FUNctions

Lecture 03 – Spring 2018

Announcements

- PS0 – Due Tomorrow at 11:59pm
- WS1 – Released soon, due next Friday 2/2 at 11:59pm
- Not quite understand a topic in lecture this week?
  - Come to Tutoring – Tomorrow 12-4pm in SN115

Warm-up #1) What is the output of these programs?

```
let x: number = 13;
if (x < 18) {
   print("A");
}

if (x === 13) {
   print("B");
} else {
   print("C");
}
```

…

```
let x: number = 13;
if (x < 18) {
      print("A");
} else {
   if (x === 13) {
      print("B");
   } else {
      print("C");
   }
}
```

Warm-up Question: **Variables**

- What is the final value of the variable **x** after this code runs?

```
let x: number;
let y: number;
x = 1;
y = 2;
y = y + x;
x = x + y * 2;
```

- Respond on PollEv.com/comp110

Functions are *inspired by* their mathematical relatives...

$$f(x) = (x \times 3) + 1$$

- What is $f(3)$? What is $f(f(1))$? Answer on PollEverywhere.

- We know that to compute $f(5)$ we
  1. Assign 5 to x such that the expression becomes $(5 \times 3) + 1$
  2. Using arithmetic, simplify to $15 + 1$
  3. Using arithmetic, simplify to 16. Final answer.

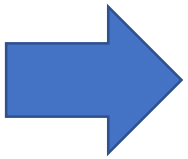
- Let's express the same function in code.

Follow along: Our First Function

- Open lec03 / 00-function-demo-app.ts
- Let's define the function ***f*** together!
  - Notice: It is defined outside of the *main* function!
  - We will break down the syntax next.
- Then let's call function ***f*** from within main.

```
let f = (x: number): number => {
   return (x * 3) + 1;
};

export let main = async () => {
  let input: number = await promptNumber("?");
  let answer: number = f(input);
  print(answer);
};

main();
```

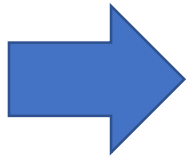
Tracing a function call (1/11)



```
let f = (x: number): number => {
   return (x * 3) + 1;
};

export let main = async () => {
  let input: number = await promptNumber("?");
  let answer: number = f(input);
  print(answer);
};

main();
```

The user is prompted for a number. The number entered is assigned to the **input** variable. Let's imagine **5** was entered.

Tracing a function call (2/11)



```
let f = (x: number): number => {
   return (x * 3) + 1;
};

export let main = async () => {
  let input: number = await promptNumber("?");
  let answer: number = f(input);
  print(answer);
};

main();
```

5

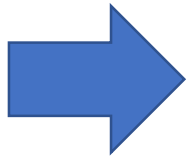
input

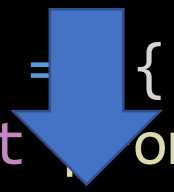
The variable answer is declared and initialized to be **f(input)**

But what is f(input)? The computer must compute it!

Tracing a function call (3/11)

```
let f = (x: number): number => {
   return (x * 3) + 1;
};

export let main = async () => {
  let input: number = await promptNumber("?");
  let answer: number = f(input 5);
  print(answer);
};

main();
```





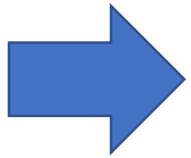
5

input

First, it is going to substitute the **input** variable reference with its value **5**.

Note: You will never *see* this happen. The computer is doing this as it runs your program.

Tracing a function call (4/11)



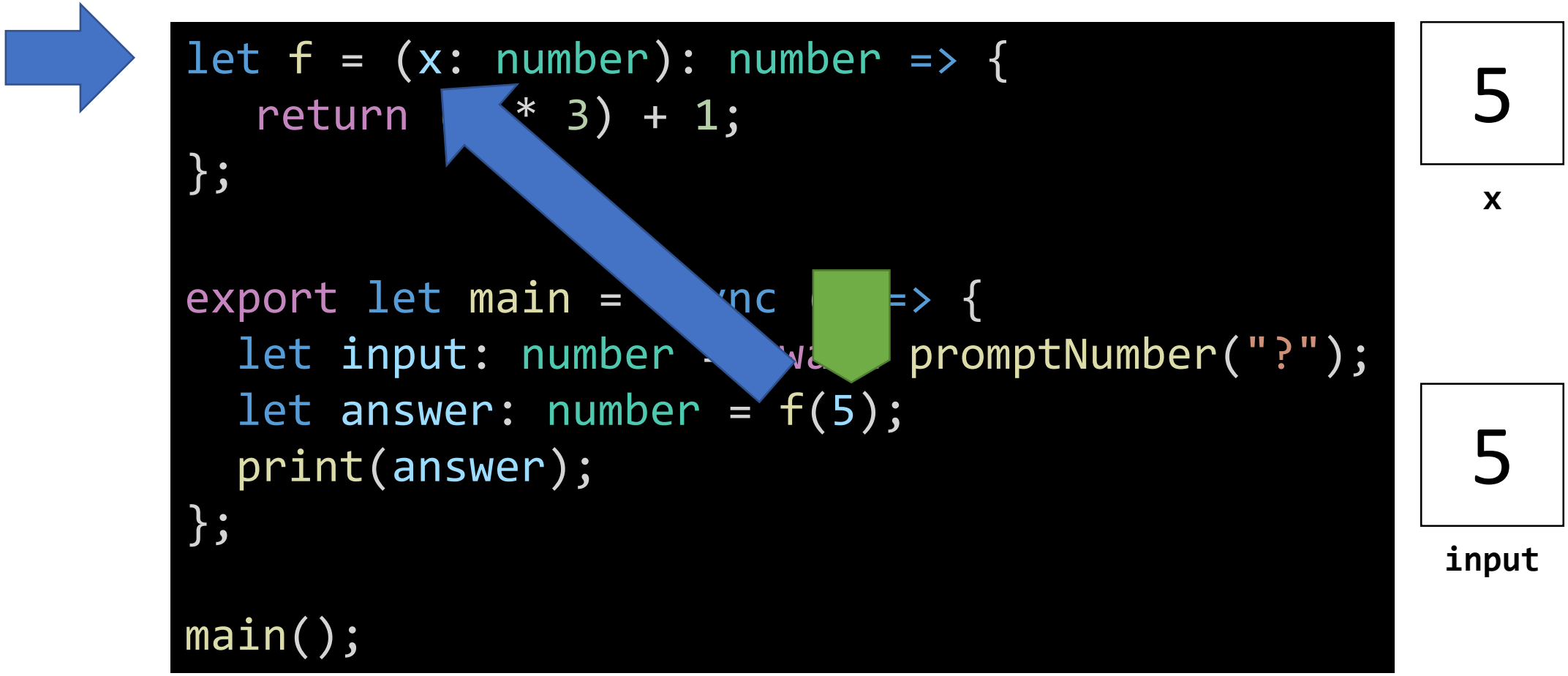
```
let f = (x: number): number => {
   return (x * 3) + 1;
};

export let main = asyn    ) => {
  let input: number =    it promptNumber("?");
  let answer: number = f(5);
  print(answer);
};

main();
```

5

input

Now the computer is ready to call function ***f.*** It drops a bookmark.

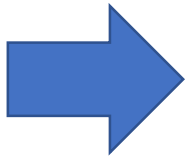
Tracing a function call (5/11)



Then, we assign the value in parenthesis **(5)** to ***f***'s variable ***x***.

We will cover this process in depth next.

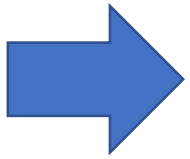
Tracing a function call (6/11)



```
let f = (x: number): number => {
   return (x * 3) + 1;
};

export let main = asyn    ) => {
  let input: number =    it promptNumber("?");
  let answer: number = f(5);
  print(answer);
};

main();
```

The function is entered and return statement is reached.
The computer needs to calculate the result of this expression.

Tracing a function call (7/11)



```
let f = (x: number): number => {
    return (5 * 3) + 1;
};

export let main = asyn    ) => {
  let input: number =    it promptNumber("?");
  let answer: number = f(5);
  print(answer);
};

main();
```

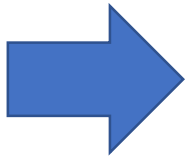
5
x

5
input

First, it will substitute the ***x*** variable with its current value.

Note: You will never *see* this happen in your code. The computer is doing this as it runs your program.

Tracing a function call (8/11)



```
let f = (x: number): number => {
   return (5 * 3) + 1;
};

export let main = asyn    ) => {
  let input: number =    it promptNumber("?");
  let answer: number = f(5);
  print(answer);
};

main();
```

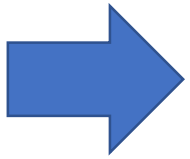
5
x

5
input

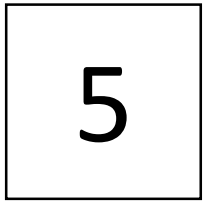
First, it will substitute the ***x*** variable with its current value.

Note: You will never *see* this happen in your code. The computer is doing this as it runs your program.

Tracing a function call (9/11)



```
let f = (x: number): number => {
   return (15) + 1;
};

export let main = asyn    ) => {
  let input: number =    it promptNumber("?");
  let answer: number = f(5);
  print(answer);
};

main();
```

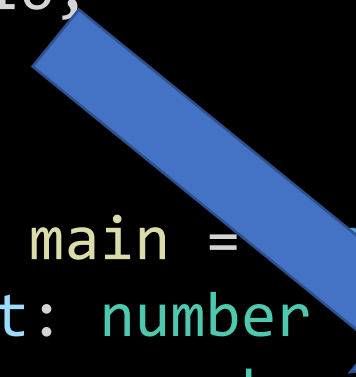


Then, it will follow PEMDAS.

Note: You will never *see* this happen in your code. The computer is doing this as it runs your program.

Tracing a function call (10/11)

```
let f = (x: number): number => {
   return 16;
};

export let main = async (): => {
  let input: number = await promptNumber("?");
  let answer: number = f(5) 16;
  print(answer);
};

main();
```



5

x

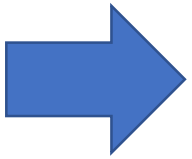
5

input

Once the return statement is computed down to a single value, it is ***returned*** *to the function call's bookmark and replaces it.*

Note: You will never *see* this happen in your code. The computer is doing this as it runs your program.

Tracing a function call (11/11)



```
let f = (x: number): number => {
   return 16;
};

export let main = async () => {
  let input: number = await promptNumber("?");
  let answer: number = 16;
  print(answer);
};

main();
```

5	16
input	**answer**

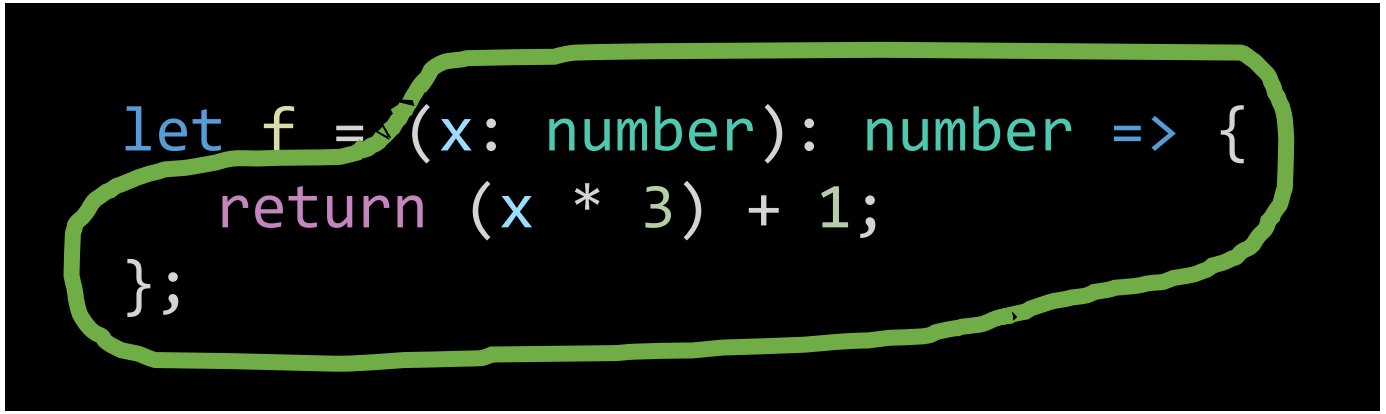
The computer returns to processing the program at this line and initializes **answer** to 16.

Function Definition Walk through (1/5)

```
let f = (x: number): number => {
    return (x * 3) + 1;
};
```

"Let ***f*** be..."

Function Definition Walk through (2/5)



"Let ***f*** be... **a function**..."

Function Definition Walk through (3/5)

```
let f = (x: number): number => {
    return (x * 3) + 1;
};
```

"Let ***f*** be... a function...

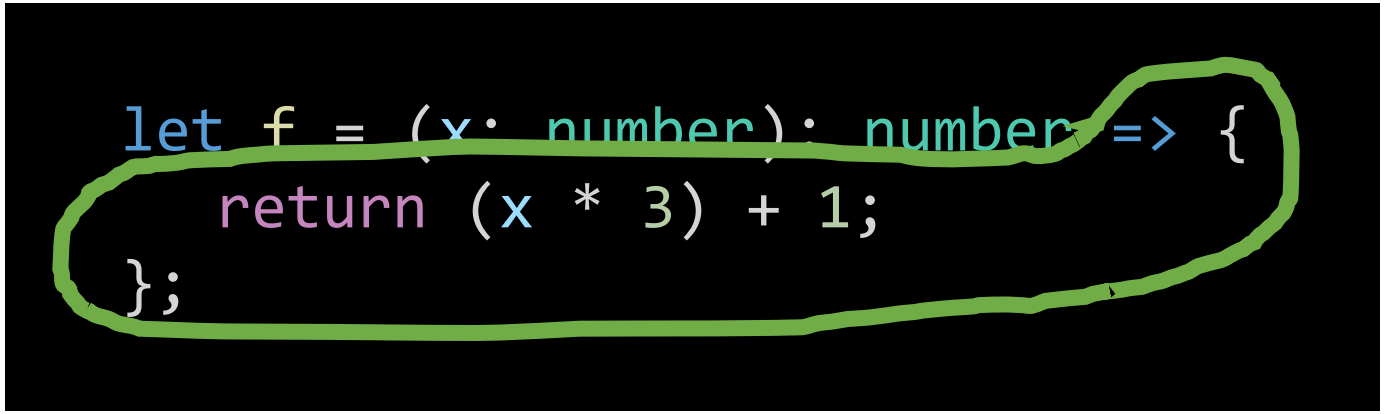
that needs a number value named *x*..."

Function Definition Walk through (4/5)

```
let f = (x: number): number => {
   return (x * 3) + 1;
};
```

"Let ***f*** be... a function... that needs a number value named *x*...
and will return a number value when called."

Function Definition Walk through (5/5)



"Let ***f*** be a function that needs a number value named *x* and will return a number value when called."

"When ***f(x)*** is called, the result of computing **(x * 3) + 1** will be returned to the caller."

What Purpose do **Functions** Serve?

- Functions make it easy for us to reuse computations or sequences of steps
- More formally, functions enable **process abstraction**
- Learning to tie your shoe was a process abstraction
  - As a child, you struggled to learn the right series of steps
  - Nowadays you can just "tie your shoe" without worrying about each step
- Defining a function is a process abstraction
  - Defining functions takes thoughtful effort to get the right series of steps
  - Once correct, you can "call", or *reuse*, your function without worrying about its steps
- Functions help us break down and logically organize our programs

Function Syntax

```
let <name> = (<parameters>): <returnType> => {
  <function body statements>
};
```

- Like variables, functions can be given a **name.**
- **Parameters** are special variables. They are the extra pieces of information, or inputs, a function needs.
- **Return type** specifies the type of data the function will return.
- **Statements** in the function definition's **block** run *only* when a function is called.

Hands-on: Calculating Perimeter

- Open *02-perimeter-app.ts*

- At TODO #1: Rather than initialize result with 0, initialize result with a call to the perimeter function:

```
let result: number = perimeter(length, width);
```

- At TODO #2: Rather than return 0, write an equation to calculate perimeter using the *length* and *width* variables.

- Check to see that your program is working. Check-in **pollev.com/compunc**

Introducing **Parameters**

- Parameters allow functions to require additional pieces of information in order to be called

- Parameters are specified within the parenthesis of function definition

- Parameters look a lot like variable declarations
  - *Because they are!*

General Form

```
// Function Definition
let <name> = (<parameters>): <returnT> => {
   <statements>
};
```

Example

```
// Function Definition
function perim(l: number, w: number): number {
   return (l * 2) + (w * 2);
};
```

What effect does declaring parameters have?

Function Definition

```
// Function Definition
let perim = (len: number, wid: number): number => {
   return (len * 2) + (wid * 2);
};
```

Function Call Usage

```
perim(3.0, 4.0);
```

- When a function declares **parameters**, it is declaring:
  "you must give me these extra pieces of information in order to call me"

- The function ***definition*** on the left says:
  "in order to call **`perim`**, you must give me two number values"

- In the *usage* to the right, when we ***call*** perim:
  "call **`perim`** and assign **3.0** to the ***len*** parameter and **4.0** to the ***wid*** parameter"

Parameters vs Arguments

These are **arguments.**

```
perim(8, 9);
```

- Arguments are the *values* we assign to parameters
- The type of the arguments must match the types of the parameters
- We couldn't write `perim("oh","no");`

These are **parameters.**

Example

```
// Function Definition
let perim = (len: number, wid: number): number => {
   return (len * 2) + (wid * 2);
};
```

Parameter Passing: Step-by-Step (1 / 3)

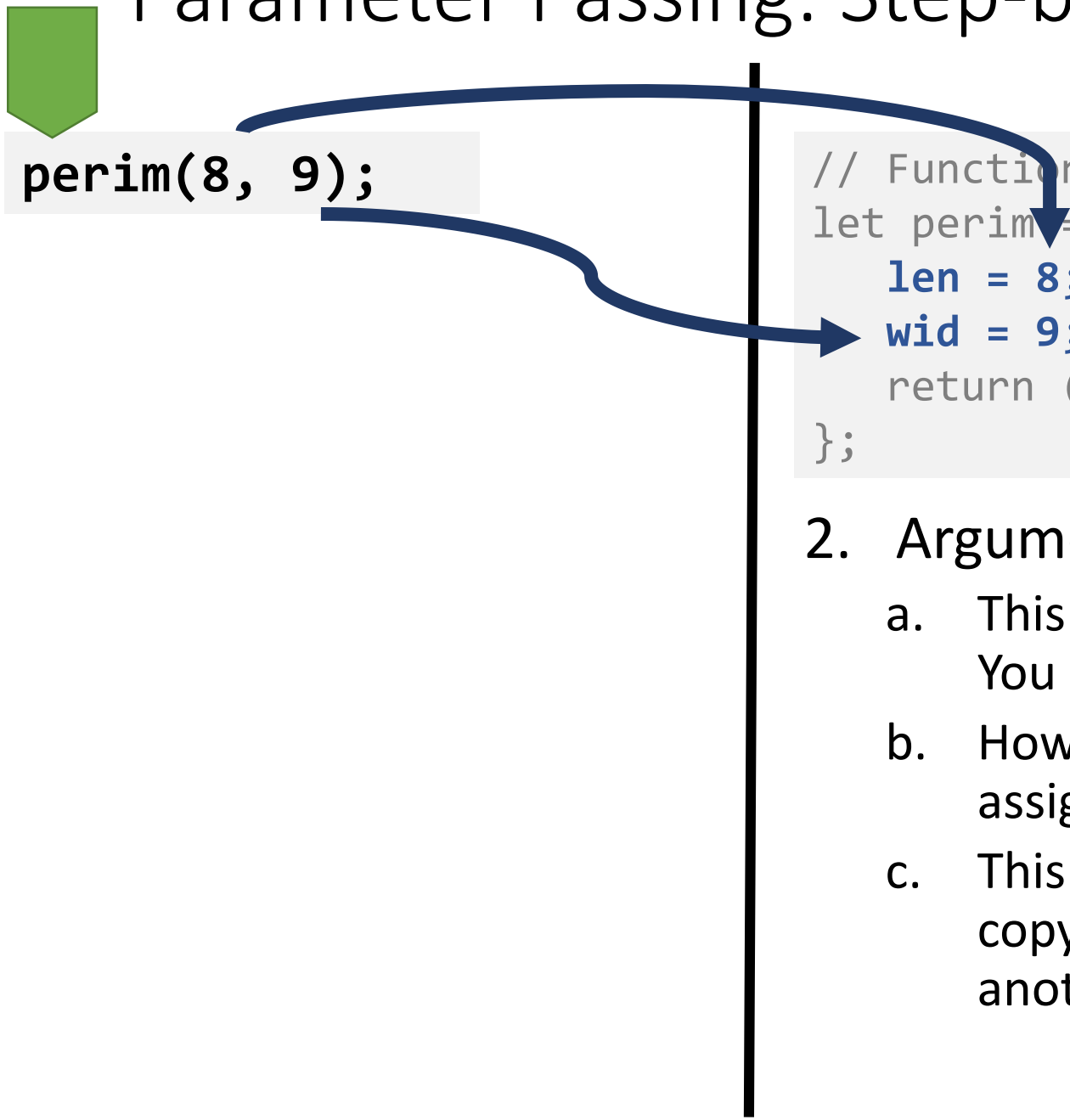
```
perim(8, 9);
```

1. When a function is called…
   a. A "bookmark" is dropped at this place in code. We'll come back!
   b. The processor finds the function definition with the *same name*.
   c. Error if no match is found!

```
// Function Definition
let perim = (len: number, wid: number): number => {
   return (len * 2) + (wid * 2);
};
```

Notice the argument *matches* the parameters in type (number) and count (2)!

Parameter Passing: Step-by-Step (2 / 3)



```
perim(8, 9);
```

```
// Function Definition
let perim = (len: number, wid: number): number => {
    len = 8;
    wid = 9;
    return (len * 2) + (wid * 2);
};
```

2. Argument values are assigned to parameters
    a. This happens invisibly when the code is running. You will never *see* lines to the right.
    b. However, each time a call happens, the processor assigns the argument value to the parameter.
    c. This is called "parameter passing" because we are copying arguments from one point in code *into* another function's block.

Parameter Passing: Step-by-Step (3 / 3)

```
perim(8, 9);
```

```
// Function Definition
let perim = (len: number, wid: number): number => {
    len = 8;
    wid = 9;
    return (len * 2) + (wid * 2);
};
```

3. Finally, the program then *jumps into* the function and continues running line-by-line

Hands-on #2) Implement the **max** function

- We're trying to calculate the price of 2 sushi rolls at SPICY 9
  - They have a BOGO deal where you pay the price of the more expensive roll and the other is free

- Your objective:
  - Write an if-then-else statement in the **max** function with the following logic

```
IF a GREATER THAN b
   THEN return a
OTHERWISE return b
```

- Test by changing the prices of the two rolls

- Check-in on PollEv.com/compunc when complete

```
if (test) {
  // then block
} else {
  // else block
}
```

The **return** Statement

- General form:

```
return <expression>;
```

- Expression's type ***must*** **match the return type** of its function

- Every function that returns a value must have at least one **`return`** statement

- IMPORTANT: As soon as *any* **`return`** statement is reached, the function call is complete.
  - The computer evaluates the expression and returns the value immediately to its bookmark.
  - The rest of the function is ignored, skipped over, and not processed.
  - ***This is ALWAYS, ALWAYS, ALWAYS true!***

Return Semantics: Consider the following function

- Consider the **max** function to the right
- Its purpose is to return the greater value of the parameters **a** and **b**
- *Does it?* What happens when **a** is greater?

```
let max = (a: number, b: number): number => {

      if (a > b) {
            return a;
      }

      return b;

};
```

Returning from a **function**

```
let result: number;
result = max(10, 5);
```

1. The **max** function is called with arguments:
   10, 5

2. The processor jumps to max function.
   - if (a > b) evaluates to true, enters **then block**

3. **return** Statement encountered. Expression **a** evaluates to **10**. The function call is complete and this value is returned to step 4.

4. Processor jumps back to bookmark it left at #1 and "max(10, 5)" evaluates to **10**.

```
let max = (a: number, b: number): number => {

    if (a > b) {
        return a;
    }

    return b;

};
```

Parameters	
a	10
b	5

Every function ***call*** can ***return only once***

- Every function call is an *expression.* By definition, an expression is something that *evaluates to a single value*.

- A function *may* contain many `return` statements

- A function *may* contain a return statement inside of a loop (coming next)

- As soon as the computer reaches *any* return statement *once* within a function, that function call is completed and the value is returned.

Follow-along: **max3**

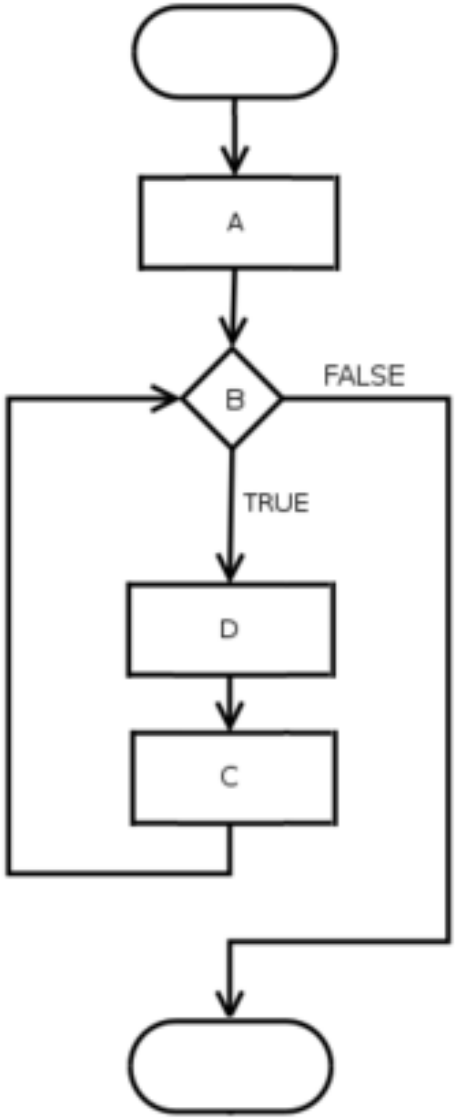
- In the file **lec 03 / 03-nested-calls-app.ts**

- We'll implement the max3 function without writing any if-then-else statements… *how?!?*... function reuse!

```
/**
* Given 3 values, max3 will return the largest of all 3.
*/
let max3 = (a: number, b: number, c: number): number => {
   return max(a, max(b, c));
};
```

Control Flow or "The Moves"

- Technically, what we're about to learn is called "control flow"
- This refers to how the computer ***moves*** through processing your instructions
- I'm going to introduce these to you with my own made up phrase: ***the moves***.



Think about a board game like **Monopoly**…

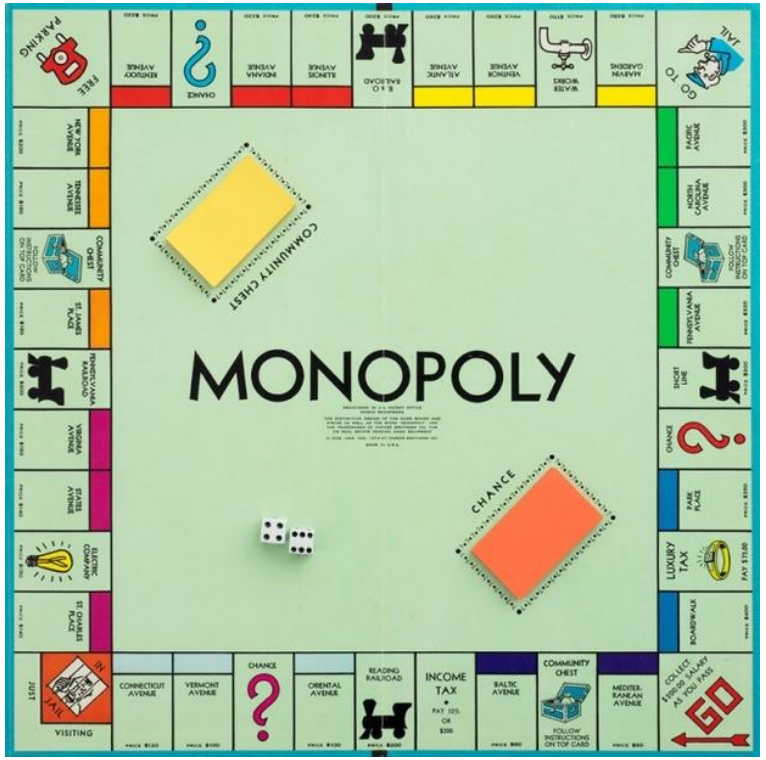
- Imagine you rolled a 1 every time.
  You ***move*** forward one space at a time.

- *Until* you reach a special square, i.e.:

  - **Chance Card Square**
    - A special move card can *jump* you around the board.

  - **Go to Jail Square**
    - Move all the way backwards to jail.

  - **How do you get out of jail?**
    - *If* you roll doubles, *then* you move forward by the number of spaces shown by the throw.



The Moves

- Like a board game, there are special moves in programs that cause the processor to jump to different points in your code.

- In COMP110, there are ***4 moves*** we will master:

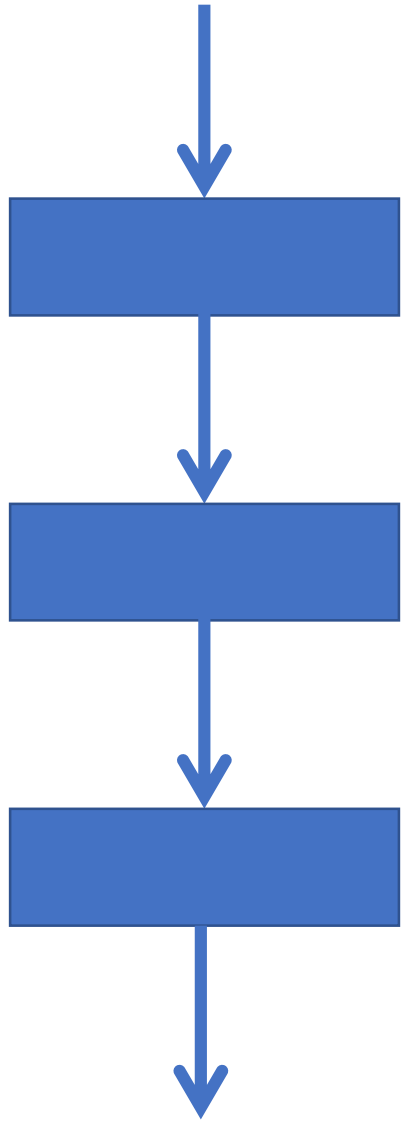
TODAY!

1. **Step Forward** ✓
2. **If-Then** ✓
3. Function Call
4. Loop

- In COMP401, you'll learn another move **`Exception handling`**.

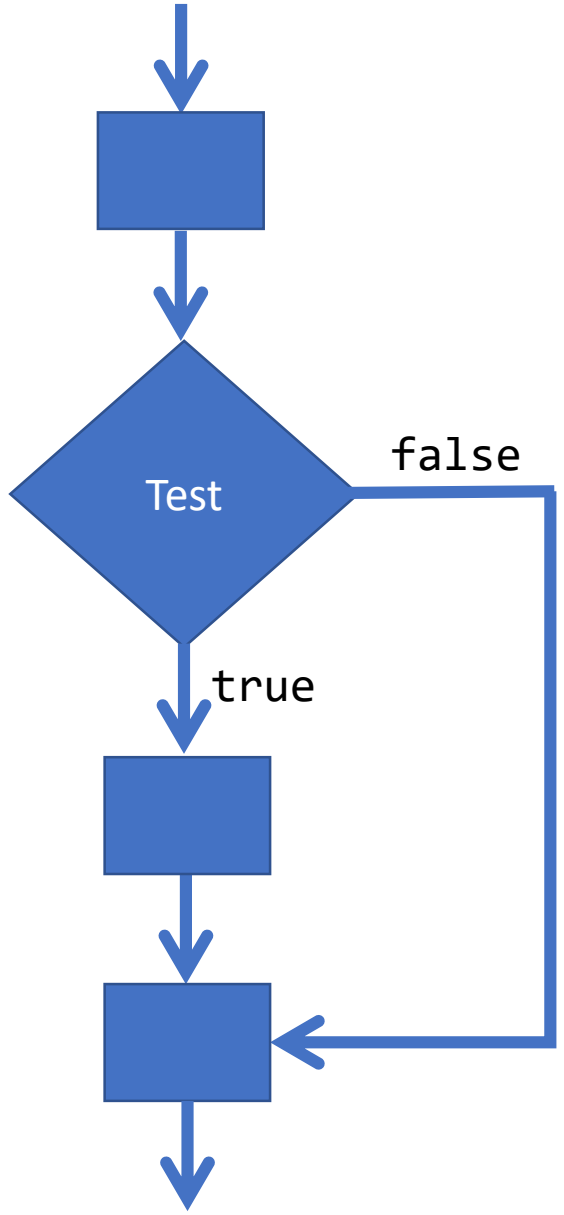
Move #1 – Step Forward

- You already know this one!
- The computer will process one line of code. Then it will process the next right after it. And the next…
- ***Until*** it encounters a *special* move or the end of the program.



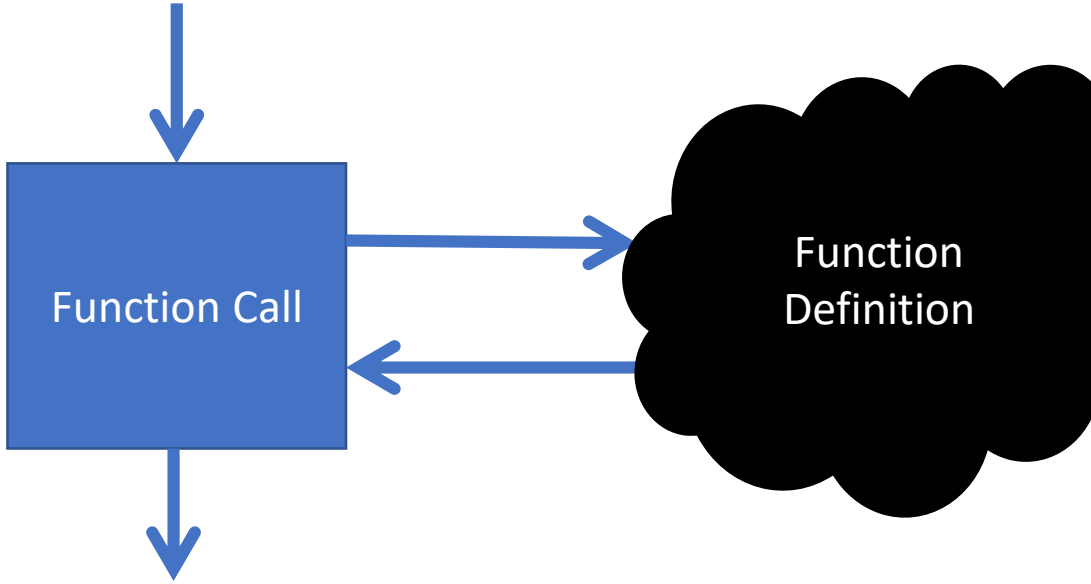
Move #2 – If-Then

- Based on some test (**`boolean`**!)…
- **if** the test is `true`
  - ***then*** the computer will continue to the next statement.
- **else** it will jump to a specific statement further down in your program



Move #3 – Function Call

- The **function call** is *beautiful* and *magical*. It's a power move.
- The computer *drops a bookmark* where the function call occurs and ***jumps*** into the function's definition... "*magic" happens...*
- ...the computer then ***returns*** back to *the bookmark it dropped*, usually with a result, and the program continues.



Expressions

- **Expressions** are a fundamental building block in programs

- Expressions are analogous to the idea of clauses in English
  - Single clause sentence:
    "I am a student."
  - Multiple clause sentence:
    "I am a student and I am currently sitting in COMP110."
  - *Sentences* are *more expressive* through the creative use of *clauses*

- ***Statements*** are *more expressive* through the creative use of ***expressions!***

How can we compute the volume of a cube using different expressions?

```
let length: number = 3;
let answer: number;
```

```
answer = 3 * 3 * 3;
```

"Hard-code" the equation with exact numbers.

```
answer = length * length * length;
```

"Hard-code" the equation with exact numbers.

Expressions

There are two **big ideas** behind expressions

1. *Every* expression *simplifies to a single value*
   - Thus, every expression has a *single result type*.
   - This occurs *only* when the program runs and the computer reaches that line of code in the program.

2. Anywhere you can write an expression, you can choose any other expression you'd like as long as their types match

Expressions – Some examples we've seen…

Expression	Resulting Type	Resulting Value	Expression Name
`"Hello, World"`	`string`	`"Hello, World"`	string Literal
`length`	`number`	?	Variable Reference
`length * length`	`number`	?	2x Variable Reference Arithmetic Operation
`"Area " + area`	`string`	?	Variable Reference Concatenation Operation

Where have we *used* expressions?

- Assignment operator:

```
let <name>: <type> = <expression of same type>;
```

- We are able to assign *any* of the expressions below because each results in a single *number* value:

```
let x: number = 1;
let y: number = x + 1;
let cubeY: number = y * y * y;
```

- Notice that we are combining *multiple* expressions in the same line.
- After each line completes, the declared variable has a *single* value.

Singular Expressions

- Literal Values
  - `1, 3.14, true, "hi"`
- Variable Access
  - `x, compCourseNumber`
- "Unary" operators (-)
  - `-x (`*`negation`*`)`

Compound Expressions

- Operators
  - Arithmetic
  - Concatenation
  - Equality
    ===
    !==
  - Relational
    >
    >=
- Function Calls
  - *Functions that **<u>return</u>**!*