

# Ground-Up Computer Science

---

## Chapter 3

(March 11, 2023)

Yin Wang

© 2021-2023 Yin Wang (yinwang0@gmail.com)

All Rights Reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photographing, photocopying, recording, or information storage or retrieval) without permission in writing from the author.

## 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.

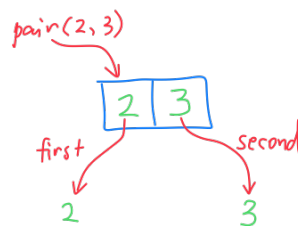
B: What is the role of pairs in this virtual world?

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

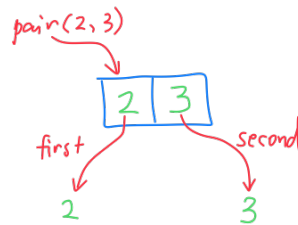
A: From now on, we need to forget about the details inside `pair`, `first` and `second`.

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.

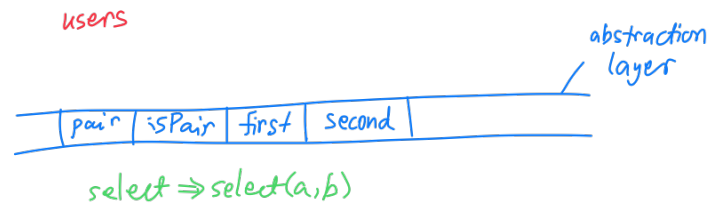
B: Nice.

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

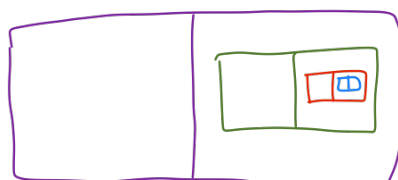
A: I guess you have now understood the definition of pair. Pair is a fundamental *data structure*. It is very important. We can use pairs to create complex and interesting structures.

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?

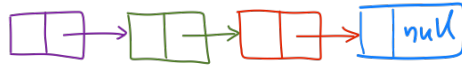
A: We don't put useful things in the **second** part of the last pair. We put in a special value which means "nothing" or "empty", much like a notice "END of TRAIN".

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)` is 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 produce 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.)