24th August 2020

Studio 3
Substitution Model and Recursion

CS1101S AY20/21 SEM 1
Studio 03A

Chen Xihao
Year 2 Computer Science

chenxihao@u.nus.edu
@BooleanValue

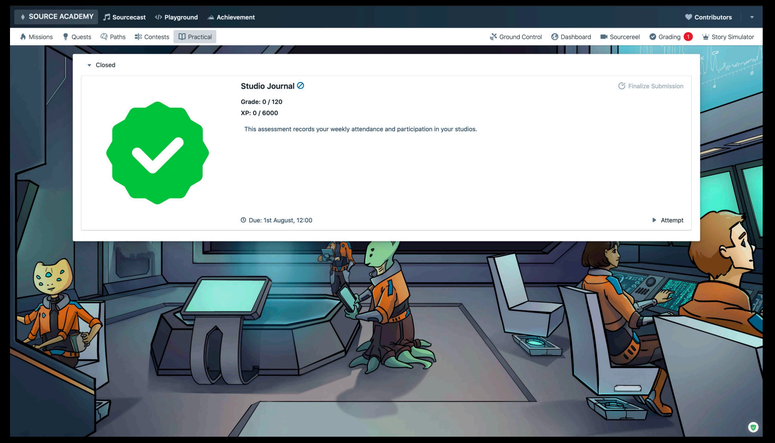
Studio 3
Agenda

- Admin stuff
- Recap
  - Substitution Model
  - Recursion
- Studio sheet
  - Main
  - In-class
  - Extra (if we have time)

Admin and Announcements

Admin stuff

Attendance Taking and Class Participation



- Attendance using a Practical on SA
  - If you are present: you get some XP
- Class participation (XP ranging from 0 - 500 per studio)
  - Being here: 200 (oh no)
  - Contribute reasonably: ~ 300 - 350
  - Contribute exceptionally: ~400
  - You steal my job: 500!
- Consistently get ~400 - high chance to be nominated for Avenger programme next year.

Admin stuff

Styling

If the *consequent-expression* or *alternative-expression* are lengthy, use indentation. The indentation is as usual four characters longer than the indentation of the previous line.

```
// good style
function A(x,y) {
    return y === 0
        ? 0
        : x === 0
            ? 2 * y
            : y === 1
                ? 2
                : A(x - 1, A(x, y - 1));

// bad style: line too long
function A(x,y) {
    return y === 0 ? 0 : x === 0 ? 2 * y : y === 1 ? 2 : A(x - 1, A(x, y - 1));
}

// bad style: too much indentation
function A(x,y) {
    return y === 0
               ? 0
               : x === 0
                     ? 2 * y
                     : y === 1
                           ? 2
                           : A(x - 1, A(x, y - 1));
}
```

Don’t use too many conditional expressions.
Maximum 2 to 3! It gets hard to follow…

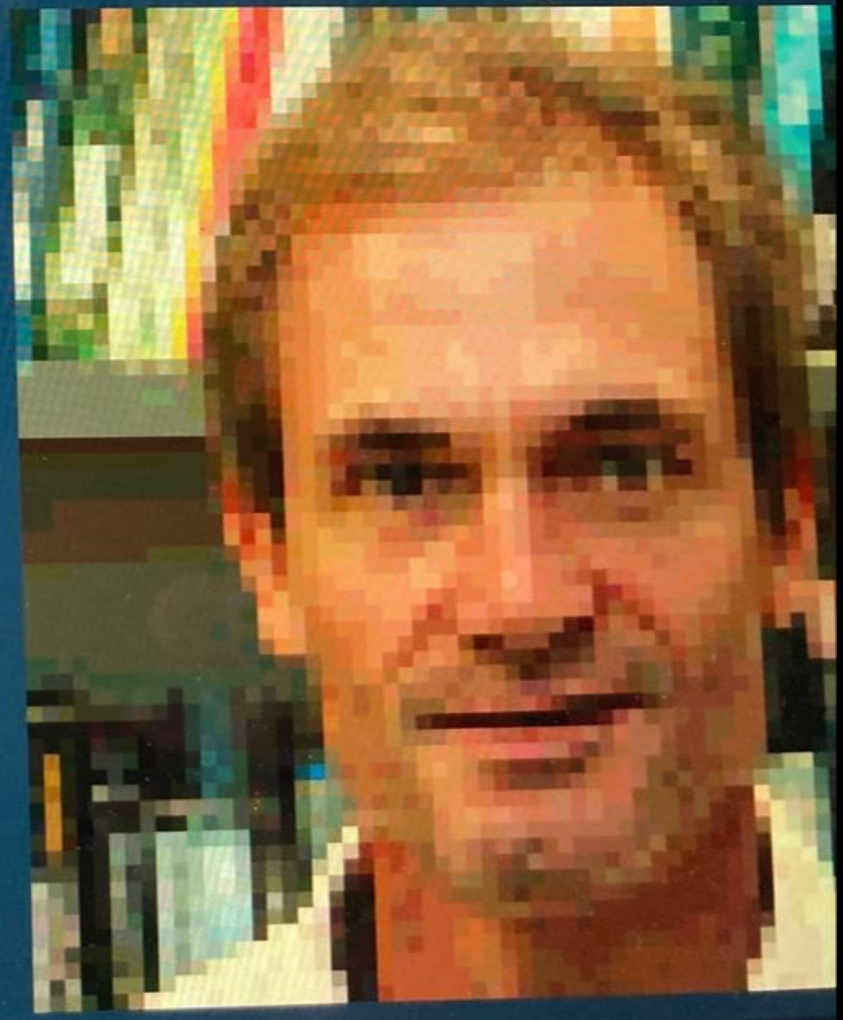
Are you getting tired of all the assignments yet?

Don’t worry, there’s more to come.

Sourcecast | Playground

Source §1 | RUNES | Session | Google Drive | 10000

```
est() {
draw_rune(depth,current_index){
 left = depth === 0 ? (divider, current_search) =>
                    // #basecase This is a function
                       divider === 0.25 ? current_se
                           divider / 2,
                           current_index+stringify(c
                               ? current_search+divi
                               : current_search-divi
                       )
                  // #nextcase Recursion of the next
                   : draw_rune(depth-1,current_index)
 right = depth === 0 ? left(2048,3072) // #basecase
                       // #nextcase Recursion of nex
                     : draw_rune(depth-1,left(true))

 x => depth === 0                          //#basecase
      ? x ? current_index + stringify(right) //retu
          : color(square,
                  right / 4096 - 0.25, //generate a
                  right % 256 / 256,
                  right % 16 / 16
          )
                            //#nextcase
      : x ? right(true) //return next index
          : quarter_turn_right( //return runes
              stack(
                  left(false),
                  right(false)
              )
          );
```



Recap: The Substitution Model

Substitution Model

Basics

- Just a form of mental model to understand stateless programming
    - What is “stateless”?
        - Names (keys) declared will always hold that value (key-value pair)
        - `const my_name = “xihao”;`
        - `function say_hello() { return my_name; }`

Substitution Model

Evaluation of Function Application

- Evaluate `some_func(5+3)`
  - `some_func(8);`
  - `> return square(8) + 3;`
  - `> return (8 * 8) + 3;`
  - `> return 64 + 3;`
  - `>> 67;`
- Sounds easy?

```
function square(x) {
    return x * x;
}
function some_func(x) {
    return square(x) + 3;
}
```

Substitution Model

Applicative Order vs Normal Order Reduction

- Source:
  - Source uses *Applicative Order Reduction*.
  - Expressions in arguments are evaluated first before applying the function to the arguments' exact values
  - (just for understanding: applicative order requires the arguments to be in their "most simplistic / basic form" before the function is applied).
  - Arguments can be expressions, primitives, runes, functions, or many more!
  - Parameters are just some names.

“Before cutting cakes, you need to know how many cakes you have and how many slices to cut into.”

- your avenger

Substitution Model

Applicative Order vs Normal Order Reduction

- First mission (Rune Trials) Q4:

```
function transform_mosaic(r1, r2, r3, r4, transform) {
    return transform(
        mosaic(r1, r2, r3, r4)
    );
}
```

- `transform_mosaic(rcross, sail, corner, nova, make_cross);`

Substitution Model

Applicative Order vs Normal Order Reduction

- `transform_mosaic(rcross, sail, corner, nova, make_cross);`
  - `> return make_cross(mosaic(rcross, sail, corner, nova));`
  - `> return make_cross(some_rune);`
  - `>> some_other_rune`
- In an essence:
  - `r1=rcross, r2=sail, r3=corner, r4=nova`
  - `transform=make_cross`

Substitution Model

Applicative Order vs Normal Order Reduction

- But why don’t we just substitute `5+3` directly?
- Evaluate ` some_func(5 + 3); `
  - `> some_func(5 + 3);`
  - `> return square(5 + 3) + 3;`
  - `> return ( (5 + 3) * (5 + 3) ) + 3;`
  - `> return ( 8 * (5 + 3) ) + 3;`
  - `… > return 64 + 3;`
  - `>> 67`

```
function square(x) {
    return x * x;
}
function some_func(x) {
    return square(x) + 3;
}
```

Actually, WE CAN!

Substitution Model

Applicative Order vs Normal Order Reduction

- This is called “Normal Order Reduction”
  - Perform the substitution before finding the exact value of the arguments.
  - In technical terms: “normal-order languages delay the evaluation of arguments until the argument values are needed”
  - Sth sth this is lazy…
    - we’ll get to that in the following lectures (and CS2030)

Substitution Model

Applicative Order vs Normal Order Reduction

- Evaluate `some_func(5 + 3);`
  - `> some_func(5 + 3);`
  - `> return square(5 + 3) + 3;`
  - `> return ( (5 + 3) * (5 + 3) ) + 3;`
  - `> return ( 8 * (5 + 3) ) + 3;`
  - `… > return 64 + 3;`
  - `>> 67`

```
function square(x) {
    return x * x;
}
function some_func(x) {
    return square(x) + 3;
}
```

Quiz

Applicative Order vs Normal Order Reduction

- What's the order of reduction used here?
- Evaluate: ` foo(2, 3, bar); `
  - `> 2 * bar( 2 - 3 ) + bar( 2 + 3 );`
  - `> 2 * bar(-1) + bar( 2 + 3 );`
  - `> 2 * ( -1 + 38 ) + bar( 2 + 3 );`
  - `> 2 * 37 + bar(2 + 3);`
  - `> 74 + ( (2 + 3) + 38 );`
  - `> 74 + ( 5 + 38 );`
  - `> 74 + 43;`
  - `>> 117`

```
function foo(a, b, func) {
    return a * func(a - b) + func(a + b);
}
function bar(x) {
    return x + 38;
}
```

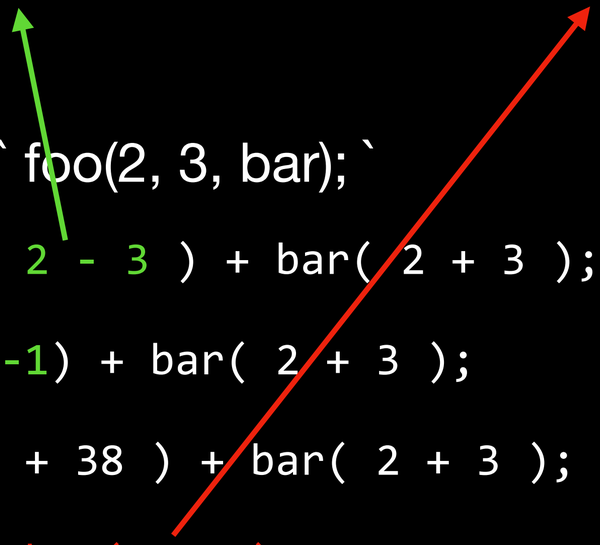
Answer:

NONE

OOPS :)

Quiz

Applicative Order vs Normal Order Reduction



- Evaluate: ` foo(2, 3, bar); `
- > 2 * bar(2 - 3) + bar(2 + 3);
- > 2 * bar(-1) + bar(2 + 3);
- > 2 * (-1 + 38) + bar(2 + 3);
- > 2 * 37 + bar(2 + 3);
- > 74 + ((2 + 3) + 38);
- > 74 + (5 + 38);
- > 74 + 43;
- >> 117

```
function foo(a, b, func) {
    return a * func(a - b) + func(a + b);
}
function bar(x) {
    return x + 38;
}
```

Substitution Model in Mission 1

Substitution Model

Mission 1 Question 4

- Write a function that takes five arguments:
  - Four runes and a transformation function.
- The function should create a mosaic using the runes and then apply the given transformation function to it and return the resulting image.

```
function transform_mosaic(r1, r2, r3, r4, transform) {

    // your answer here

}
```

Substitution Model

Mission 1 Question 4

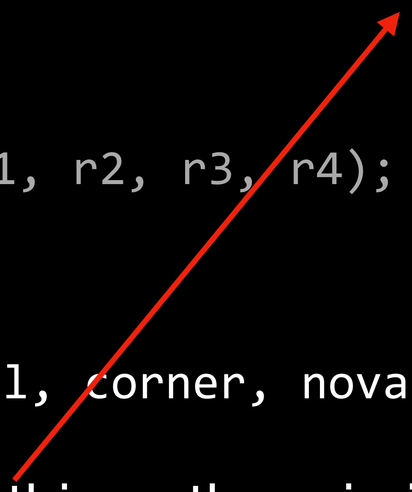
- Common issues:
  - Changing the parameter name `transform` to `make_cross`

```
function transform_mosaic(r1, r2, r3, r4, make_cross) {
    return make_cross(mosaic(r1, r2, r3, r4);
}
```

Substitution Model
Mission 1 Question 4

- Problem with doing this: what if I want to use the primitive function `make_cross`?

```
function transform_mosaic(r1, r2, r3, r4, make_cross) {
    return make_cross(sail);
    // return make_cross(mosaic(r1, r2, r3, r4);
}
show(transform_mosaic(rcross, sail, corner, nova, turn_upside_down));
```

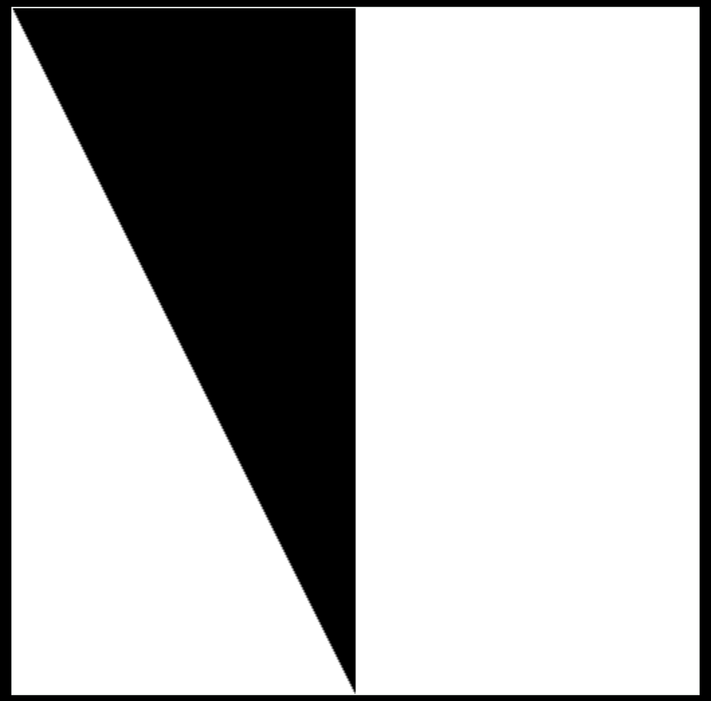


- So does this “make_cross” refer to this or the primitive `make_cross` function?
  - *Let’s test!*

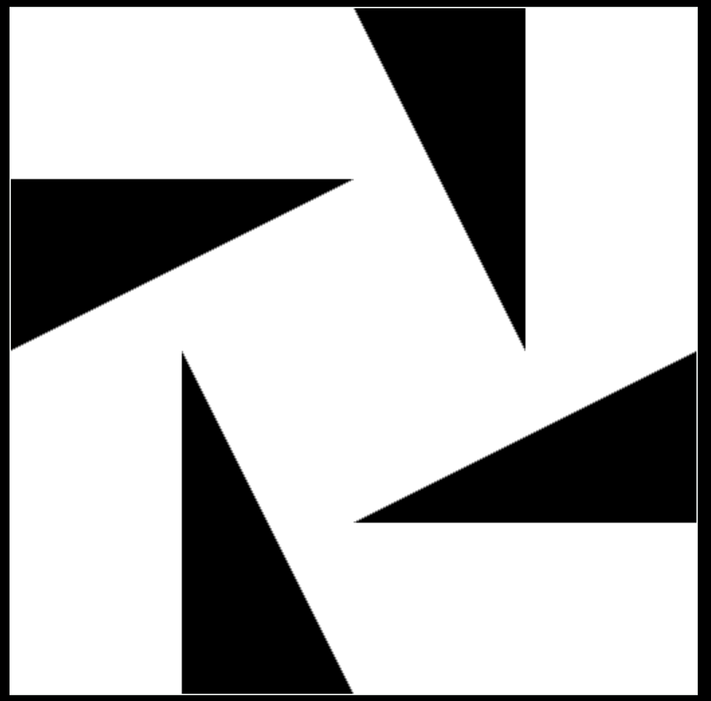
Substitution Model

Mission 1 Question 4

- Turns out it shows this
  - `turn_upside_down(sail)`



- Instead of this:
  - `make_cross(sail)`



Substitution Model

Mission 1 Question 4

```
function transform_mosaic(r1, r2, r3, r4, make_cross) {
    return make_cross(sail);
}
```

- Why?
  - Let’s apply the substitution model!
  - `> transform_mosaic(rcross, sail, corner, nova, turn_upside_down);`
  - `> return turn_upside_down(sail);`
  - `>> (sail turned upside down)`
- Recall: parameter names are similar to constant declarations!
- So, the name `make_cross` is now bound to `turn_upside_down` instead!
- (this is a super bad practice! we’ll learn more about this later in Scoping Rules!)

Substitution Model
Mission 1 Question 4

- Correct answer:

```
function transform_mosaic(r1, r2, r3, r4, transform) {
    // calling make_cross here refers to the primitive rune function!
    return transform(mosaic(r1, r2, r3, r4));
}
```

- Guiding rule: leave the function heading (function / param names) the same as the template!
  - The names are chosen by the Profs / head Avengers and they are for a reason
  - Unless the function uses parameters such as “m”, “n”, then feel free to change them if you think it’s clearer!

Substitution Model

Mission 1 Question 4

- Recap on function declaration:
  - `function foo(a, b, c) { return a * b + c; }`
  - During declaration, we don’t know what a, b, c are!
  - They are only bound to values when we call the function and supplying arguments
    - `foo(1, 42, 117);`

Substitution Model

Mission 1 Question 4

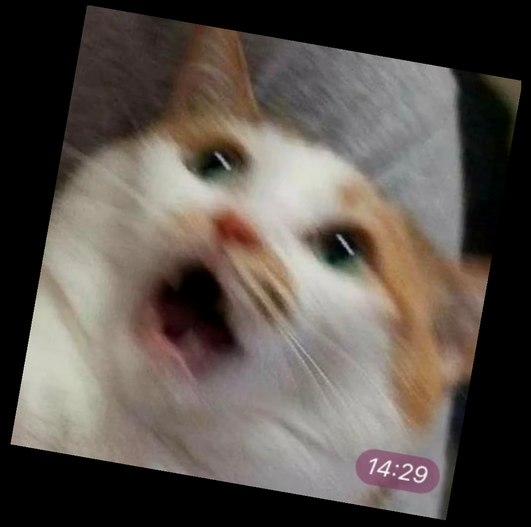
- Mathematically:
  - `f(a, b, c) = a * b + c`
  - Evaluating f(1, 42, 117) = 1 * 42 + 117 = 159
- We <u>don't</u> define math functions using some values:
  - `f(1, 2, 3) = 1 * 2 + 3`
  - if you do this, pls go back to high school -.-||
- Similarly, we don't define programming functions using "values"

Substitution Model

Applicative Order vs Normal Order Reduction

- If you still have doubts or need examples:
  - bit.ly/app_vs_norm_reduc
  - or search **@44** in piazza to go directly to my explanation / example
  - ~~and watch me get wrecked by prof Henz~~

Recap: Recursion

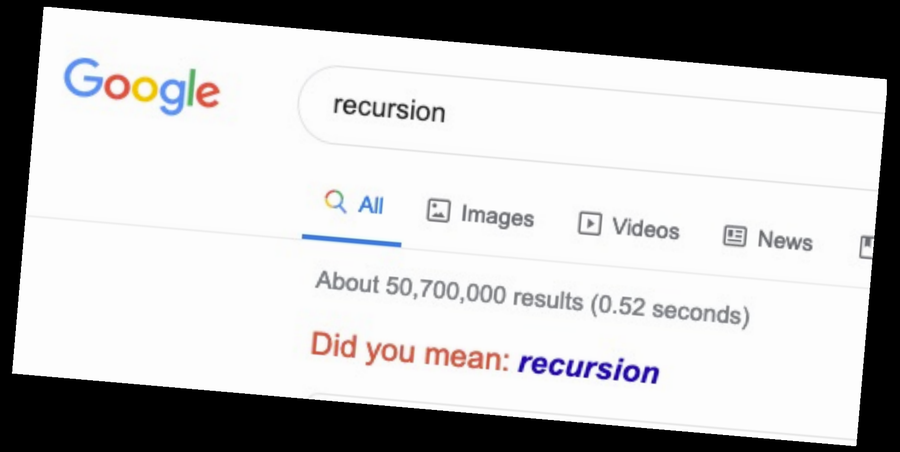


“To understand recursion, one must first understand recursion.”

- idk some dude on the internet

Recursion

Linear Recursion and Iteration



- Definition of recursion:
  - “the repeated application of a recursive procedure or definition”
- In computing:
  - Recursion is when something is defined in terms of itself

Recursion

Simple Recursion - the Factorial

- FACTORIAL!
  - `n! = n * (n-1) * (n-2) * … * 1`
  - `n! = n * (n-1)! = n * ( (n-1) * (n-2)!) = …`
- Similarly if we use a function to represent ‘!’:
  - `factorial(n)`
  - `= n * factorial(n-1)`
  - `= n * ( (n-1) * factorial(n-2) ) = …`
  - high school mathematics!

Recursion

Simple Recursion - the Factorial

- First try:

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

- Does this work?

Recursion

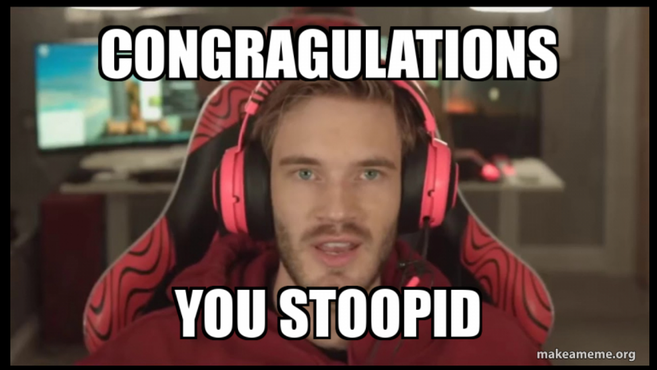
Simple Recursion - the Factorial

- What happens when (n = 0) ?
- Simple example: `factorial(2)`
  - `> factorial(2);`
  - `> 2 * factorial(1);`
  - `> 2 * 1 * factorial(0);`
  - `> 2 * 1 * 0 * factorial(-1);`
- WAIT A SECONDDDDDD…

Recursion

Simple Recursion - the Factorial

- Definition: “the factorial of a positive integer n, is the product of all positive integers less than or equal to n”
- Ending condition!!! (aka base case)
  - Turns out the computer doesn’t know when to stop
  - Specify when we want to terminate



Recursion

Simple Recursion - the Factorial

- Definition: “the factorial of a positive integer n, is the product of all positive integers less than or equal to n”
- New structure (let’s assume the input is always correct):
  - `IF n = 0 or n = 1 THEN return 1`
  - `IF n > 1 THEN return `n * factorial(n-1)``

Recursion

Simple Recursion - the Factorial

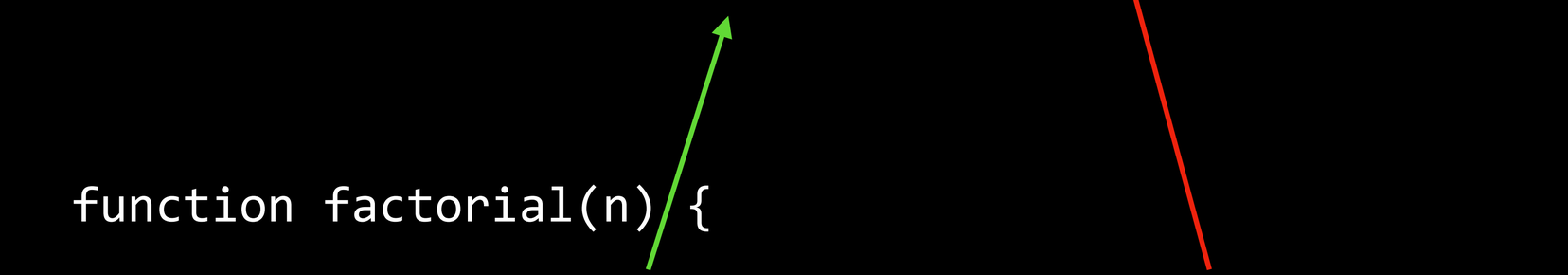
- Definition: “the factorial of a positive integer n, is the product of all positive integers less than or equal to n”
- In Source:

```
function factorial(n) {

    return (n == 0 || n== 1)    // or (n <= 1)

        ? 1

        : n * factorial(n-1);

}
```

Deferred operations!

Recursion
Summary

- You have some heavy-duty task
- Suppose I tell you I can magically do the subsequent iterations for you (huge problem)
- Now you only need to handle the current iteration (small problem)



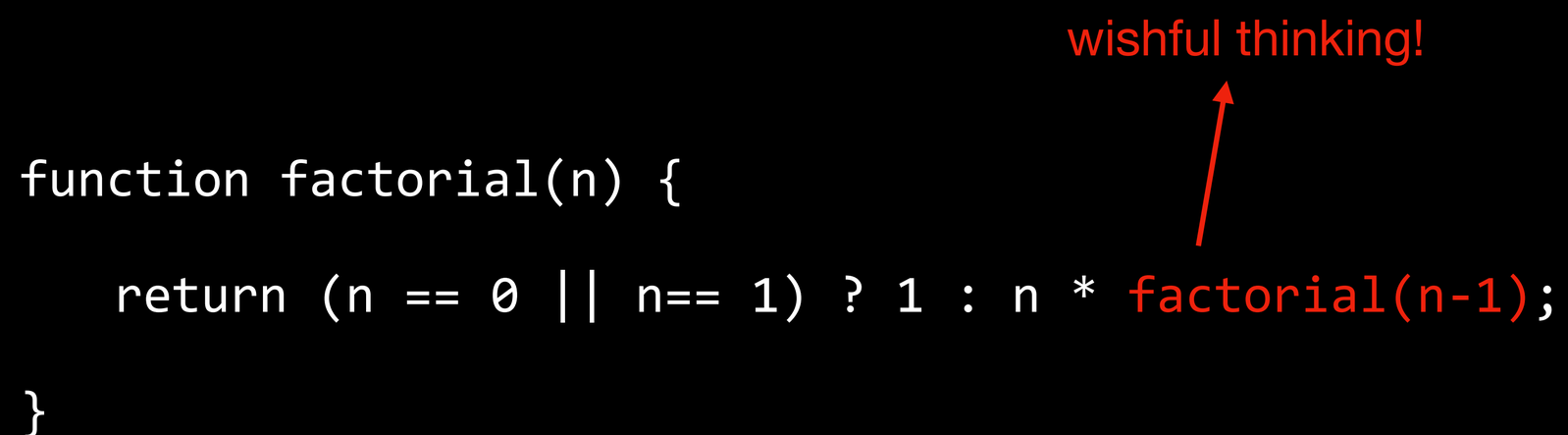
```
function factorial(n) {

    return (n == 0 || n== 1) ? 1 : n * factorial(n-1);

}
```

“Wishful thinking!”

- Prof Henz, Martin J.

Recursion
Summary

- Wishful thinking:
  - The mindset you should take when employing the recursive process
  - Assume that the bigger, subsequent problem will be taken care for you



credits: r/ProgrammerHumor

Recursion

Iterative Factorial

- Perform factorial iteratively:
  - Instead of “chaining” and having deferring the operations, let’s calculate whatever we know first!
  - Hence, we need something to “carry” the values that we’ve already-evaluated to the next iteration

Recursion

Iterative Factorial

- Carry what's already-evaluated as an argument!

```
function factorial_iter(n, current_product) {
    return (n === 0 || n === 1)
        ? current_result
        : factorial_iter(n-1, current_product * n);
}
```

- Notice we aren't "chaining" the evaluations
- Instead, we carry `current_product * n` as an argument

Recursively:

```
function factorial(n) {
    return (n == 0 || n== 1)
        ? 1
        : n * factorial(n-1);
}
```

Recursion

Iterative vs Recursive Processes

- Iterative process
  - More “diligent”
  - No deferred operations
  - Evaluates the currently know values first before going on to the next iteration
  - Uses less space
  - Hardworking student who finishes work immediately
- Recursive process
  - More “lazy”
  - Have deferred operations
  - Chains the operations to be calculated at once later
  - Uses more space
  - Lazy ass who dumps all the holiday assignments until the end to do at once

Recursion

Recursive functions? Recursive process? HUH???

- Remember the definition: “defined using itself”
  - Recursive functions can give rise to different processes!

```
function sum(n) {

  return n === 0

    ? 0

    : n + sum(n-1);

}
```

recursive process

```
function sum(n) {

    function helper(n, total)

        n === 0 ? 0 : helper(n-1, total + n);

    }

    return helper(n, 0);

}
```

iterative process

Recursion

Recursive functions? Recursive process? HUH???

- Key point: Recursive function !== Recursive process

```
function sum(n) {
  return n === 0
    ? 0
    : n + sum(n-1);
}
```

recursive process

```
function sum(n) {
    function helper(n, total)
        n === 0 ? 0 : helper(n-1, total + n);
    }
    return helper(n, 0);
}
```

iterative process

End of Recap

Any questions?