Each one is different and nobody is perfect. Every interpreter author tried to improve what they didn't like or doesn't fit their needs, but everybody makes mistakes, sometimes new mistakes, so this goes on forever.

B: That's funny.

A: Even big companies like Microsoft make mistakes, over and over. If we learn from old mistakes, we won't make the same mistakes again.

B: I should learn from mistakes, either made by myself or by others. They are valuable information.

A: I'm glad that you realized this. In case you know what is a CPU, CPUs are interpreters too. They are interpreters implemented in electronic circuits. They execute machine code.

B: I'm surprised. Interpreters can also be hardware?

A: Yes. Computer hardware circuits are not that different from software. Both hardware and software can implement the same logic. I hope you can see their connections from this course.

B: This idea is fascinating! I also heard of GPUs. Are GPUs interpreters too?

A: GPUs are not that different from CPUs, so they are interpreters too.

B: *Interpreter* seems to be a very broad concept.

A: Actually we humans are interpreters too. We are interpreters of natural languages (English, Japanese etc). We *make sense* out of words and sentences, in a similar way as JavaScript interpreters.

B: It seems to go far beyond computers!

A: If you understand interpreters, you may apply the idea to many things, including the design of CPU, GPU, network protocols etc. It is fair to say that interpreters are the essence of computer science, because "interpret" is a synonym of "compute".

B: That makes a lot of sense.

A: If you know how to write interpreters, you have the ability to *implement* programming languages. You may then design your own programming languages. You can also readily understand languages designed by others.

B: I'm glad that I'm getting there so soon.

Datatype definitions

A: Let's get down to earth. In this lesson you will implement an interpreter for a very simple but powerful language. It can execute most of programs you wrote in Lesson 1 and 2.

B: Good. So we already have lots of test cases for it.

A: Indeed, I made some tests using Lesson 1 functions, for example the `compose` function. You may open the exercise document and use it as a reference.

Exercise Set 7

B: Thank you. I opened it.

A: Don't read too much into it yet. Let me explain.

B: Okay.

A: You have already implemented an interpreter. The calculator (`calc`) that you wrote in Lesson 5 is also an interpreter. It interprets a very simple language of arithmetic expressions, such as `1 + 2 * 3`.

B: Yes. We wrote it as `binop("+", 1, binop("*", 2 3))`.

A: Yes. I wrote it in JavaScript syntax `1 + 2 * 3`, but you should understand this as `binop("+", 1, binop("*", 2 3))` in our own language. We will often use JavaScript's syntax to explain things in this lesson because otherwise it will be very verbose.

B: I see. Are we going to extend this language in this lesson?

A: That's the plan. Using the same ideas, you will arrive at a more powerful language. Arithmetic expressions (expressed by `binop`) will be part of the new language, and the calculator will be part of the new interpreter.

B: New language, new interpreter. Languages and interpreters seem to grow together.

A: Yes. It is often a good idea to *grow* a language instead of starting from scratch. You are very likely to make mistakes if you throw everything away.

B: What's new in the new language?

A: We need to add three constructs–variables, functions and calls.

B: The three pillars of programming languages, as we learned from Lesson 1.

A: It's good that you remembered that. Whenever you are lost in programming languages, look for those three things and think in terms of them. You will often find your way. Now let's start with their datatype definitions.

B: Okay.

A: Previously for the calculator, you have defined the `binop` datatype. It enables you to construct arithmetic expressions. You can write `2 * 3` as `binop("*", 2, 3)`.

B: I also can write `1 + 2 * 3` as `binop("+", 1, binop("*", 2, 3))` because the `binop` type can be nested.

A: This time we will create datatypes for variables, functions and calls. They are very much like the `binop` datatype. They can also be nested inside each other. For example, you can nest a `binop` inside a function. Maybe you can figure out what this expression means?

```
fun("x", binop("*", variable("x"), variable("x")))
```

B: It seems equivalent to the JavaScript function `x => x * x`, but it is in our own language constructs. I guess `fun` is the constructor of functions?

A: Right. `fun` is the constructor of functions. We use `fun` because *function* is a JavaScript keyword so we can't use it.

B: This is good too. Functions are `fun`.

A: Can you see that there are variables nested inside the `binop` structure?

B: Yes, I see two variables inside `binop("*", variable("x"), variable("x"))`, which means `x * x`.

A: Notice that we can't write this in the calculator language because it doesn't have variables.

B: I can see how the language has grown.

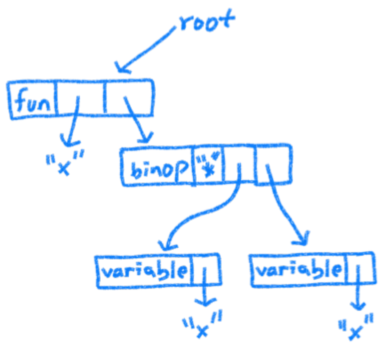
A: Once you have defined the datatypes `variable`, `fun` and `call`, you can construct programs like this. We will reuse the `binop` datatype definition from Lesson 5 so you may just copy it.

B: Since `binop` creates trees, after we have `variable`, `fun` and `call`, are we also creating trees?

A: Yes, programs are trees. We call them *abstract syntax trees* (*AST*). It may be helpful if you draw an AST for this example.

```
fun("x", binop("*", variable("x"), variable("x")))
```

B: I'll try.



A: Very good. Because trees are recursive data structures, you can construct programs of any size.

B: Indeed, trees are simple and powerful structures.

A: Now you may start writing datatype definitions for `variable`, `fun` and `call`. They are similar to `binop`.

B: Okay. I will write `variable`'s constructor first. It takes one parameter `name` which is a string.

```
function variable(name)
{
  return name;
}
```

A: That's not right. You can't just use the string as the variable.

B: Oh, I thought a variable is just a string.

A: If you just return the string, you won't be able to tell our variables from JavaScript strings.

B: It seems even if there is only one member, I still need to return a structure. How about this?

```
function variable(name)
{
  return pair("variable", pair(name, null));
}
```

A: Correct. The type tag will make it recognizable as a `variable`. Now you may finish the definitions of the type predicate and visitors of `variable`.

B: *(Write the rest of the datatype definition of* `variable`*, consult the exercise set, and send it to the teacher for a check.)*

A: Now write the datatype definition of `fun`. Do you still remember how many parts are in a function?

B: A function has two parts–parameters and function body.

A: Right, but in our language a function has only one parameter. This is to make things simple.

B: Just one parameter. Wouldn't that be limited?

A: Not really. You may still simulate multiple parameters by nested functions, such as `x => y => x + y`. For example, the `compose` function in Lesson 1 was written as `(f, g) => x => f(g(x))`. In our language, you may write it as `f => g => x => f(g(x))`. This is more complex, but you can still express it.

B: Got it. Whenever we have a multi-parameter function `(x, y, ...) => body`, we write it as `x => y => ... => body`

A: Yes, remember that you need to change the calls too. Where you normally write `f(2, 3)`, now you have to write `f(2)(3)`.

B: This style of calls often appear in Lesson 1's exercises. They puzzled me a lot, but now I'm used to it.

A: After you understand the interpreter, multi-parameter functions are fairly easy to add. Now you may write the datatype definition for `fun`. Its two members should be named "param" and "body".

B: *(Write the definition of* `fun` *(constructor, type predicate and visitors) and send it to the teacher.)*

A: Next is the definition for `call`. Do you remember how many parts are in a call?

B: Two. The operator and the operand.

A: Good. You may write the datatype definition of `call` now. Its two members can be named "op" and "arg".

B: *(Write the definition of* `call` *and send it to the teacher.)*

A: Let's do a small exercise. Can you translate the JavaScript function `x => y => x + y` into our language?

B: *(Write your answer and send it to the teacher.)*

A: Now translate the call `(x => y => x + y)(2)(3)` into our language. You may copy the previous answer because it is part of the call.

B: *(Write your answer and send it to the teacher.)*

Structure of the interpreter

A: You are almost done with datatype definitions. Before we get into more details, I hope you can have a working interpreter really soon. This can motivate you. Let's look at the general structure of the interpreter.

B: Okay.

A: One thing that makes the interpreter different from the calculator is that the interpreter supports multiple language constructs (`variable`, `fun`, `call`, `binop`), whereas the calculator supports only `binop`. For this reason we need multiple branches in the interpreter. Its structure looks like this.

```
function interp(exp)
{
  if (typeof(exp) == "number")
  {
    ...
  }
  else if (isVariable(exp))
  {
    ...
  }
  else if (isFunction(exp))
  {
    ...
  }
  else if (isCall(exp))
  {
    ...
  }
  else if (isBinOp(exp))
  {
    ...
  }
  else
  {
    throw "Illegal expression: " + pairToString(exp);
  }
}
```

It is not complete code so don't copy it as yet.

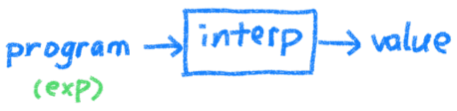
B: So we have a branch for each language construct. The last branch will report unrecognized expressions.

A: Yes, the structure is quite regular. We also have a branch for *literal values*. Literals are such as `2`, `3`, `"hello"`, `true`, `false`. They each represent one specific value in the language. For now it's okay to have just numbers. We may want to add other literals later.

B: I see the first branch is for number literals.

A: Like the calculator, the interpreter is a recursive function on trees. The parameter `exp` is the input program (expression). The interpreter will compute its value.

B: This input-output pattern of interpreter seems to be exactly the same as the calculator.



A: Actually this is not the whole picture, but it helps. First, think about this question. What do you return for the first branch, when `exp` is a number?

B: This is similar to the calculator's base case. For that case I just return the number itself.

```
if (typeof(exp) == "number")
{
  return exp;
}
```

A: Correct. Since you noticed that this is also the base case of the calculator, maybe it is helpful for you to implement the `binop` branch as the next step.

B: That seems to make a very smooth transition. I can then use the interpreter as a calculator even though it is not complete yet.

A: Remember that we are *growing* from the calculator.

B: Yes.

A: The `binop` branch should be almost the same as the `calc` function, except that you need to recursively call `interp` instead of `calc`.

B: *(Finish the `binop` branch so your interpreter is equivalent to the calculator. Test it with simple arithmetic expressions. Send the interpreter to the teacher for a check.)*

Variables

A: Very good. Now we can proceed to add other constructs. Let's look at the branch for variables. What should we do for this branch?

```
  else if (isVariable(exp))
  {
    ...
  }
```

B: We should return the variable's value. The value must be stored somewhere, but I have no idea where.

A: Actually, I lied about the interpreter's parameters. There is one more parameter for the `interp` function. Its name is `env`, meaning ***environment***. `env` is a lookup table, but it may also be a BST. It contains mappings from variable names to values.

Here is the new framework of `interp`. You may now add the `env` parameter to your `interp` function.

```
function interp(exp, env)
{
  if (typeof(exp) == "number")
  {
    ...
  }
  else if (isVariable(exp))
  {
    ...
  }
    ...
}
```

B: I see. I can just lookup `env` for the variable's value.

```
  else if (isVariable(exp))
  {
    return lookupTable(exp, env);
  }
```

A: You are almost there, but `env`'s keys are strings, not variables, so you need to get the variable's name first.

B: That's easy to fix.

```
  else if (isVariable(exp))
  {
    return lookupTable(variableName(exp), env);
  }
```

A: Good. There is one more problem. If the programmer makes a mistake, we may have *undefined variables* in the input program. For example we may have `(x => x * y)(3)`. In this case you need to report the error.

B: I will change it this way then.

```
  else if (isVariable(exp))
  {
    var value = lookupTable(variableName(exp), env);

    if (value == null)
    {
      throw "undefined variable: " + variableName(exp);
    }
    else
    {
      return value;
    }
  }
```

A: Good. Reporting the variable's name is a good idea. This will help the programmer locate the problem.

B: But I'm a little sad because I didn't think of the problem of undefined variables. Is there a systematic way of thinking so that I won't miss a case like this?

A: The wisdom is, always think about *all possibilities* of a variables or a function's return value. If there are possibilities which you haven't checked, the program will go wrong there.

B: That seems to be a good way of thinking. I'll keep that in mind.

A: There is one more thing to change here. We may want to change `env`'s data structure later (for example to a BST), so it is better not to hardcode `lookupTable` here. We may abstract this out using *abstract interfaces*.

B: I have used abstracted interfaces, but I'm not sure how to use it for this case.

A: Names are the essence of abstraction. You may just define three variables like the following. If we ever want to switch to BST, we change just three lines. No other code needs to be changed even if you used them a thousand times.

```
var emptyEnv = emptyTable;
var extEnv = addTable;
var lookupEnv = lookupTable;
```

B: The idea of abstraction is so profound. Now the code of the variable branch looks like this.

```
  else if (isVariable(exp))
  {
    var value = lookupEnv(variableName(exp), env);

    if (value == null)
    {
      throw "Undefined variable: " + variableName(exp);
    }
    else
    {
      return value;
    }
  }
```

A: Remember that when you wrote the `binop` branch, the `interp` function had only one parameter. Now we have two parameters, so you should add `env` to the recursive calls in the `binop` branch, otherwise they will go wrong.

B: I will do that.

(Send your extended interpreter code to the teacher for a quick check.)

Functions

A: The variable branch is all good now. You may start thinking about the function branch. What is the value of a function?

B: From Lesson 1, we learned that the value of a function is the function itself, plus some extra information.

A: What's the extra information?

B: When we have a call `(x => y => x + y)(2)`, Chrome's console gives us `y => x + y`, but there is extra information that "`x` is 2 inside `y => x + y`".

A: It's good that you remembered this, but we will first pretend that the value of a function is just the function itself. This can make the transition smoother. Just keep a note that we have something missing here.

B: Okay. I just return the function itself for now.

```
else if (isFunction(exp))
{
  return exp;  // something is missing here
}
```

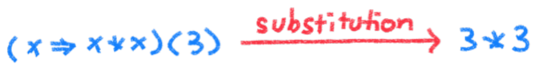
A: Let's move on to the `call` branch. After that everything will be connected together.

B: Okay.

Calls

A: Do you remember how calls happen?

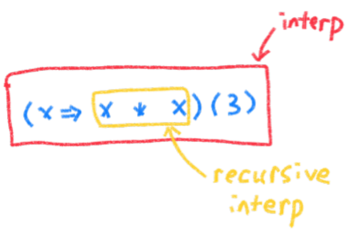
B: I remember *substitution*. In the function body, we replace every occurrence of the parameter with the operand. The value of the call is then the value of the substituted function body. I remember this example from Lesson 1.



A: Good, but substitution takes significant computing time and is complex to implement, so our interpreter will use a more practical strategy. Instead of substitution, we just recursively call the `interp` function on the function body.

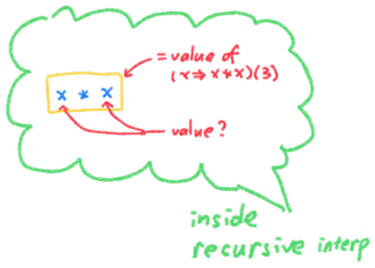
B: You mean, for this example `(x => x * x)(3)`, we recursively call `interp` on the function body `x * x`?

A: Right. If you draw a picture, it may look like this.



B: That's clear. If we call interp recursively on the function body, and we know the value of x, then we can compute the value of the call (x => x * x)(3).

A: That's the idea. The question is how can we find the value of x while inside the recursive interp.



B: From the variable branch, I know we get the value of x from env.

A: But how did env contain the value of x? It didn't yet contain the value of x when interp sees (x => x * x)(3).

B: Hmm, so we must put the key-value pair x:3 into env somewhere, otherwise it just won't happen.

A: Actually, we extend env with the key-value pair x:3 before the recursive call of interp on the function body x * x.

B: I see. First we do addTable("x", 3, env). Oh, actually extEnv("x", 3, env) because we need abstraction.

A: Right. extEnv("x", 3, env) creates a new environment (call it newEnv). newEnv contains the value of x, so the recursive call interp(funBody(exp), newEnv) knows the value of x.

B: This is my current code.

```
  else if (isCall(exp))
  {
    var newEnv = extEnv(funParam(callOp(exp)), callArg(exp),
env);
    return interp(funBody(callOp(exp)), newEnv);
  }
```

A: This is not right yet. Can you really call funBody on the operator callOp(exp)? Is the operator a function?

B: Isn't the operator a function?

A: It may not be. For example f(3), which is call(variable("f"), 3) in our language. The operator is variable("f"), which is a variable, not a fun!

B: This is interesting. We habitually say that f is a function, but actually f is just a variable!

A: Right. You are confused because of the imprecise way of describing things in natural languages and math. Now with programming languages, we have to be very accurate, otherwise things won't work.

B: Somebody said that programming is just another name for the lost art of thinking. Now I seems to understand it.

A: Very good. Actually for the call branch, you need to first evaluate both the operator and the operand. Otherwise you can't even ask if the operator is a function.

B: Got it. When `exp` is `f(3)`, we must first evaluate the operator `f` by a recursive call to `interp`. The recursive `interp` will lookup the value of the variable `f` in `env`. We need the same thing for the operand.

```
else if (isCall(exp))
{
  var op = interp(callOp(exp), env);
  var arg = interp(callArg(exp), env);
  var newEnv = extEnv(funParam(op), arg, env);
  return interp(funBody(op), newEnv);
}
```

A: You forgot to check whether `op` is a function. The programmer may make a mistake. He may have written `2(3)`. In this case you have to report the error "Calling non-function: 2".

B: Oh, old mistake again. I should always consider all possibilities.

```
else if (isCall(exp))
{
  var op = interp(callOp(exp), env);
  var arg = interp(callArg(exp), env);

  if (isFunction(op))
  {
    var newEnv = extEnv(funParam(op), arg, env);
    return interp(funBody(op), newEnv);
  }
  else
  {
    throw "Calling non-function: " + pairToString(op);
  }
}
```

A: This is much better. You are almost there, but there is still something wrong in the `call` and `fun` branch.

B: I guess we need to talk about the missing information in the function branch?

A: Yes. Let's talk about the example `(x => y => x + y)(2)` as you have seen in Lesson 1. Translate `(x => y => x + y)(2)(3)` into our language and run it with `interp`. Can you get the correct result `5`?

B: No, it complains that the variable `x` is undefined.

(Show that this happens to your teacher.)

A: The error message came from the variable branch. When `interp` sees the inner function's body `x + y`, is `x`'s value 2?

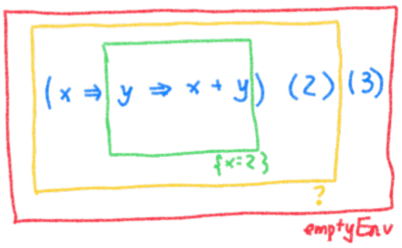
B: I think so, because when it sees `(x => y => x + y)(2)`, the call branch has put the key-value pair `x:2` into `env`, and it's called `newEnv`.

A: But `newEnv` is passed to the recursive call, so it is only visible inside the function body of `x => y => x + y`, which is `y => x + y`. When the recursive call returns, the original `env` doesn't contain `x`'s value.

B: You mean `env` doesn't contain x's value when we see `(x => y => x + y)(2)(3)`?

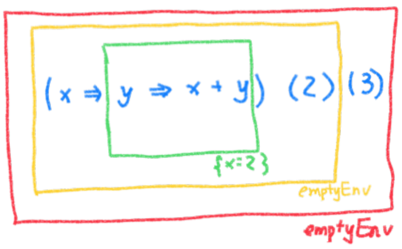
A: Right. You can draw a picture of the environments. Each nested expression may have a different environment.

B: I tried, but I have trouble about the yellow one. The green one is created when we evaluate `(x => y => x + y)(2)` and it contains `x:2`. I'm not sure about the yellow one.



A: The yellow environment is actually the same as the red one. Remember that we evaluate `(x => y => x + y)(2)` as the operator part of `(x => y => x + y)(2)(3)`. When we evaluate the operator and the operand, we use `env` without extend it.

B: I see. This is the new picture.



A: Any way, only the green environment contains `x:2`. When we evaluate the outermost call `(x => y => x + y)(2)(3)`, we can't see `x:2` because the red environment is empty.

B: What can we do about this? Can we do a substitution and create `y => 2 + y`?

A: No. As I said, we don't do substitutions in this interpreter.

B: Then I have no idea.

A: Here is a way. When `interp` sees a function, it can bundle the function and the current environment together, forming a so-called *closure*.

B: What does a closure look like?

A: For this example, the closure will contain the function `y => x + y` and the green environment.

B: I see. The green environment contains `x:2`. But I'm still not sure what to do.

A: You need to define a new datatype `closure`. It has two members "fun" and "env", so the visitors will be called `closureFun` and `closureEnv`.

B: *(Write the definition of `closure` datatype and show it to the teacher.)*

A: After you have the closure datatype, the function branch will just be like this:

```
else if (isFunction(exp))
{
  return closure(exp, env);
}
```

B: That's simple.

A: There is one more change you need for the call branch. For the call, we no longer get a function when evaluating the operator. We get a closure instead.

B: What do we do with the closure?

A: Do you remember why we have the closure?

B: To access the information in the green environment.

A: We use this environment so that we can know the variables' values when the function was created.

B: For example, `x` as in `y => x + y`?

A: Yes. We call `x` a *free variable* because it is not a parameter of this function. Its value came from outside of the function.

B: I see, *free variable*.

A: So in the recursive call of `interp` on the function body, we don't use the `env` parameter. We use the environment stored in the closure.

B: You mean something like this?

```
else if (isCall(exp))
{
  var op = interp(callOp(exp), env);
  var arg = interp(callArg(exp), env);

  if (isClosure(op))
  {
    var f = closureFun(op);
    var newEnv = extEnv(funParam(f), arg, closureEnv(op));
    return interp(funBody(f), newEnv);
  }
  else
  {
    throw "Calling non-function";
    }
  }
```

A: Correct.

B: That feels strange. Won't I miss any useful information in the current `env`?

A: You won't miss anything because the function is not created here. It is created where we could see the green environment.

B: I see. So it makes sense to bundle the environment at the moment where the function is created.

A: Yes. This will solve all our problems.

B: Nice.

A: This is all I will teach you about interpreters. There are some extensions to the interpreter which I left as exercises. They will deepen your understanding of interpreters. If you have questions, just let me know. (Exercise Set 7)

B: Thank you!

(Exercise omitted for the sample.)