1. Open Visual Studio Code
2. At the top click on View->Integrated Terminal (if not already open)
3. In the terminal, first run: `npm run pull`
4. After this finishes run: `npm start`

FUNction Tracing

Lecture 18

“One Step At A Time”
Announcements

• Quiz 7 grades posted

• PS04 – Walk due midnight tomorrow

• PS04 – Sprint posted

• comp101.org/videos
  • Great, underutilized resource!
3. What is returned by the call $f(3)$?

```typescript
let f = (i: number): string => {
    let str: string = "";
    while (i >= 0) {
        str = i + str;
        i--;
    }
    return str;
};
```
Evaluating nested expressions

let x: number = (2 * 2 + (6 - 5 * (3 - 2 * (7 - 5))));
Evaluating nested expressions

let x: number = (2 * 2 + (6 - 5 * (3 - 2 * (7 - 5))));

Evaluate innermost expression first

7 - 5
Evaluating nested expressions

let x: number = (2 * 2 + (6 − 5 * (3 − 2 * (7 − 5))));

Evaluate innermost expression first

2
Evaluating nested expressions

let x: number = (2 * 2 + (6 - 5 * (3 - 2 * 2)));
Evaluating nested expressions

```javascript
let x: number = (2 * 2 + (6 – 5 * (3 – 2 * 2)));
```

Continue evaluating innermost expression

```
3 – 2 * 2
```
Evaluating nested expressions

let x: number = (2 * 2 + (6 - 5 * (3 - 2 * 2)));

Continue evaluating innermost expression

3 - 4
Evaluating nested expressions

```javascript
let x: number = (2 * 2 + (6 - 5 * (3 - 2 * 2)));
```

Continue evaluating innermost expression

-1
Evaluating nested expressions

```javascript
let x: number = (2 * 2 + (6 - 5 * -1));
```
Evaluating nested expressions

```javascript
let x: number = (2 * 2 + (6 - 5 * -1));
```

Continue evaluating innermost expression

```
6 - 5 * -1
```
Evaluating nested expressions

let x: number = (2 * 2 + (6 - 5 * -1));

Continue evaluating innermost expression
Evaluating nested expressions

```javascript
let x: number = (2 * 2 + (6 - 5 * -1));
```

Continue evaluating innermost expression

```javascript
6 + 5
```
Evaluating nested expressions

let x: number = (2 * 2 + (6 - 5 * -1));

Continue evaluating innermost expression
Evaluating nested expressions

let x: number = (2 * 2 + 11);
Evaluating nested expressions

let x: number = (2 * 2 + 11);

Continue evaluating innermost expression
Evaluating nested expressions

let x: number = (2 * 2 + 11);

Continue evaluating innermost expression
Evaluating nested expressions

let x: number = (2 * 2 + 11);
Evaluating nested expressions

let x: number = 15
Evaluating nested expressions

let x: number = 15

• (Nearly) impossible to know this at the start
• Only solvable through careful tracing and application of nesting rules
Introducing: Nested functions

- We can do this same **nesting** with function calls

- Remember: Functions are *inspired by* their mathematical relatives

\[ f(x) = (x \times 3) + 1 \]

- What is \( f(f(1)) \)?
Introducing: Nested functions

• We can do this same **nesting** with function calls

• Remember: Functions are *inspired by* their mathematical relatives

\[ f(x) = (x \times 3) + 1 \]

• What is \( f(f(1)) \)?

\[ f(1) \]
Introducing: Nested functions

• We can do this same **nesting** with function calls

• Remember: Functions are *inspired by* their mathematical relatives

\[ f(x) = (x \times 3) + 1 \]

• What is \( f(f(1)) \)?

\[ (1 \times 3) + 1 \]
Introducing: Nested functions

• We can do this same **nesting** with function calls

• Remember: Functions are *inspired by* their mathematical relatives

\[ f(x) = (x \times 3) + 1 \]

• What is \( f(f(1)) \)?

\[ 3 + 1 \]
Introducing: Nested functions

• We can do this same **nesting** with function calls

• Remember: Functions are *inspired by* their mathematical relatives

\[ f(x) = (x \times 3) + 1 \]

• What is \( f(f(1)) \)?
Introducing: Nested functions

• We can do this same **nesting** with function calls

• Remember: Functions are *inspired by* their mathematical relatives

\[
f(x) = (x \times 3) + 1
\]

• What is \( f(4) \)?
Introducing: Nested functions

• We can do this same **nesting** with function calls

• Remember: Functions are *inspired by* their mathematical relatives

\[ f(x) = (x \times 3) + 1 \]

• What is \( f(4) \)?

\[ (4 \times 3) + 1 \]
Introducing: Nested functions

• We can do this same **nesting** with function calls

• Remember: Functions are *inspired by* their mathematical relatives

\[ f(x) = (x \times 3) + 1 \]

• What is \( f(4) \)?

12 + 1
Introducing: Nested functions

• We can do this same **nesting** with function calls

• Remember: Functions are *inspired by* their mathematical relatives

\[ f(x) = (x \times 3) + 1 \]

• What is \( f(4) \)?
Introducing: Nested functions

• We can do this same **nesting** with function calls

• Remember: Functions are *inspired by* their mathematical relatives

\[ f(x) = (x \times 3) + 1 \]

• \( f(f(1)) = 13 \)
1. 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

2. Your objective:
   1. Write a function named `max` that takes in two parameters \((a: \text{number}, b: \text{number})\) and returns a number
   2. Write an if-then-else statement in the `max` function with the following logic

   \[
   \begin{align*}
   \text{IF } a \text{ GREATER THAN } b & \quad \text{THEN return } a \\
   \text{OTHERWISE return } b
   \end{align*}
   \]

3. Test by changing the prices of the two rolls
4. Check-in on pollev.com/compunc when complete
let max = (a: number, b: number): number => {
    if (a > b) {
        return a;
    } else {
        return b;
    }
};
Follow-Along: max3

• Open lec18 / 00-bogo-app.ts

• We will implement the max3 function without writing any if-then-else statements...how?!!...function reuse!

• This function can compute the max of 3 numbers

• Remember, when evaluating nested expressions always evaluate from the innermost expression outwards
At home: How would you use this same pattern to write a max4 function?
Dwight's Hype Machine

https://www.youtube.com/watch?v=hkRjL2A41QQ
1. Open lec18 / 01-hype-machine-app.ts

2. At the first TODO comment, declare a function with the following properties:
   • Name: **hypeUp**
   • Parameters:
     1. name: string
   • Return Type: **string**

3. The function should return a string value that concatenates the name parameter to a hype sentence. For example:
   
   ```
   return name + " IS AWESOME";
   ```

4. At the second TODO comment, print the result of calling `hypeUp` with the given name:

   ```
   print(hypeUp(name));
   ```

5. Save, test, and check-in on PollEv.com/compunc
import { print, promptString } from "introcs";

// TODO: define the hypeUp function here
let hypeUp = (name: string): string => {
    return name + " IS AWESOME";
};

export let main = async () => {
    print("Welcome to the Hype Machine");
    let name: string = await promptString("What is your name?");
    // TODO: call the hypeUp function and pass name as an argument
    print(hypeUp(name));
};

main();
What if we want it to respond with a random hype up message?

- Let's import a function that generates random numbers for us!

```javascript
import { print, promptString, random } from "introcs";
```

- The random function's definition looks like this...

```javascript
let random = (floor: number, ceiling: number): number => {
  // Magic
};
```

- What do we know just by looking at the definition?
Using the `random` Function

```typescript
let random = (floor: number, ceiling: number): number => {
    // ...
};
```

- Looking at this definition, we know:
  1. To call `random` we must provide 2 arguments, both numbers
  2. When `random` returns, it will give us back a number

- Thus, we can *call* the random function like so: `random(1, 6)`

- The `floor` and `ceiling` parameters represent the smallest and largest numbers `random` will choose a number between, inclusive.
How do we generate a random string when all we have is a random number?

• Using **if-then-else** statements based on the random number!

```javascript
let choice: number = random(1, 6);
if (choice === 1) {
    return name + " IS AWESOME";
} else {
    if (choice === 2) {
        return name + " IS THE BEST";
    } else {
        return name + " IS GREAT";
    }
}
```

• Let's try this in the next hands-on!
Hands-on: Random Messages

1. Still working in 01-hype-machine-app.ts

2. Add `random` to the list of functions being imported from "introcs"

3. In the `hypeUp` function, before returning, declare a variable of type number named `choice`. Initialize `choice` to the result of calling `random(1, 3)`

4. Write `if-then-else` statements that test if `choice` is === 1, otherwise if it's === 2, and so on (see previous slide), and return a different message in each case.

5. Save. Check-in on pollEv.com/compunc when your program generates messages at random. Refresh your browser to try multiple times.
let hypeUp = (name: string): string => {
    let choice: number = random(1, 3);
    if (choice === 1) {
        return name + " IS AWESOME";
    } else {
        if (choice === 2) {
            return name + " IS GOAT";
        } else {
            return "THE REAL MVP IS " + name + "!!!";
        }
    }
};
More function tracing: `print(v(3))`

```javascript
let p = (n: number): number => {
  print("p");
  if (n > 3) {
    return z(n) + n;
  } else {
    return v(n + 1);
  }
};

let v = (n: number): number => {
  let x: number = z(n) + 1;
  print("v");
  return x;
};

let z = (n: number): number => {
  print("z");
  return n;
};
```
What is printed with the call: \texttt{print(p(3))} \\

\texttt{let p = (n: number): number => {} print("p"); if (n > 3) { return z(n) + n; } else { return v(n + 1); } }; \\

\texttt{let v = (n: number): number => { let x: number = z(n) + 1; print("v"); return x; }; } \\

\texttt{let z = (n: number): number => { print("z"); return n; };}
Challenge: What is printed with the call: `print(p(v(4)))`

```javascript
let p = (n: number): number => {
  print("p");
  if (n > 3) {
    return z(n) + n;
  } else {
    return v(n + 1);
  }
};

let v = (n: number): number => {
  let x: number = z(n) + 1;
  print("v");
  return x;
};

let z = (n: number): number => {
  print("z");
  return n;
};
```
We’ve been calling functions since the first day of class!

// Hello, world!
import { print } from "introcs";

export let main = async () => {
    print("Hello");
    print("World");
};

main();
introcs

• introcs is a file containing various helper functions we have been using throughout the semester

• This is the power of functions
  • We can use a function without needing to know about all the gory implementation details

• Abstraction is a key pillar of programming
  • Functions can “abstract away” details and prevent code repetition