Welcome to the Functional Programming *Power Monday*! This first lecture in the series begins not with functional programming per se, but a bit of background context.

Introduction to Programming Paradigms

And the Myths Thereof

Today we're going to begin by discussing broad categories or styles of programming languages, called paradigms.

You may have heard terms like this…


Functional!
Object Oriented!
Procedural!
Declarative!

Perhaps you felt like this…


What do these words
even mean?

paradigm

noun | par·a·digm | \ˈper-ə-ˌdīm , ˈpa-rə- *also* -ˌdim\

"…3. a philosophical and theoretical framework of a scientific school or discipline…"

MERRIAM WEBSTER

- **Broad categories of programming languages**
- **Traditionally viewed as competing styles** 🤔
- **May have very different syntax, capabilities, goals, and concepts**

We consider these buzzwords to be examples of *paradigms*.

(Some) Oft-Cited Examples

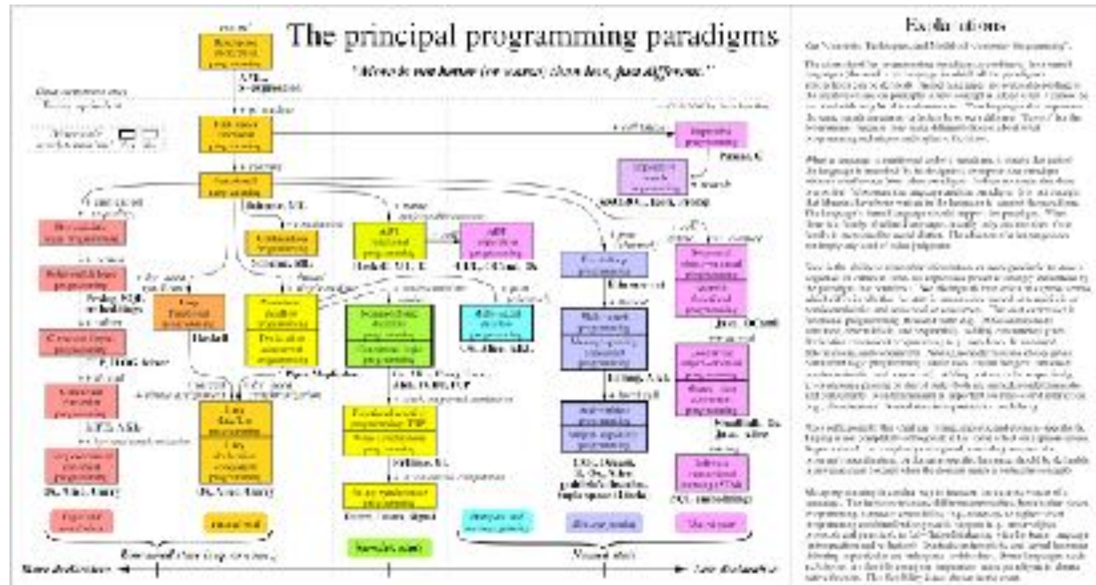
Paradigm	Languages
Procedural	FORTRAN, ALGOL, COBOL, C, BASIC
Object Oriented	Simula, Smalltalk, Self, C++, Java, Ruby, Python
Functional	Lisp, Scheme, OCaml, Haskell, F#, Elm, ReasonML
Declarative	SQL, HTML, RegEx, Wolfram

Some languages were either built for, or considered to be archetypes of, particular paradigms.

But Wait, There's More!™

Lazily Evaluated
Purely Functional
Concatenative
Procedural
Stateful
Structured
Logic
Concurrent
Markup
Toy
Eagerly Evaluated
Esoteric
Imperative
Statically / Dynamically Typed
Reactive
Strongly / Weakly Typed
Impure Functional
Symbol
Proof Systems

A Large and Partly Arbitrary Topic



Before we really get into what a paradigm is, a disclaimer – classifying programming languages into neat little boxes is doomed to failure, as the boxes are not perfectly defined, and the languages don't play by such rigid rules. (Image from Peter Van Roy's book "Concepts, Techniques, and Models of Computer Programming".

WHAT ABOUT JS ?

*“JavaScript is a prototype-based, **multi-paradigm**, dynamic language, supporting **object-oriented**, **imperative**, and **declarative** (e.g. **functional** programming) styles.”*

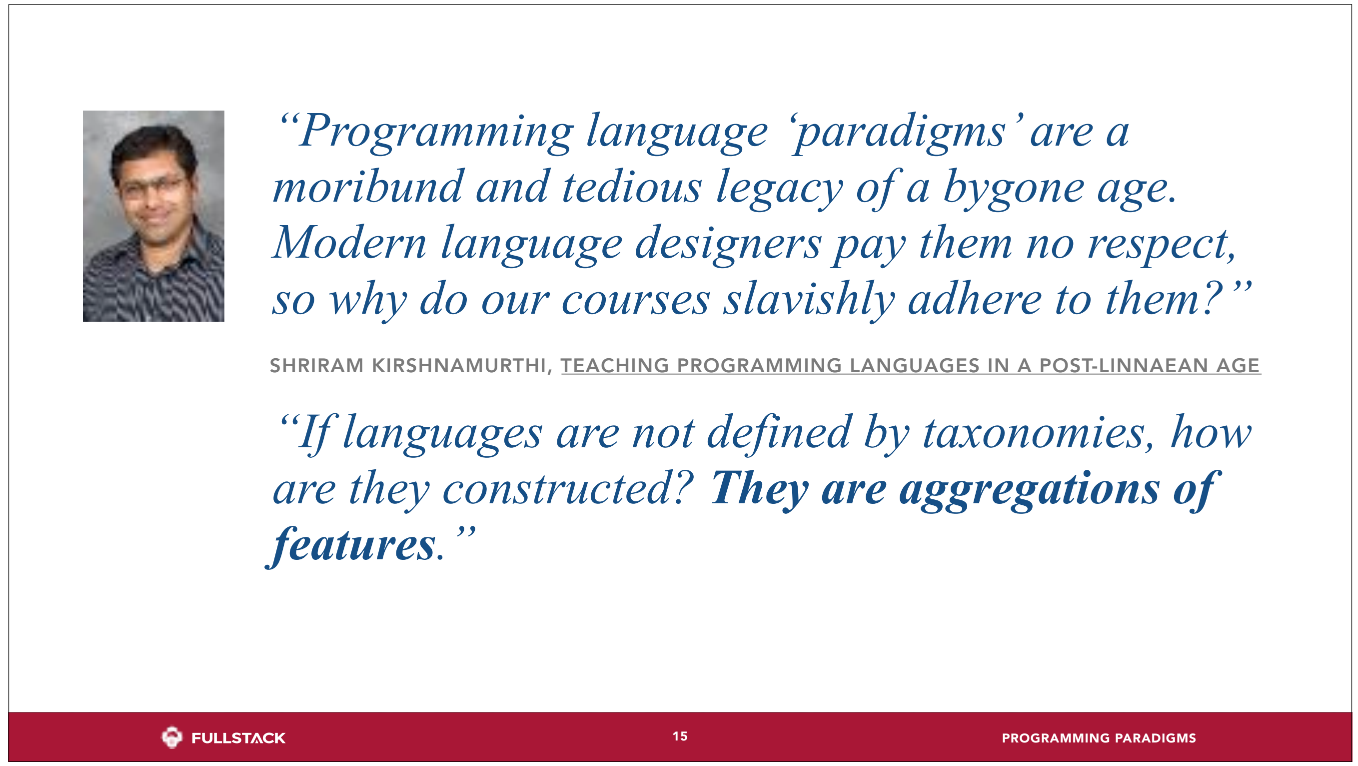
MDN

…literally the first paragraph on the MDN page introducing JS.

SAY WHAT!?

Multi-Paradigm

- In fact, many modern languages cannot be easily placed into sharply-delineated paradigms.
- JS, Java, C++, Python, Swift, and <u>many</u> others blur the lines and / or support multiple approaches.
- Paradigms themselves are hard to define and mean different things to different sources.



Real languages are specific mixtures of certain features / capabilities rather than easily classifiable entities. (*Side note, this is actually similar to how some programming trends favor mixins over class inheritance.*)

A Random Set of Language Features

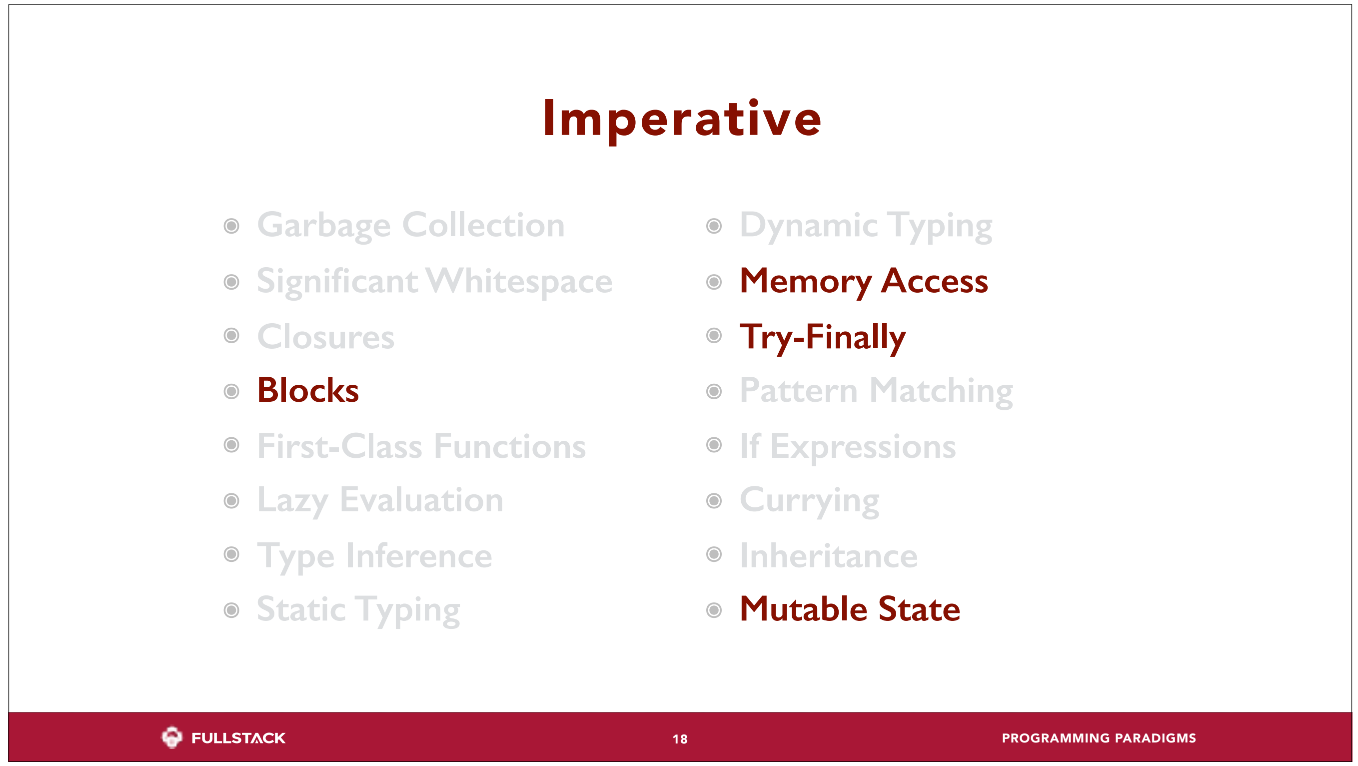
- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State

Here is a random collection of features…

Structured

- Garbage Collection
- Significant Whitespace
- Closures
- **Blocks**
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- **Try-Finally**
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State

Some of these features might be considered to fall within a given paradigm



But a given feature might be often found across multiple paradigms

Object-Oriented

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing*
- Dynamic Typing
- Memory Access*
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- **Inheritance**
- Mutable State*

Declarative

- **Garbage Collection**
- Significant Whitespace*
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference*
- Static Typing*
- Dynamic Typing
- Memory Access
- Try-Finally
- **Pattern Matching**
- If Expressions*
- Currying
- Inheritance
- Mutable State

Is significant whitespace declarative? What about if-expressions (which produce a value)? These features may be found in non-declarative contexts.

Functional

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference*
- Static Typing*
- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State

If a programming language lacks pattern matching, is it no longer functional? Trying to define paradigms in terms of features is a flawed approach.

JavaScript

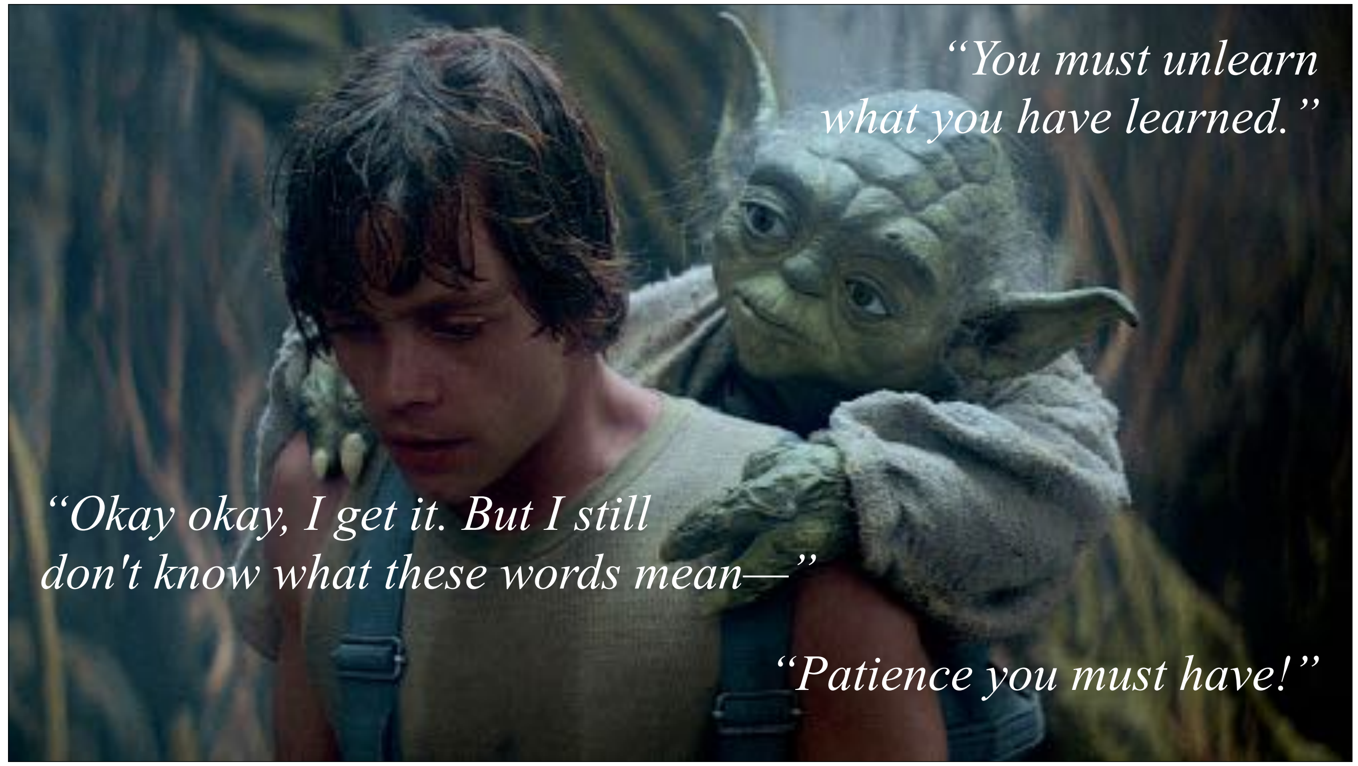
- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching*
- If Expressions
- Currying*
- Inheritance
- Mutable State

In contrast, languages definitely **DO** or **DO NOT** have a certain feature, or **CAN** emulate a feature (asterisks). This makes it a lot more concrete to classify a language via features.

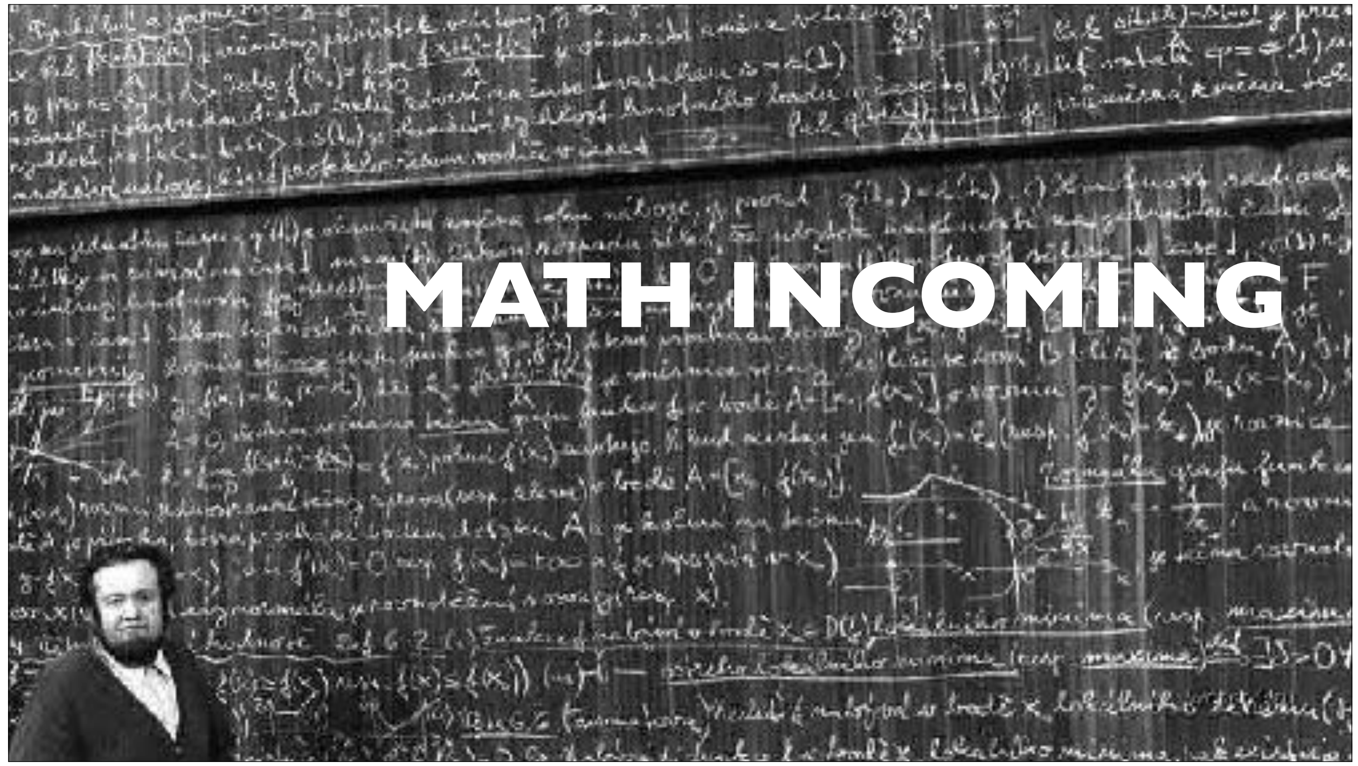
Haskell

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks*
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access*
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State*

Notice that Haskell and JS share some features, typically considered functional. But they definitely have differences too. And Haskell has things beyond the "functional" features.


“You must unlearn
what you have learned.”
“Okay okay, I get it. But I still
don't know what these words mean—”
“Patience you must have!”

Paradigms: Imperative, Structured, Procedural



To help us understand how computers and by extension computer languages work, let's use a small example algorithm.

Add numbers from 1 to n (exclusive?)

take a moment...

O(n): sum = 1 + 2 + 3 + … + (n - 1)

O(1): sum = n * (n - 1) / 2

(we'll do the naive way, for the sake of demonstration)

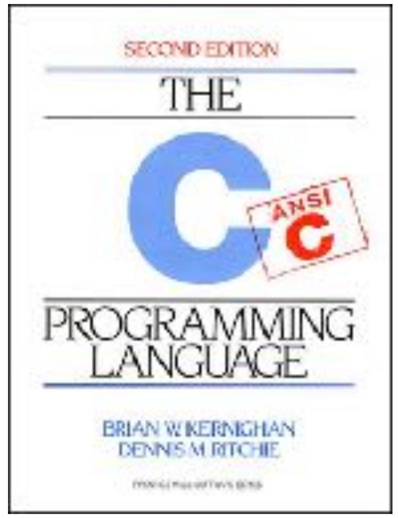
This is a straightforward problem. The mathematician Carl Friedrich Gauss (1777-1855) famously solved it as a young boy in primary school. We'll use a more naive solution for demonstration purposes.

Example JS Program (Fragment)

```
function sumSeries (n) {
    let sum = 0;
    for (let i = 1; i !== n; i++) {
        sum += i;
    }
    return sum;
}

const res = sumSeries(4)
// do something with res (= 6)
```


Here's how one might code the naive solution in JS.

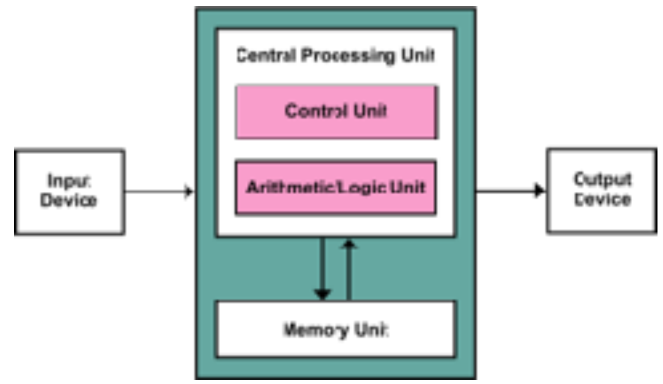
Example C Program (Fragment)




```c
int sumSeries(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}

int main(void) {
    int res = sumSeries(4);
    // use res (= 6) somehow
}
```


…and here is an equivalent solution in C. Note that JS borrowed its syntax from Java, which borrowed its syntax from C. So it's no wonder that the two snippets are almost identical.

Let's get closer to the metal



“The metal” refers the CPU. Virtually all commercial computers use the Von Neumann Architecture, in which the CPU accesses addressable memory – fetching data, computing results, and storing data.

x86 Assembly

```
series:   mov ecx, 1
          xor eax, eax
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          rep ret
main:     mov edx, 4
          call series
          ; use eax somehow
          ret
```

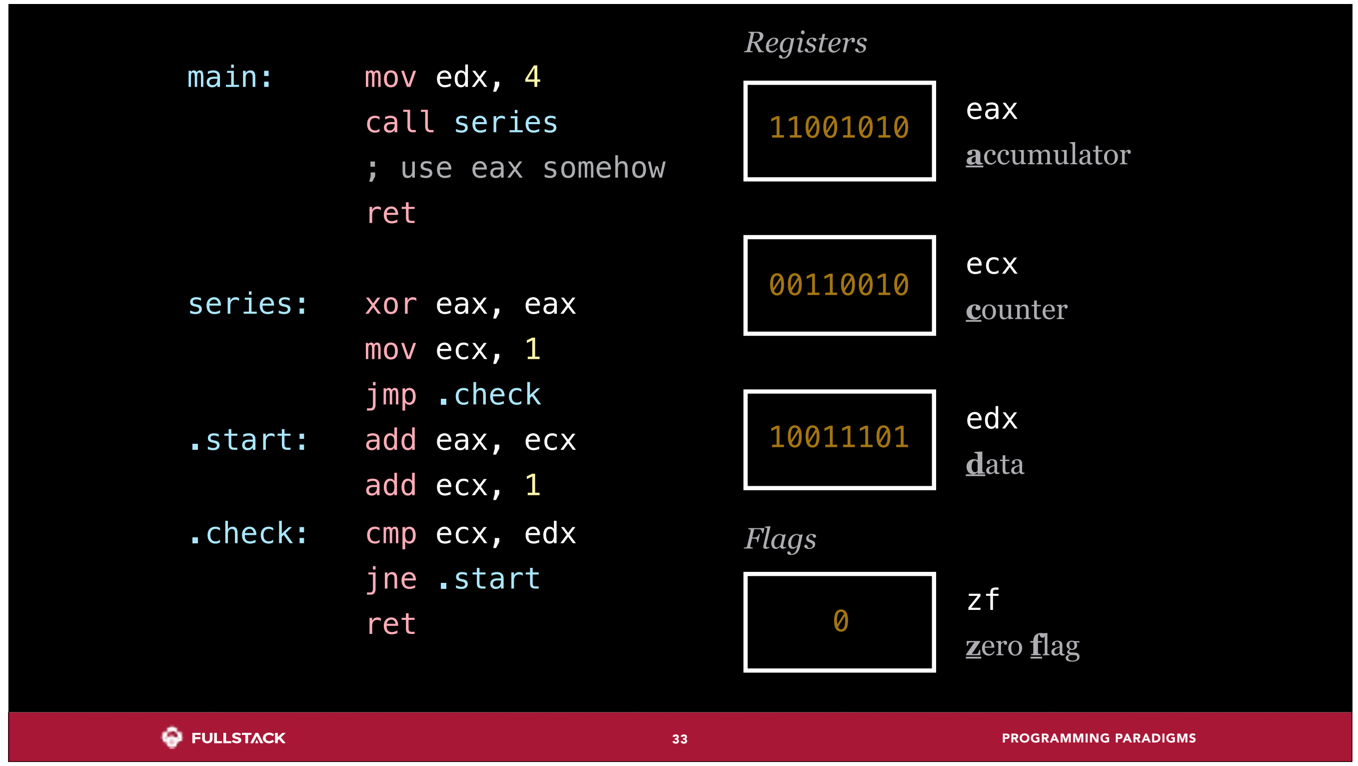
C

```c
int sumSeries(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}

int main(void) {
    int res = sumSeries(4);
    // use res (= 6) somehow
}
```

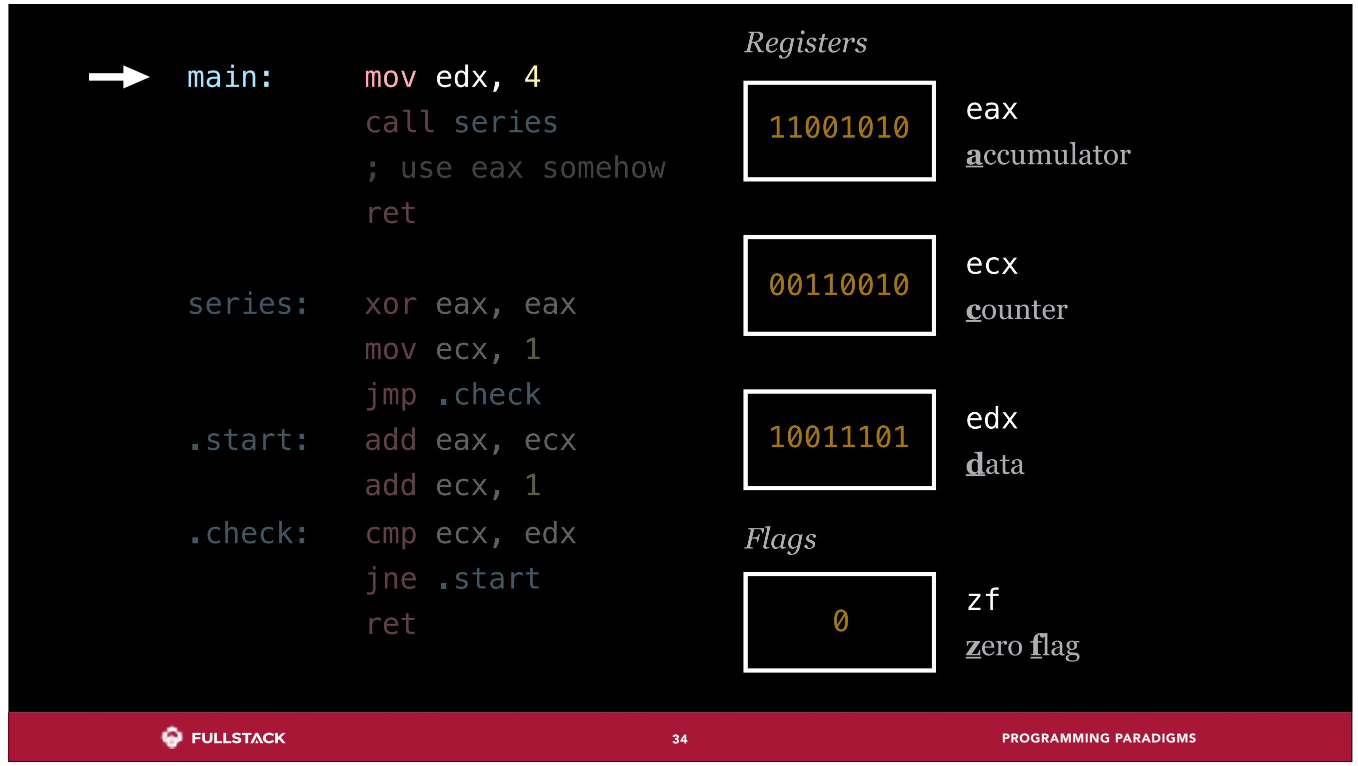

C is a high-level language which gets compiled to *machine code* – the actual signals which direct the computer to perform actions. We can express the same concepts using textual *assembly language* which is a 1-1 mapping of human-readable commands to machine code.

How the Machine Works

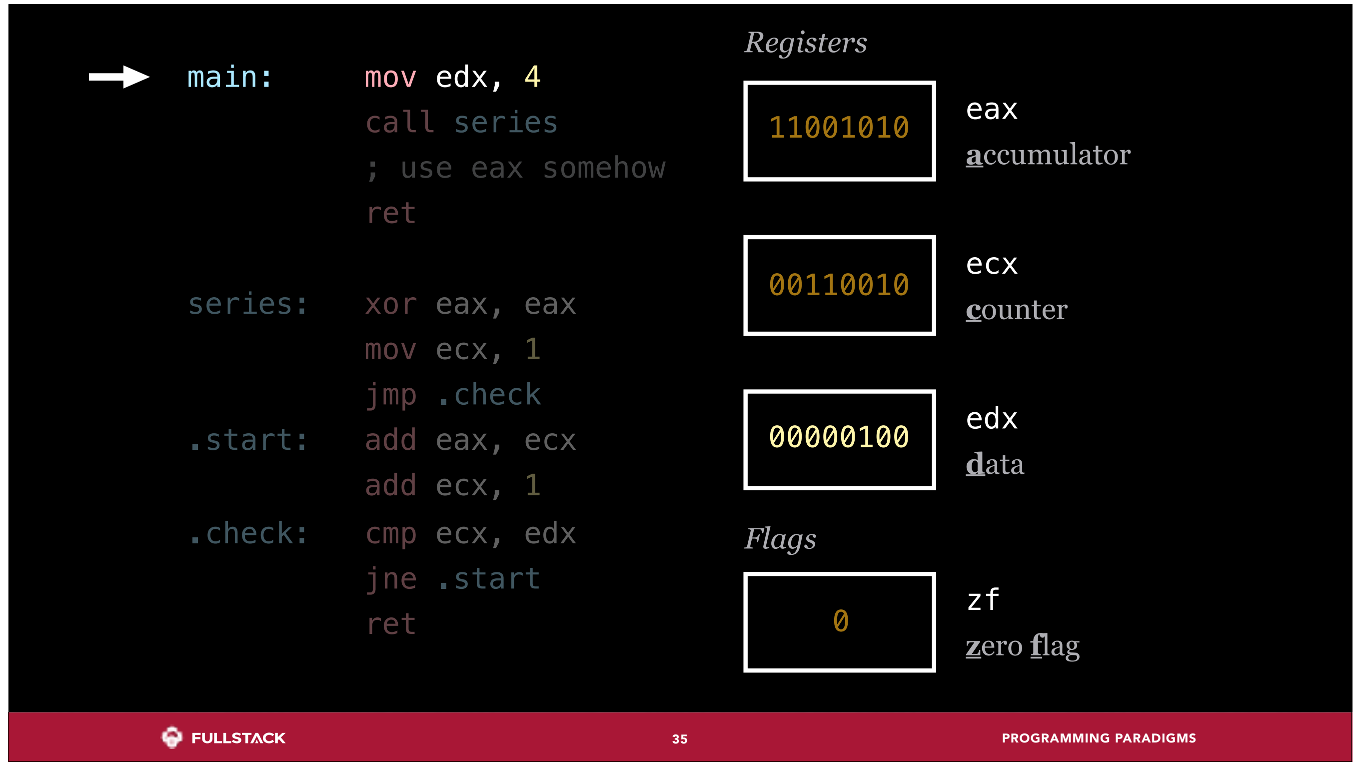
Let's see a simulated computer run this code to get a sense for what is really happening.



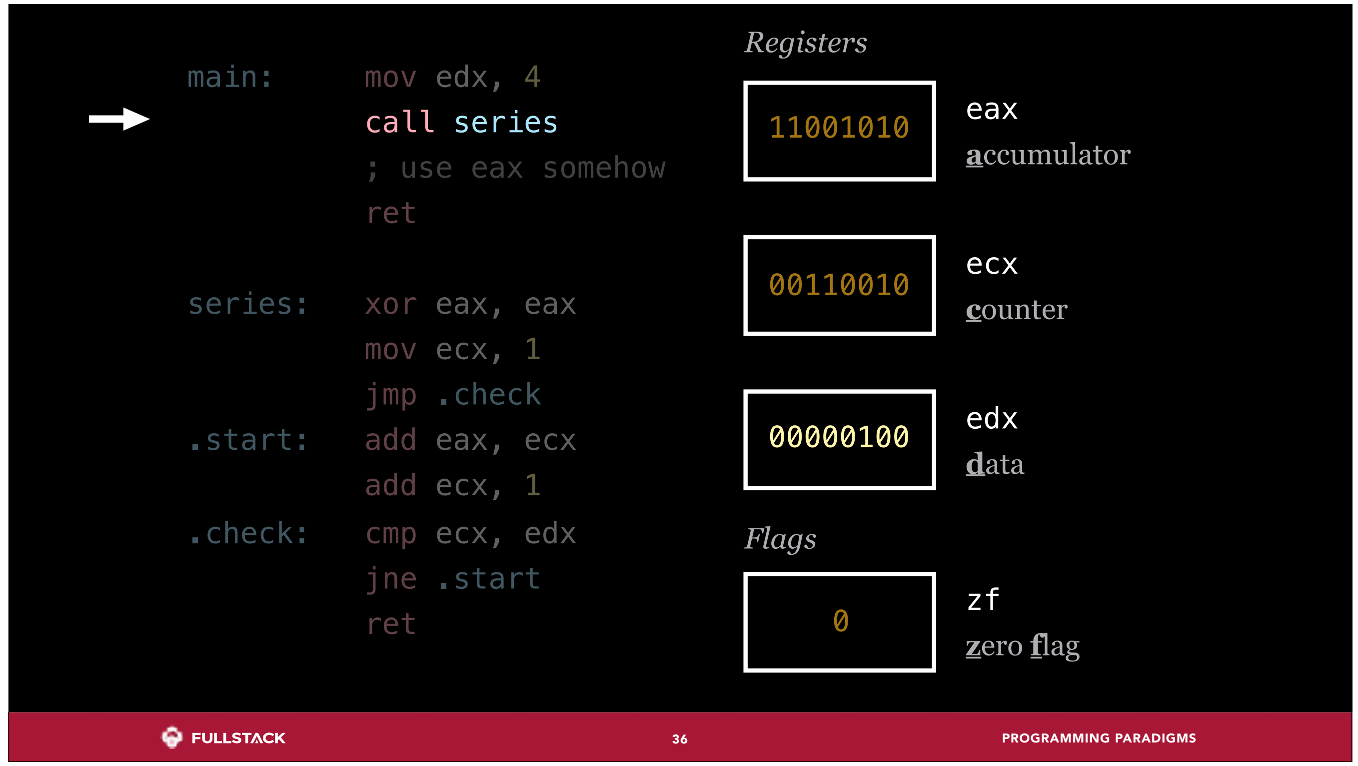
A CPU has a limited set of named *registers,* physical storage buckets which hold a little bit of data at a time. It also has a *flag* (actually one digit from a flags register) for storing the results of (in)equality checks.



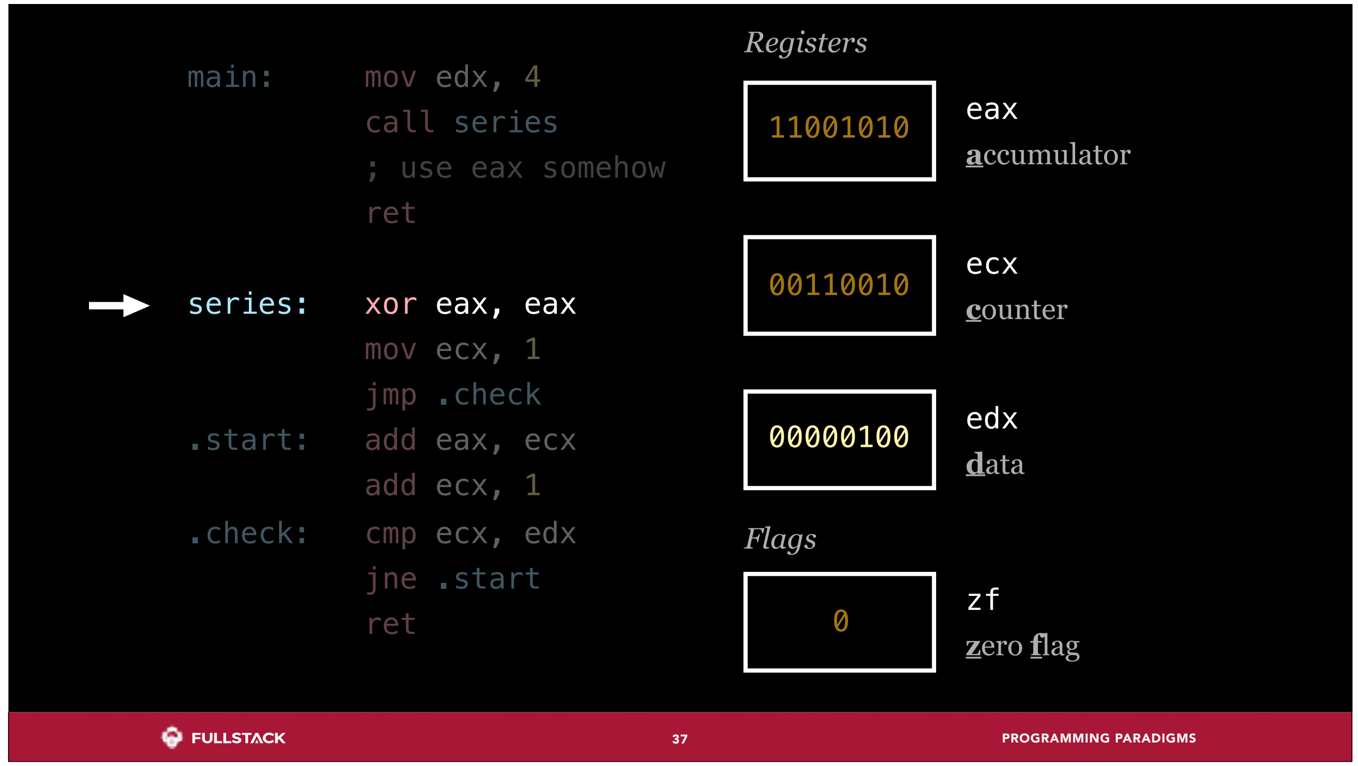
The program is stepped through in order according to a *program counter* which simply increments through the commands one by one. This command is "move (i.e. store) the number 4 in the register edx".



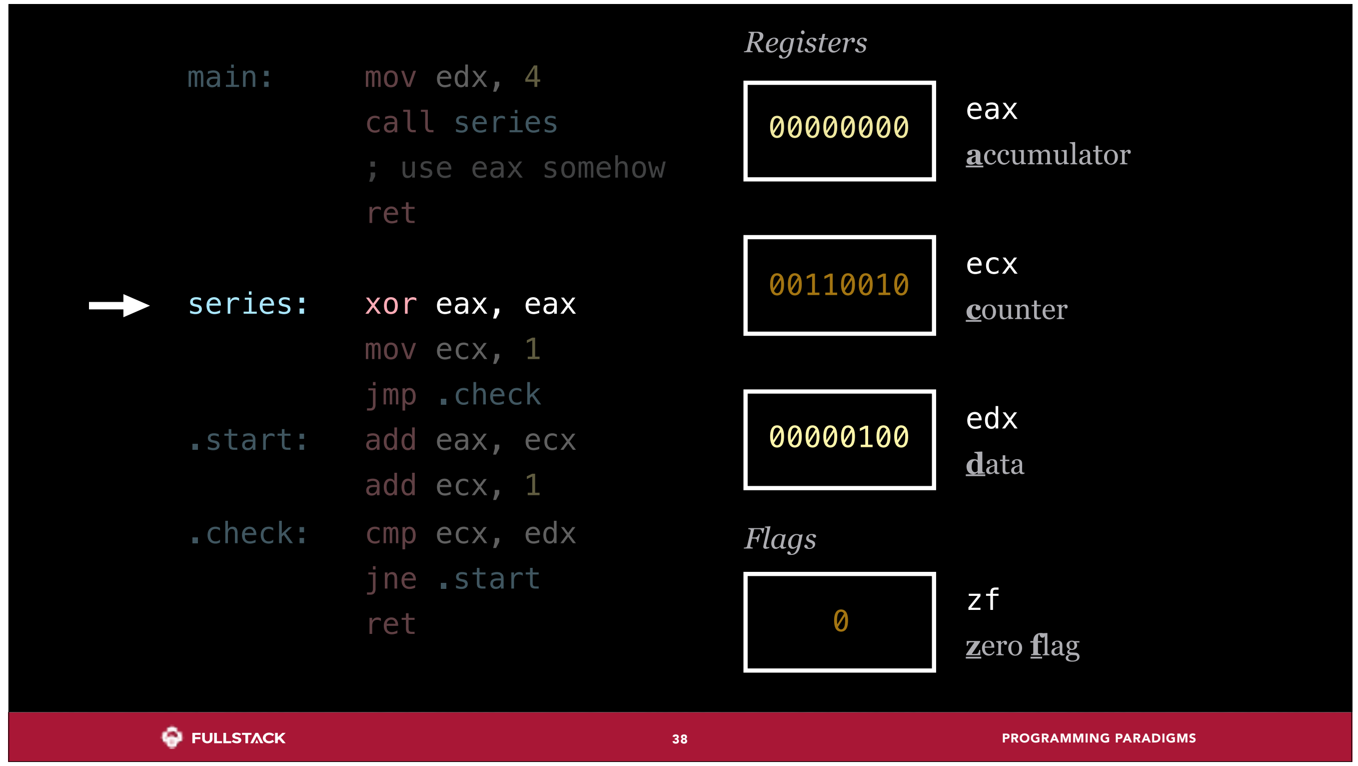
We will use edx (the "data" register) as our function argument / loop limit.



`call` invokes a *subroutine*. Subroutines are functions, but much less capable than JS functions – they are better considered *procedures* and are used primarily for control flow, not as data.



We're going to zero out the accumulator register, which we will use as our "sum".



`xor`-ing a value against itself yields 0. We could have also `mov`ed 0, but xor is faster.

```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
→         mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

Registers

00000000	eax accumulator
00110010	ecx counter
00000100	edx data

Flags

0	zf zero flag

Move the number 1 into ecx (counter). Ecx will be our i value.

```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
       →  mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

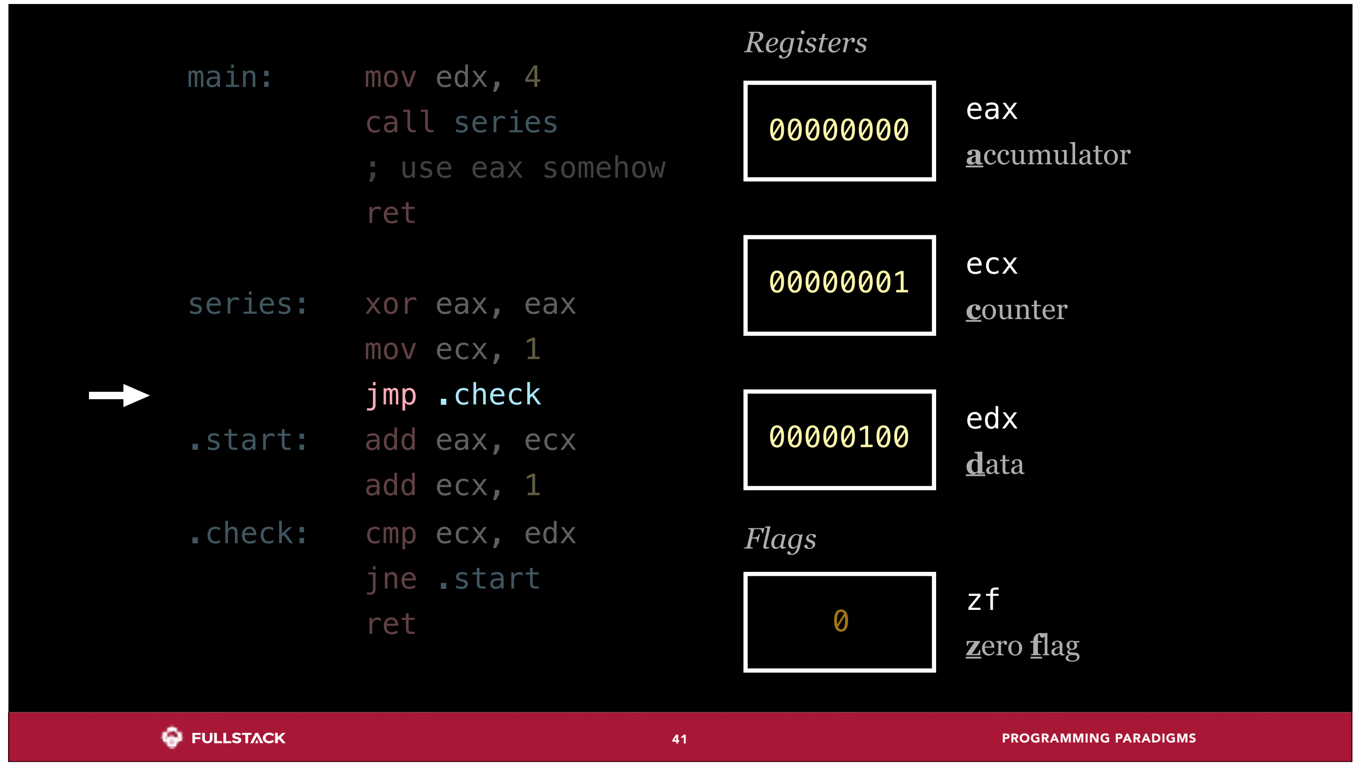
Registers

00000000	eax	**a**ccumulator
00000001	ecx	**c**ounter
00000100	edx	**d**ata

Flags

0	zf	**z**ero **f**lag

We initialized the counter i to 1.



Jump commands, i.e. GOTO statements, tell the program counter to be moved to a different address.

```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

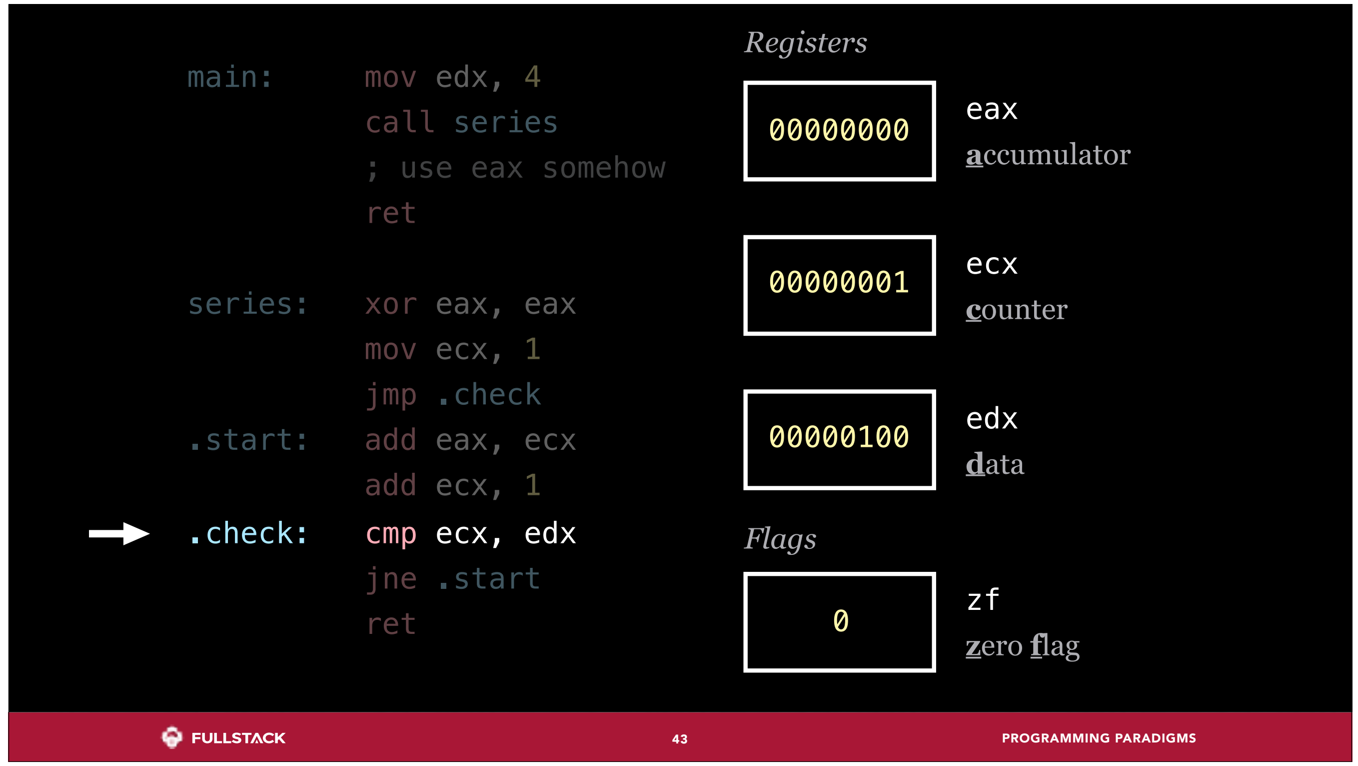
Registers

00000000	eax **a**ccumulator
00000001	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Here we compare (`cmp`) our counter and limit. If they are equal, the zero flag will be set to 1.



Not equal yet…

```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
→         jne .start
          ret
```

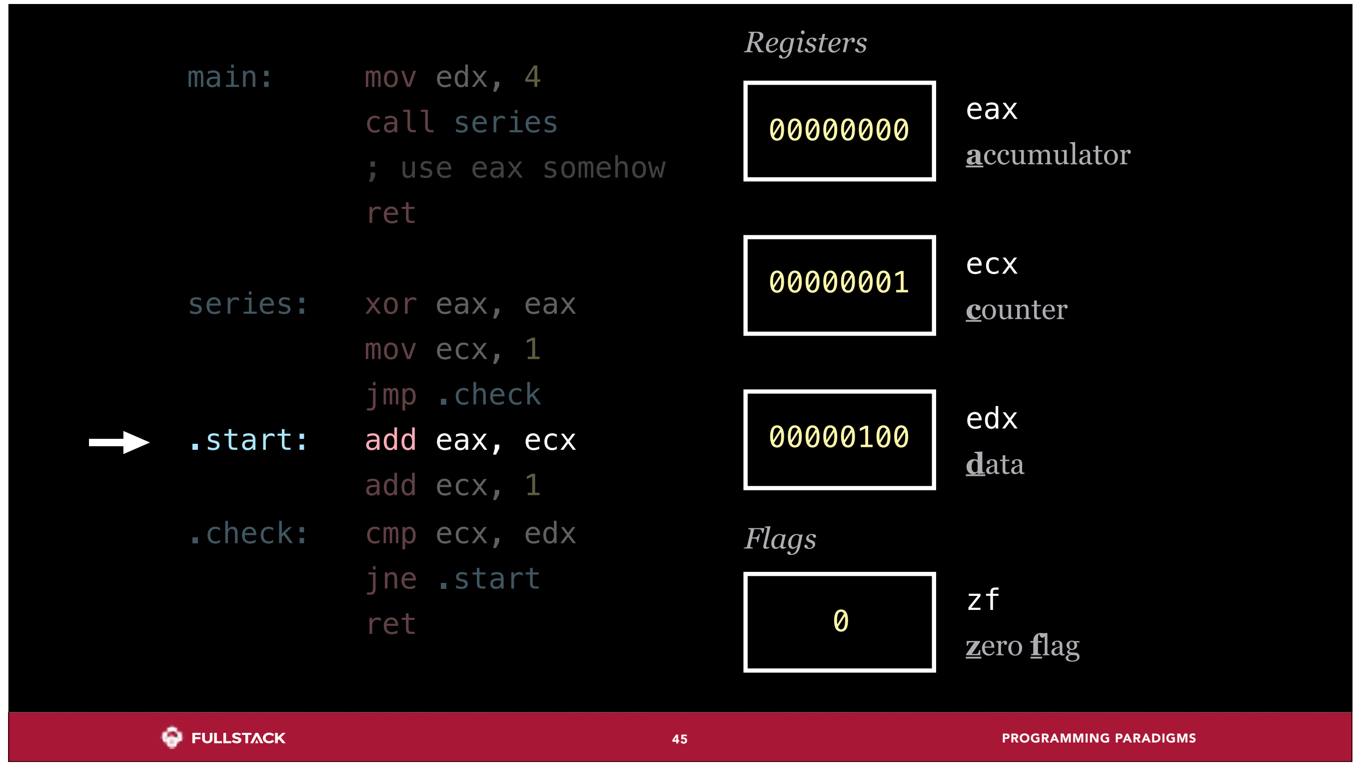
Registers

00000000	eax	**a**ccumulator
00000001	ecx	**c**ounter
00000100	edx	**d**ata

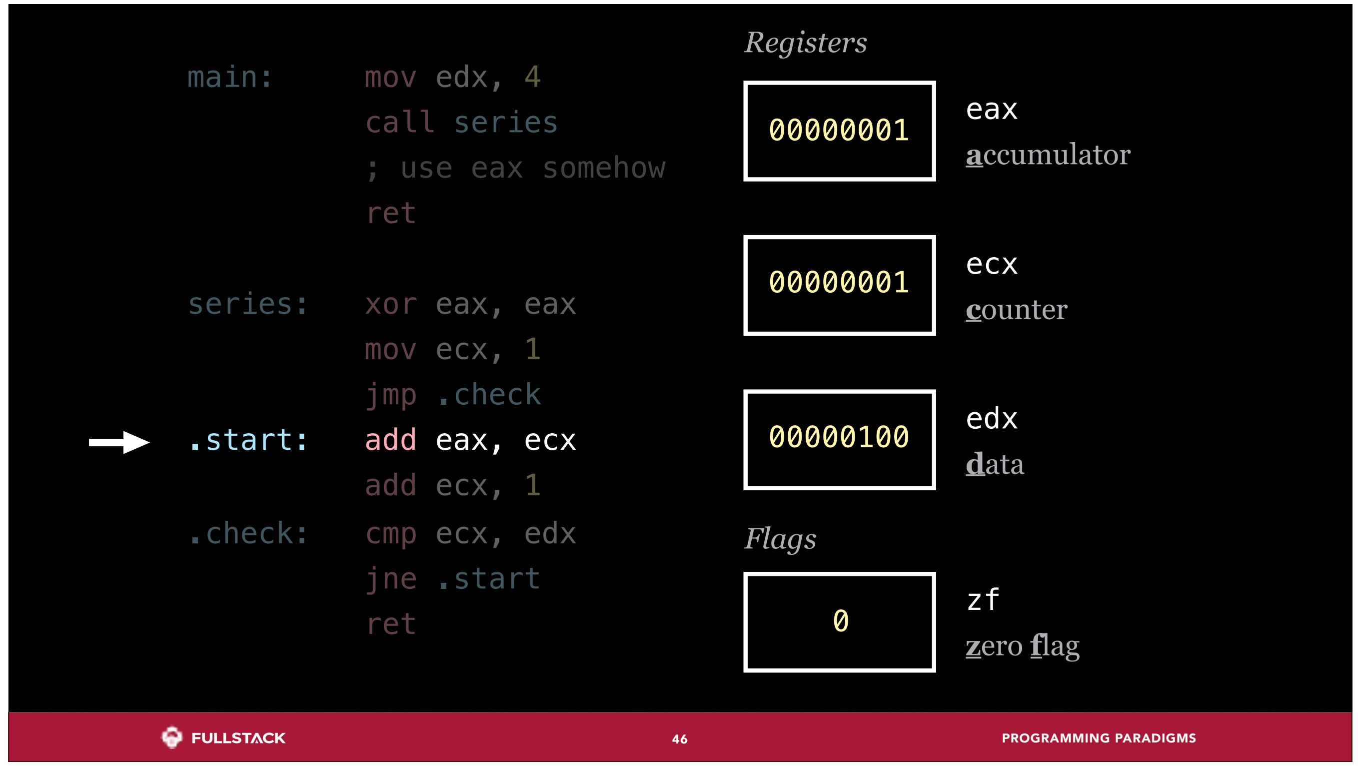
Flags

0	zf	**z**ero **f**lag

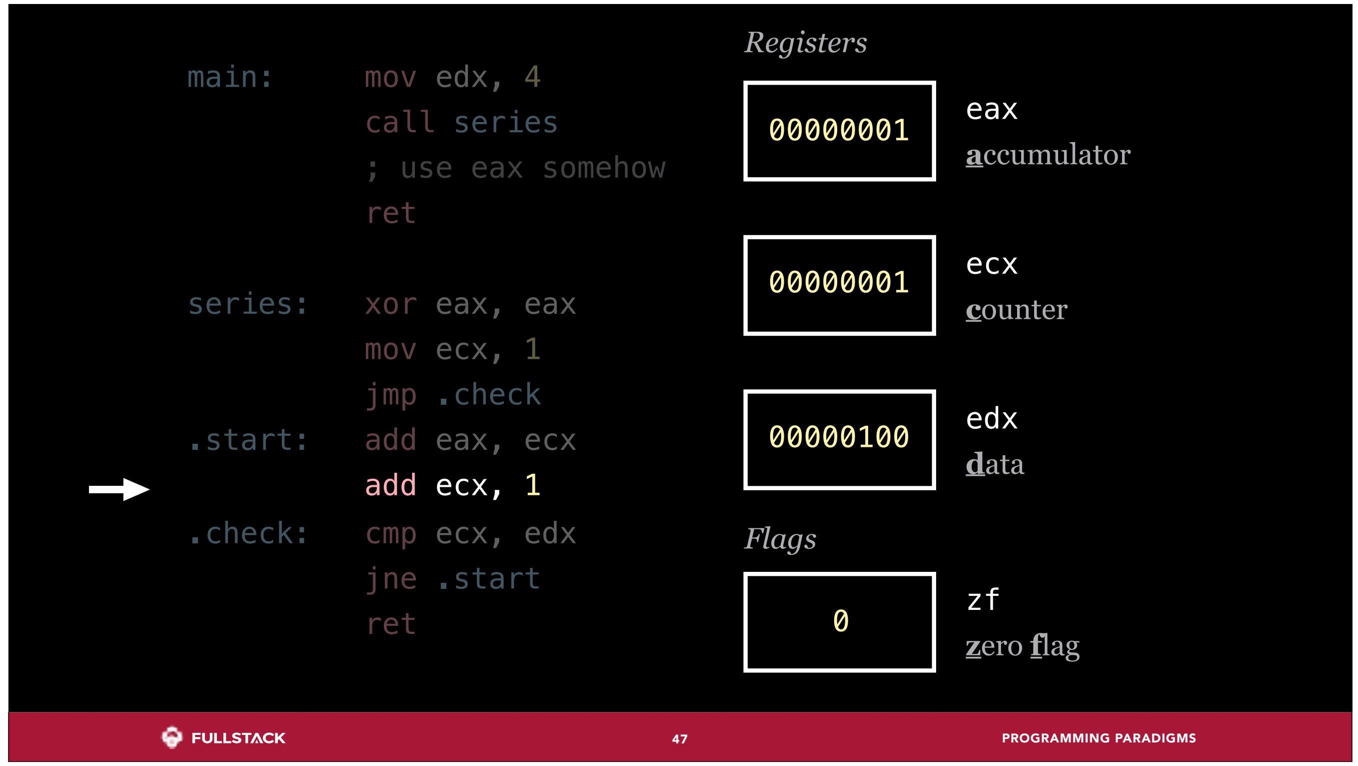
This is a conditional jump (Jump if Not Equal). It jumps if the zero flag is still off.



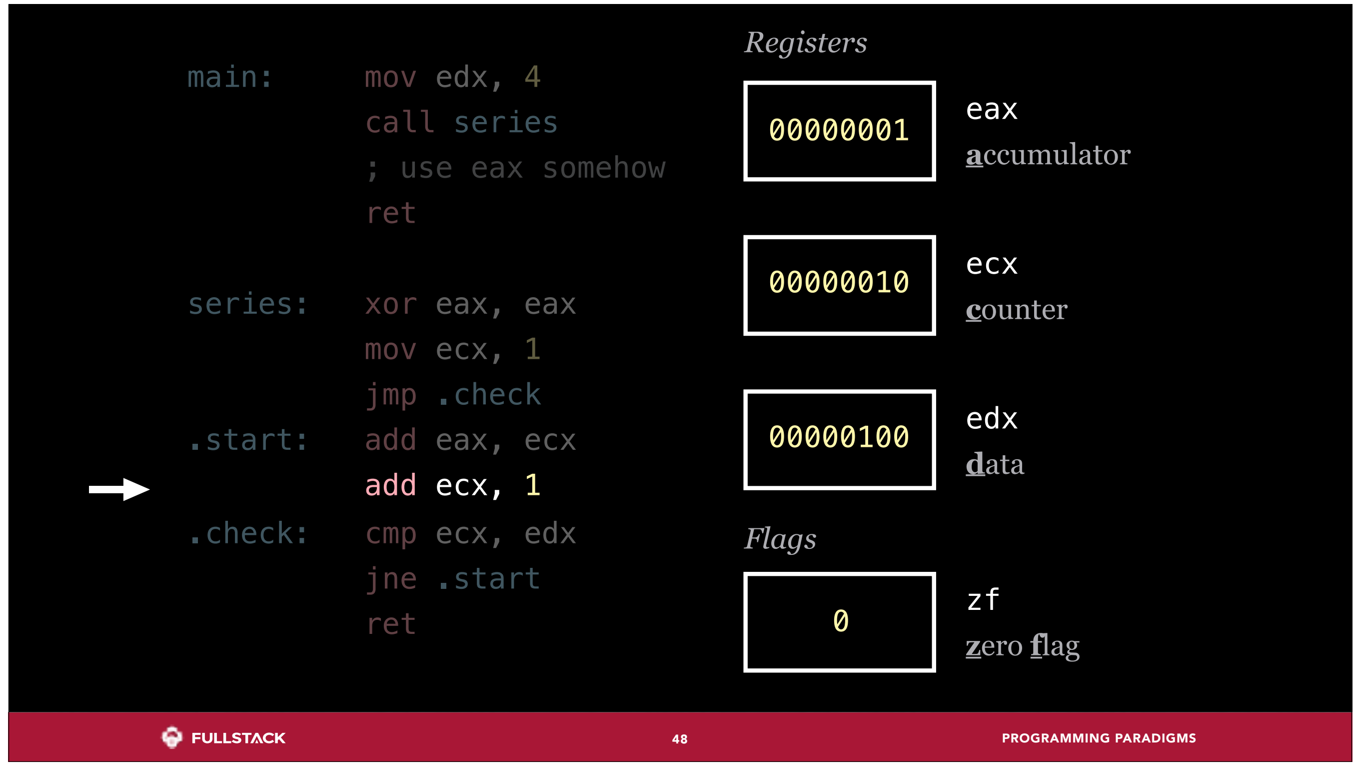
The loop is allowed to progress! We add the counter to the sum.


main: mov edx, 4
call series
; use eax somehow
ret
series: xor eax, eax
mov ecx, 1
jmp .check
.start: add eax, ecx
add ecx, 1
.check: cmp ecx, edx
jne .start
ret
Registers
00000001
eax
accumulator
00000001
ecx
counter
00000100
edx
data
Flags
0
zf
zero flag
FULLSTACK
46
PROGRAMMING PARADIGMS

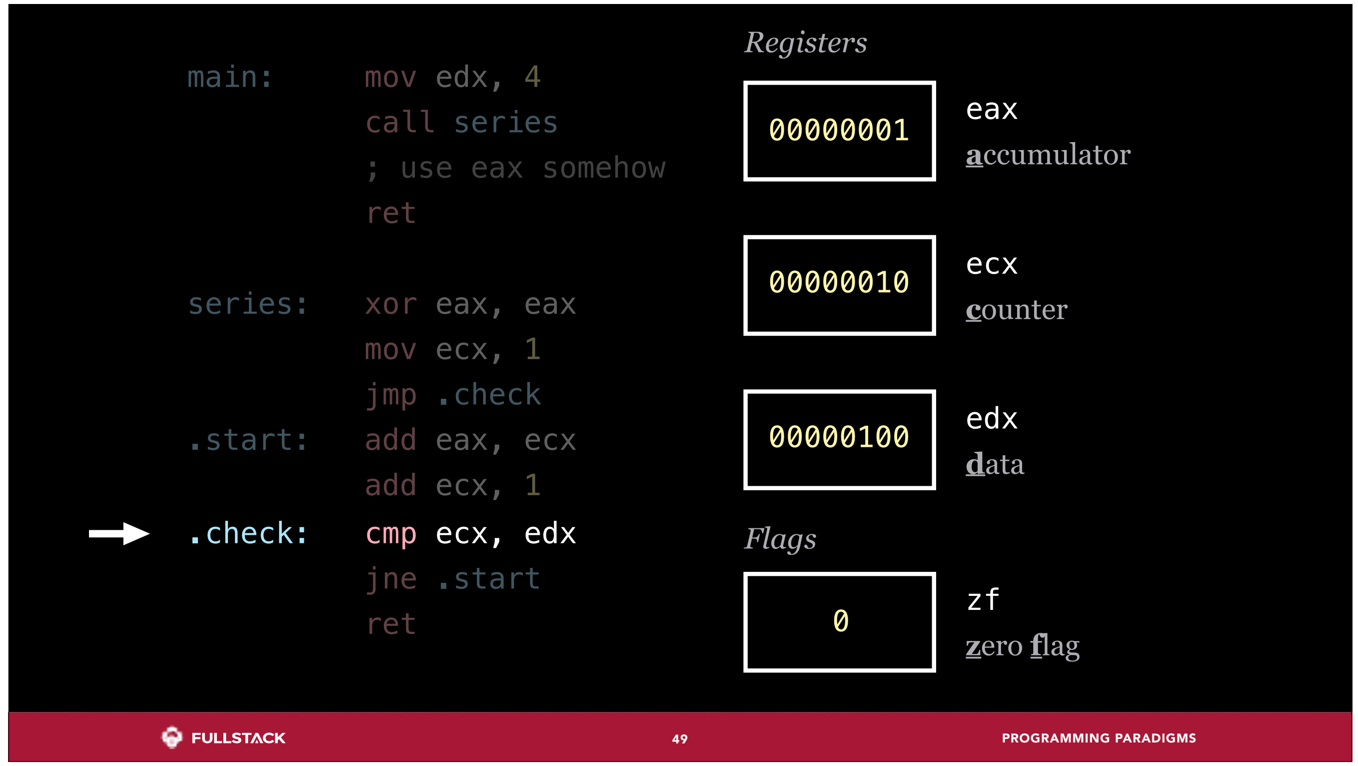
Added.



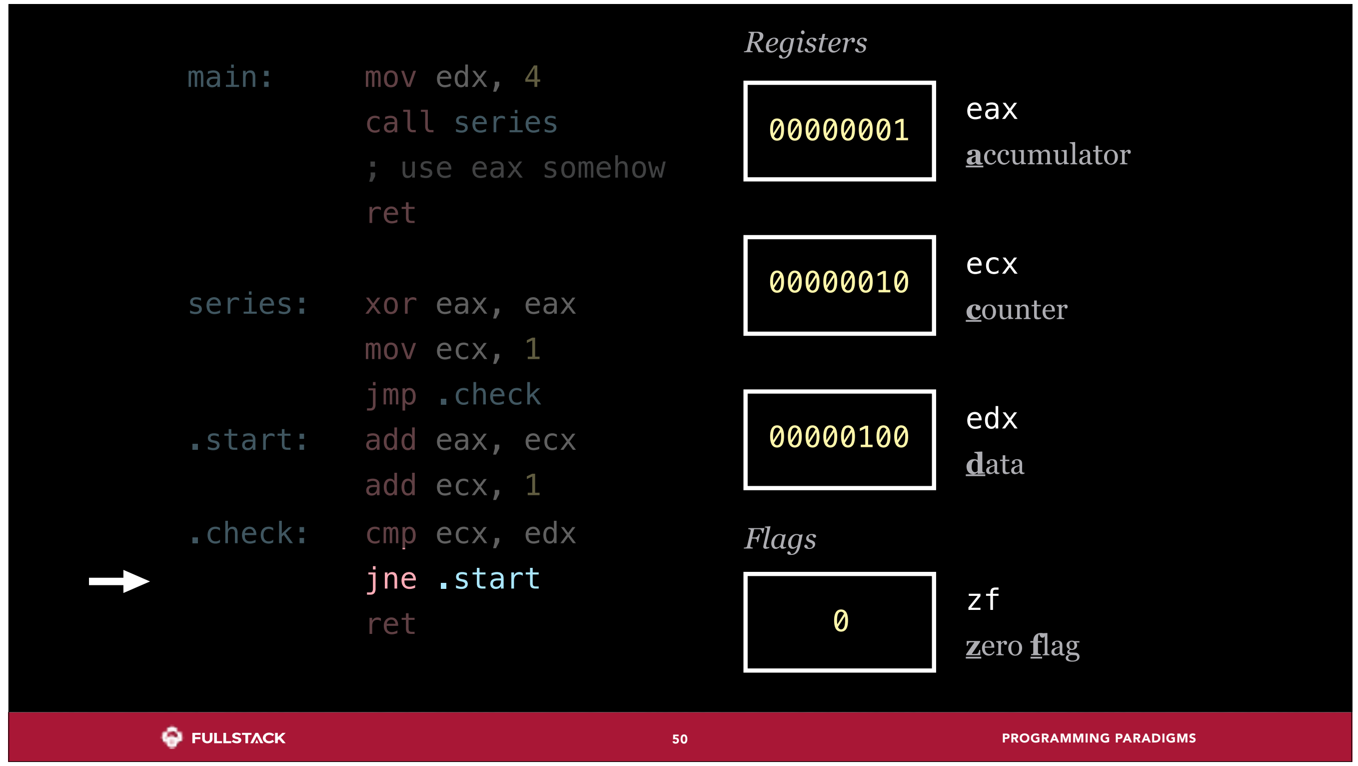
We increment the counter…


main: mov edx, 4
call series
; use eax somehow
ret
series: xor eax, eax
mov ecx, 1
jmp .check
.start: add eax, ecx
add ecx, 1
.check: cmp ecx, edx
jne .start
ret
Registers
00000001
eax
accumulator
00000010
ecx
counter
00000100
edx
data
Flags
0
zf
zero flag
FULLSTACK
48
PROGRAMMING PARADIGMS

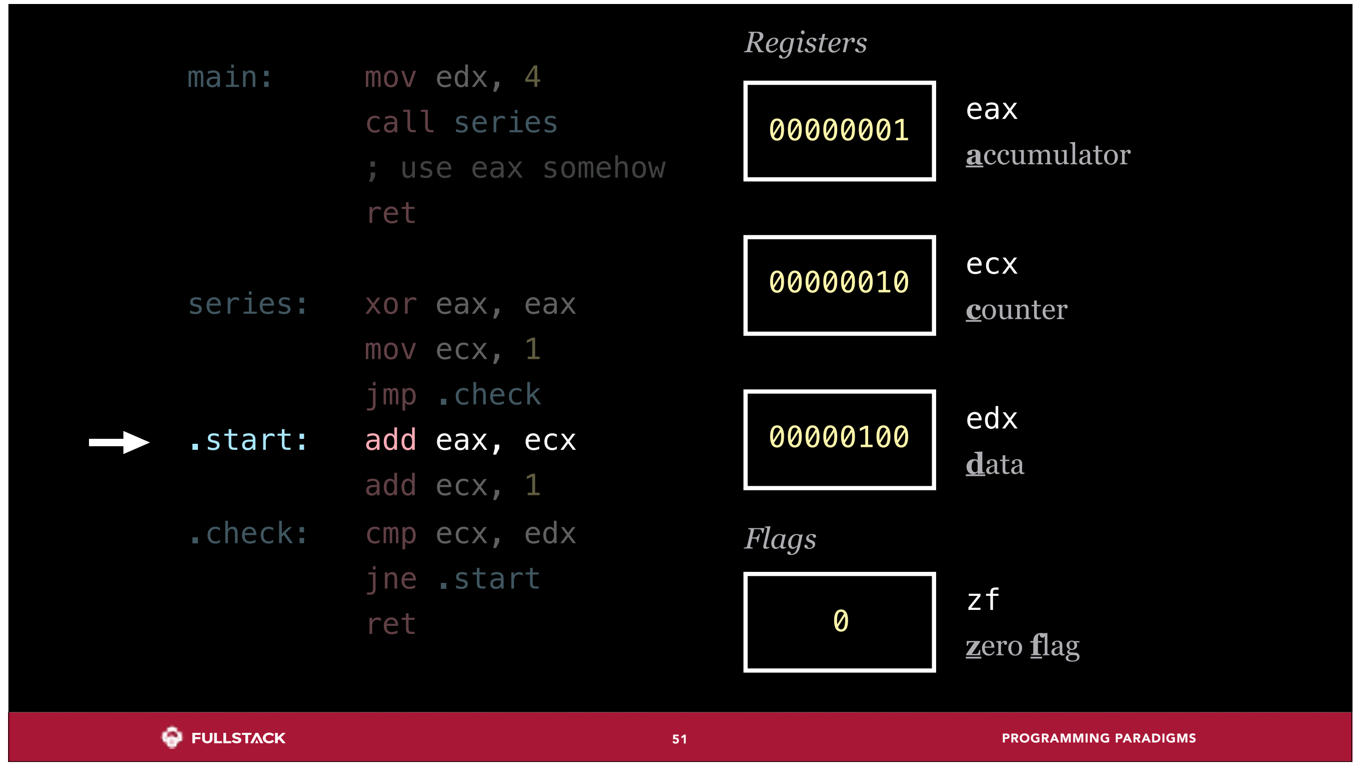
…to 2.



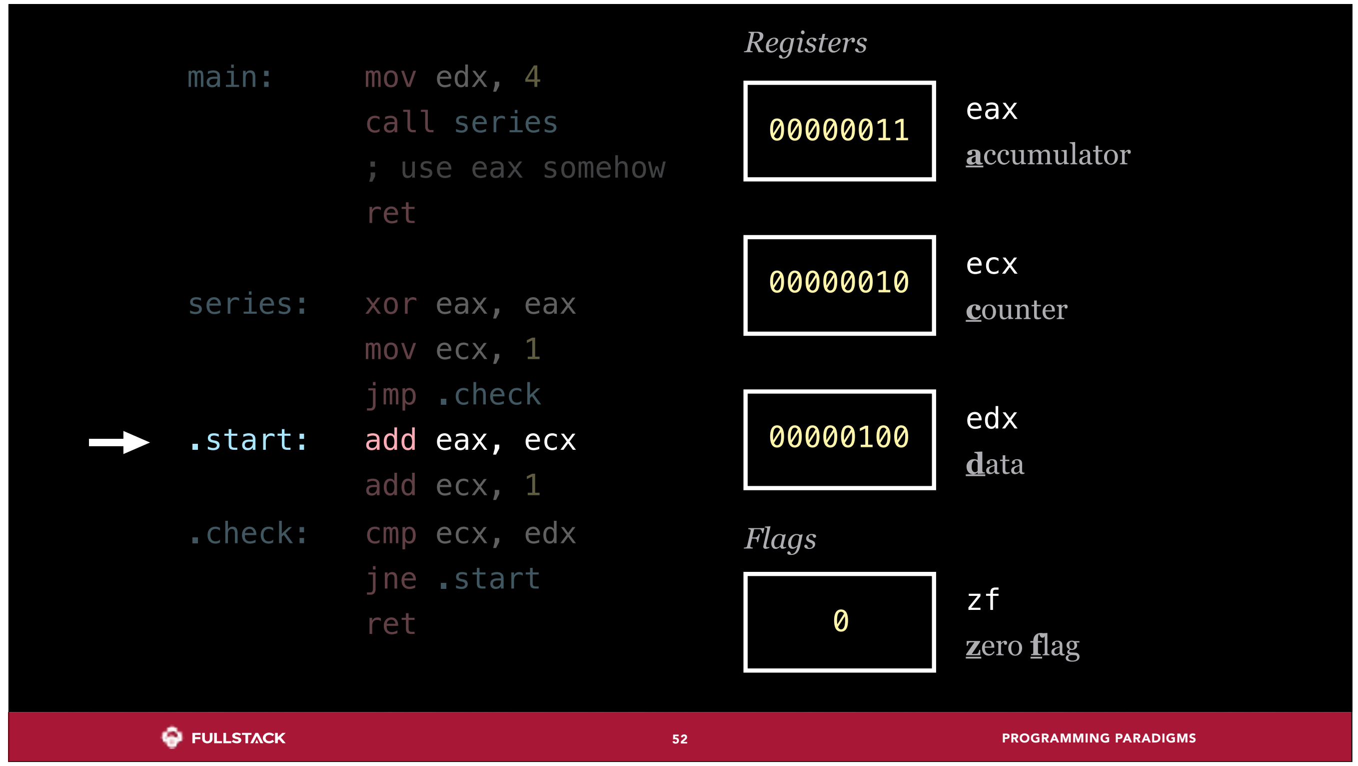
And run the loop check again. The counter and limit are still not equal, so the ZF is still 0…



…which means the jump occurs again.

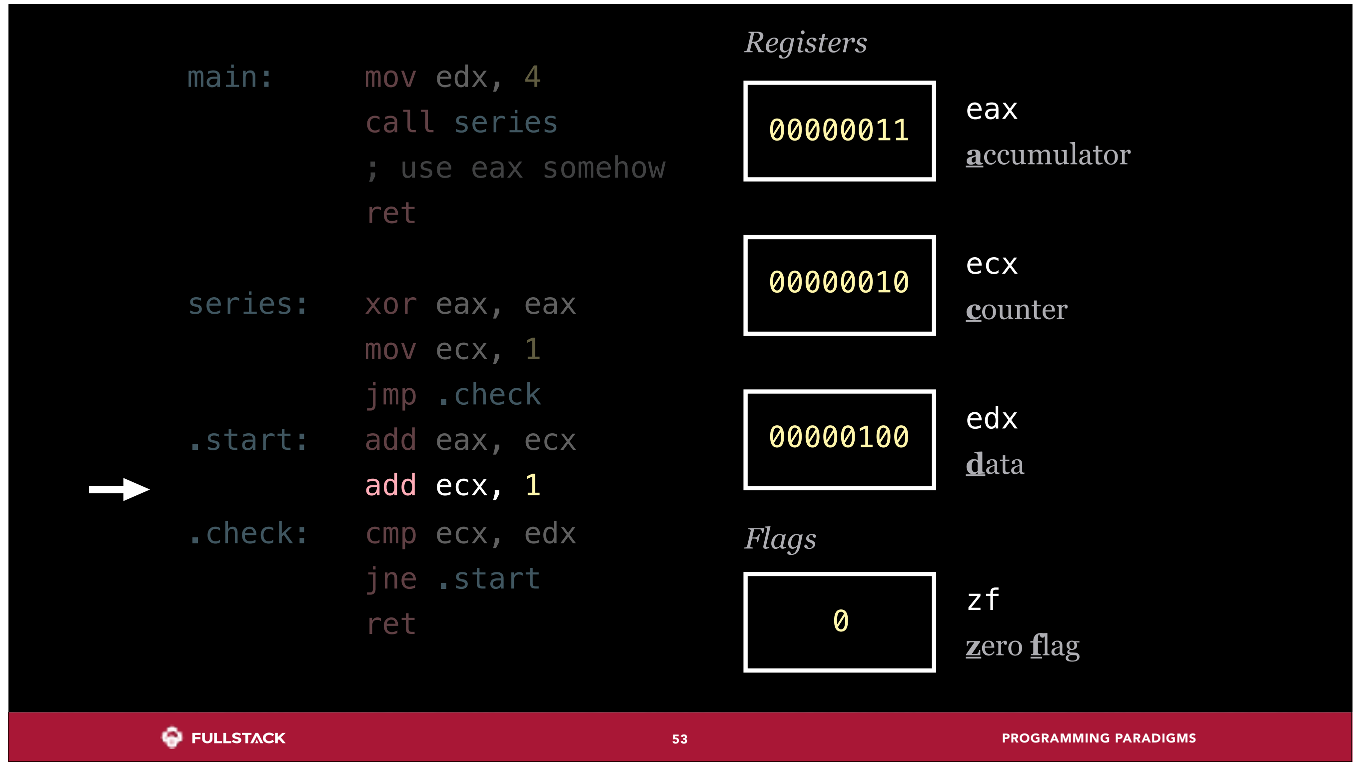


Adding i to sum…



main: mov edx, 4
call series
; use eax somehow
ret
series: xor eax, eax
mov ecx, 1
jmp .check
.start: add eax, ecx
add ecx, 1
.check: cmp ecx, edx
jne .start
ret
Registers
00000011
eax
accumulator
00000010
ecx
counter
00000100
edx
data
Flags
0
zf
zero flag
FULLSTACK
52
PROGRAMMING PARADIGMS

…sum is now 3.


main: mov edx, 4
call series
; use eax somehow
ret
series: xor eax, eax
mov ecx, 1
jmp .check
.start: add eax, ecx
add ecx, 1
.check: cmp ecx, edx
jne .start
ret
Registers
00000011
eax
accumulator
00000010
ecx
counter
00000100
edx
data
Flags
0
zf
zero flag
FULLSTACK
53
PROGRAMMING PARADIGMS

Increment i…

```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
→         add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

Registers

00000011	eax **a**ccumulator
00000011	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

…to 3. There is also an `inc` command, but `add ___, 1` is faster.

```
main:      mov edx, 4
           call series
           ; use eax somehow
           ret

series:    xor eax, eax
           mov ecx, 1
           jmp .check
.start:    add eax, ecx
           add ecx, 1
.check:    cmp ecx, edx
           jne .start
           ret
```

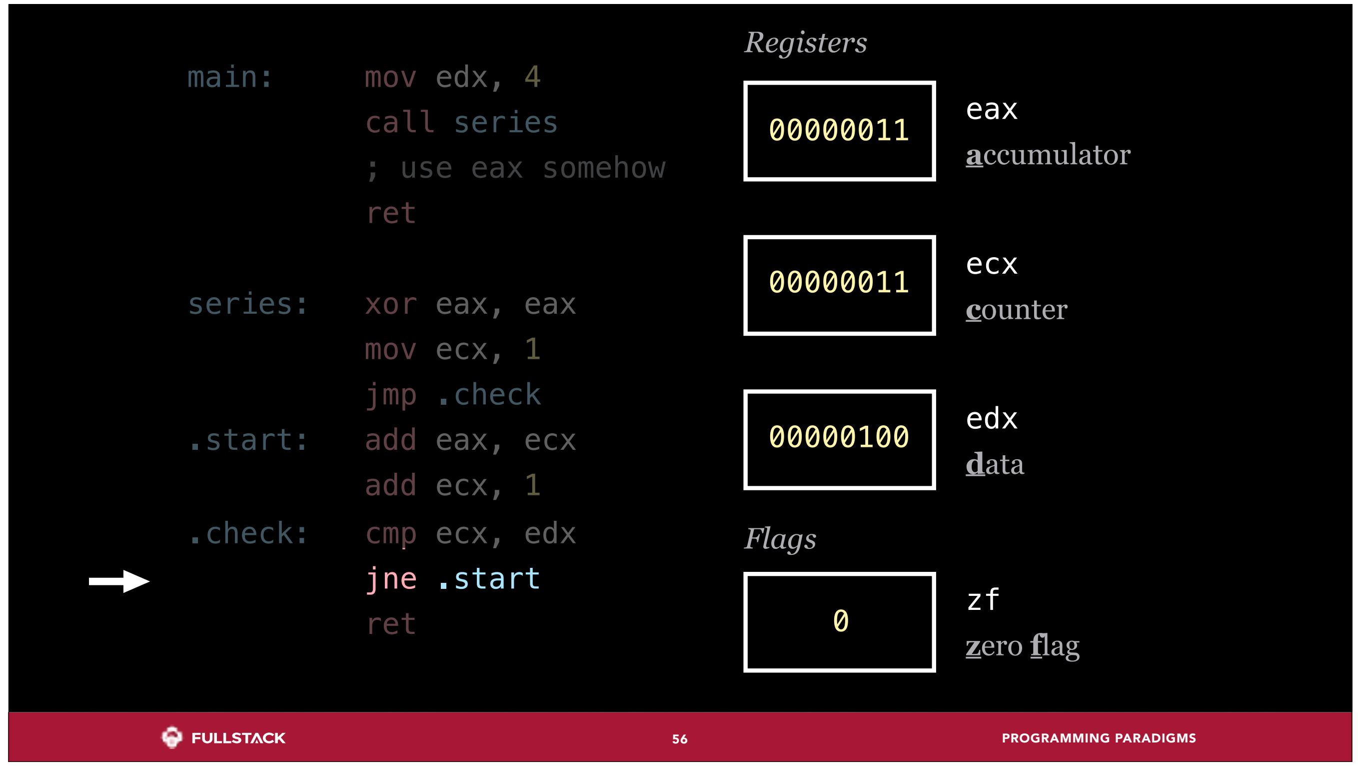
Registers

00000011	eax — **a**ccumulator
00000011	ecx — **c**ounter
00000100	edx — **d**ata

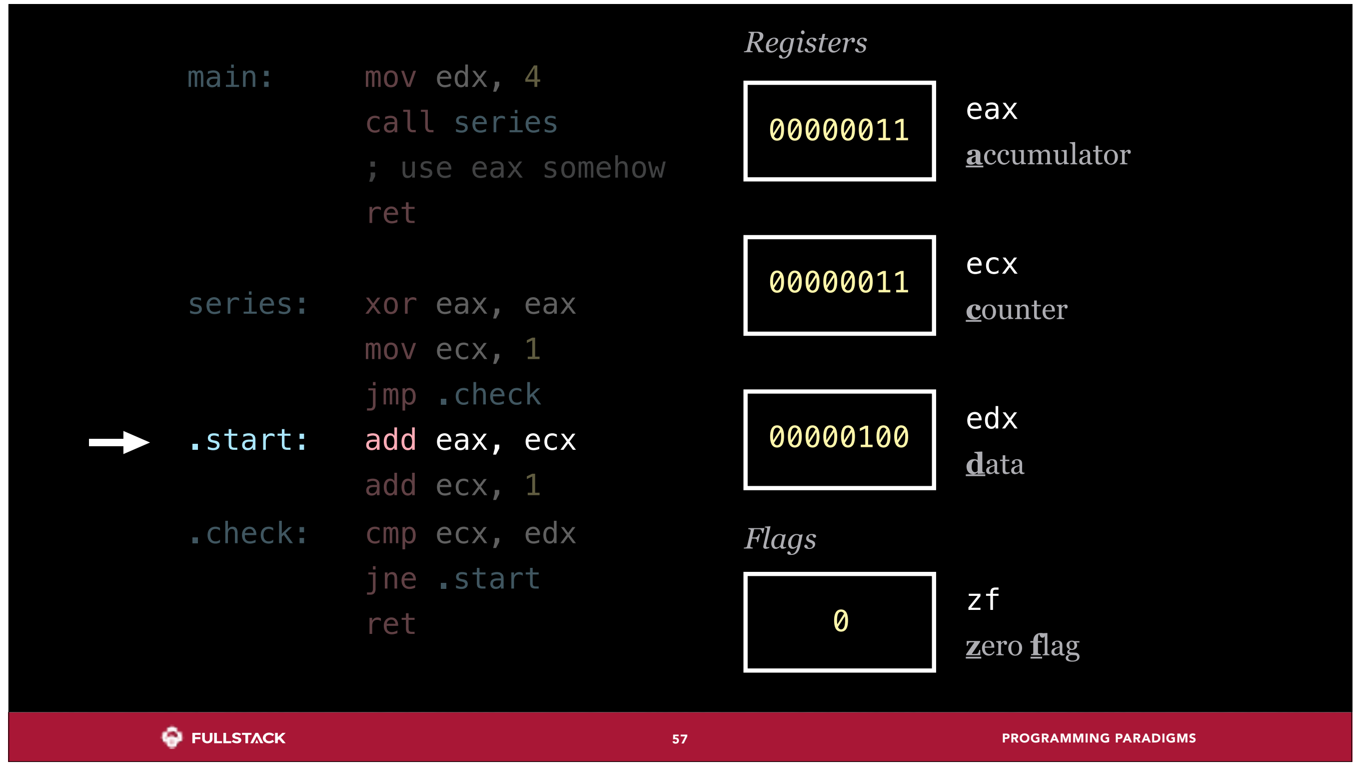
Flags

0	zf — **z**ero **f**lag

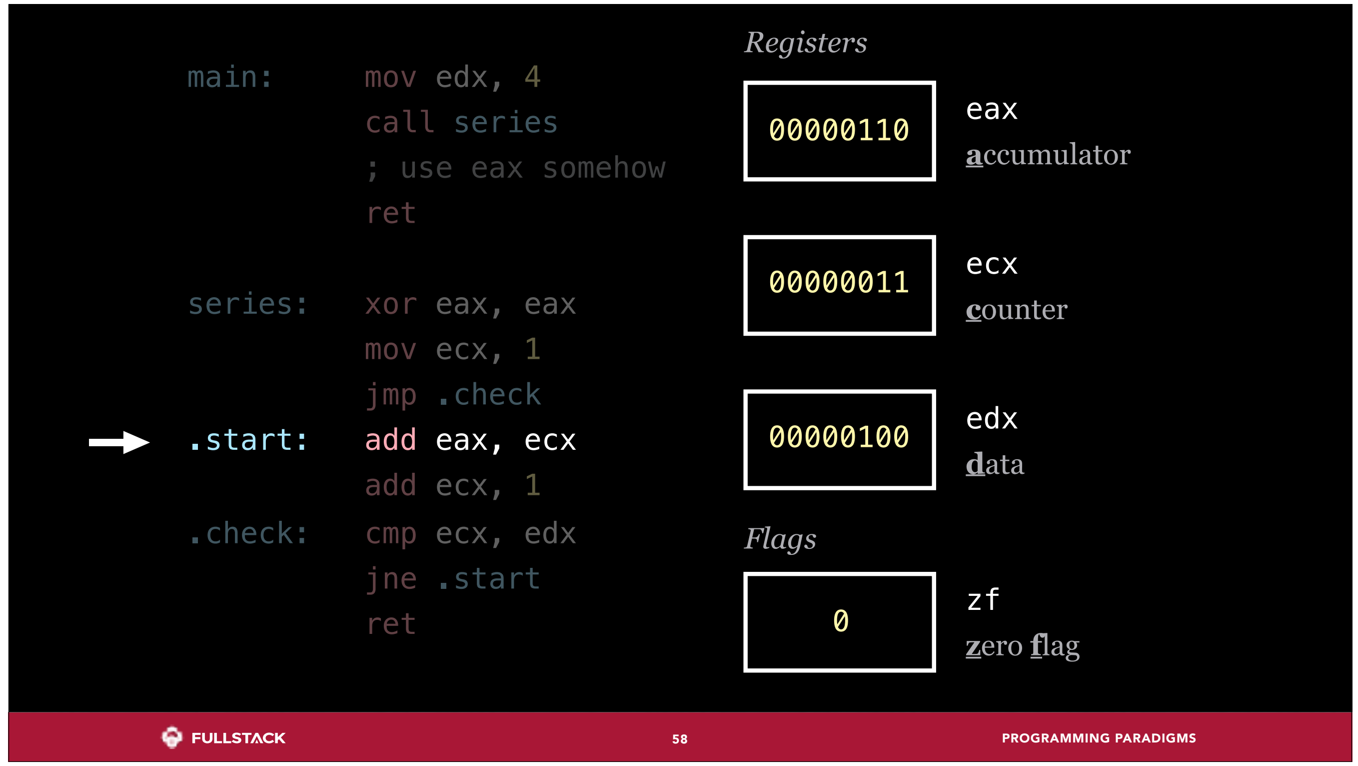
Check if i == n. Not yet.



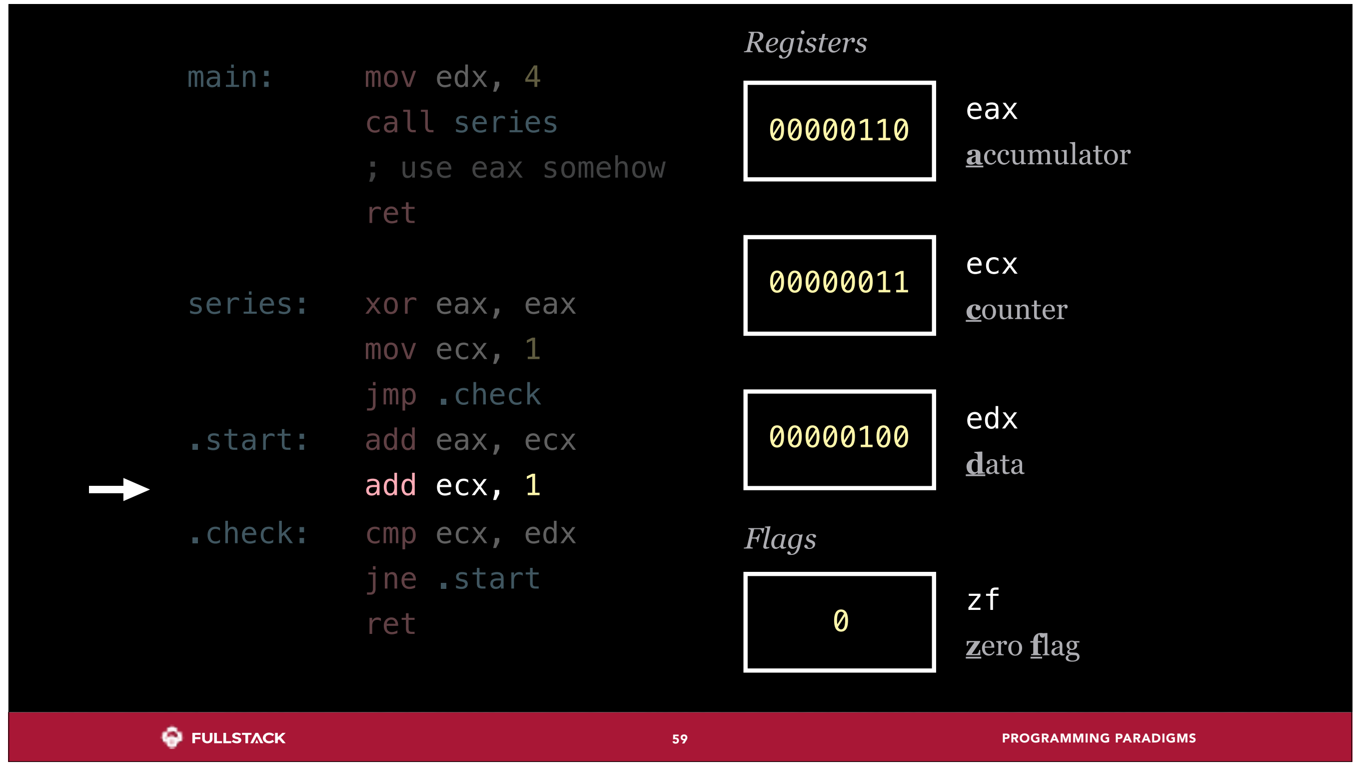
So jump again!



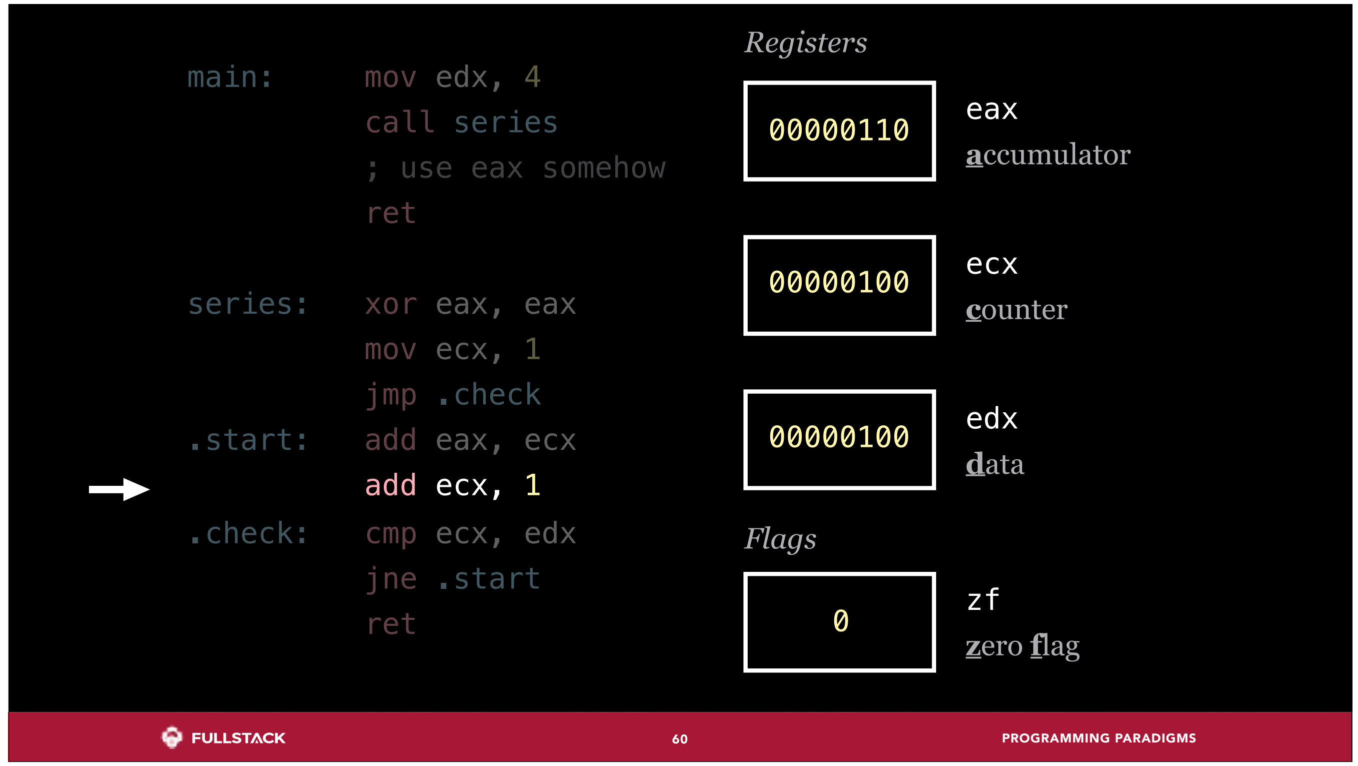
Adding i to sum,



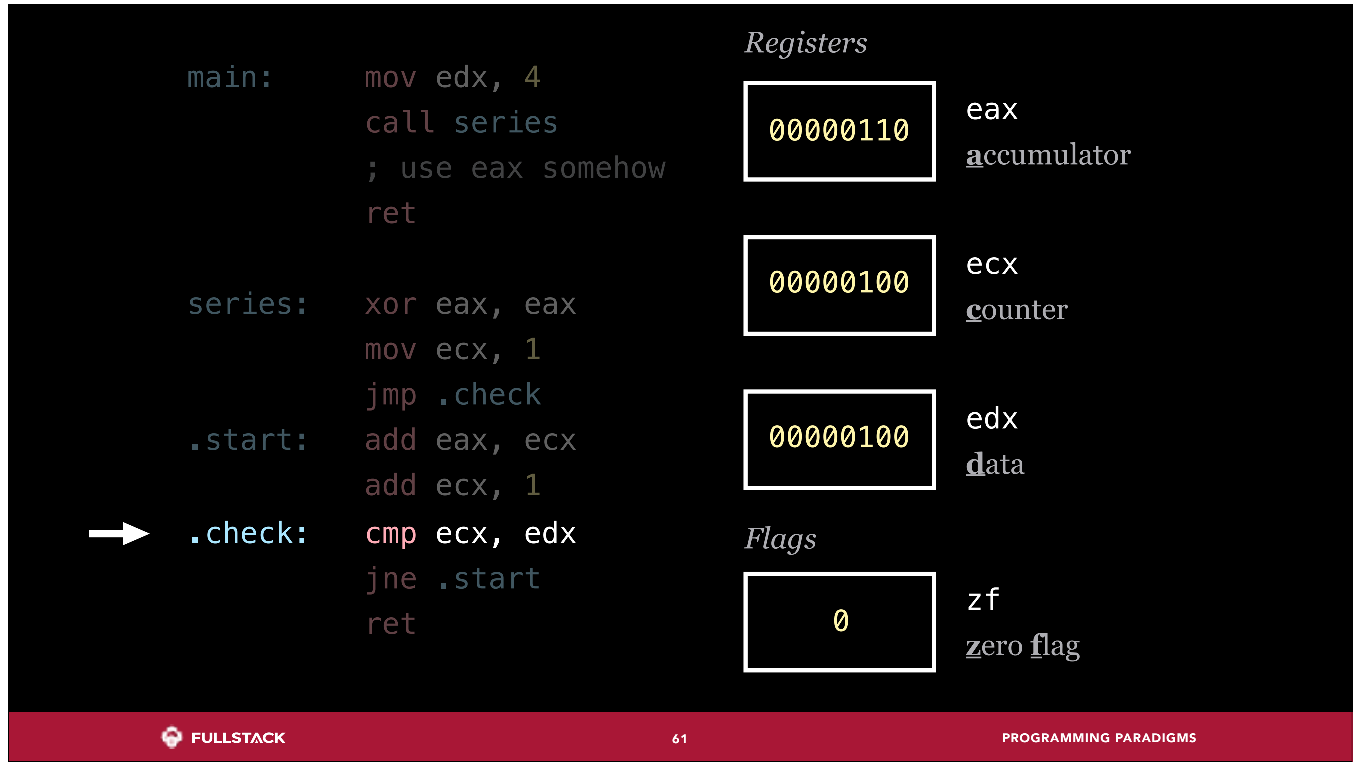
we now have 6 in sum.



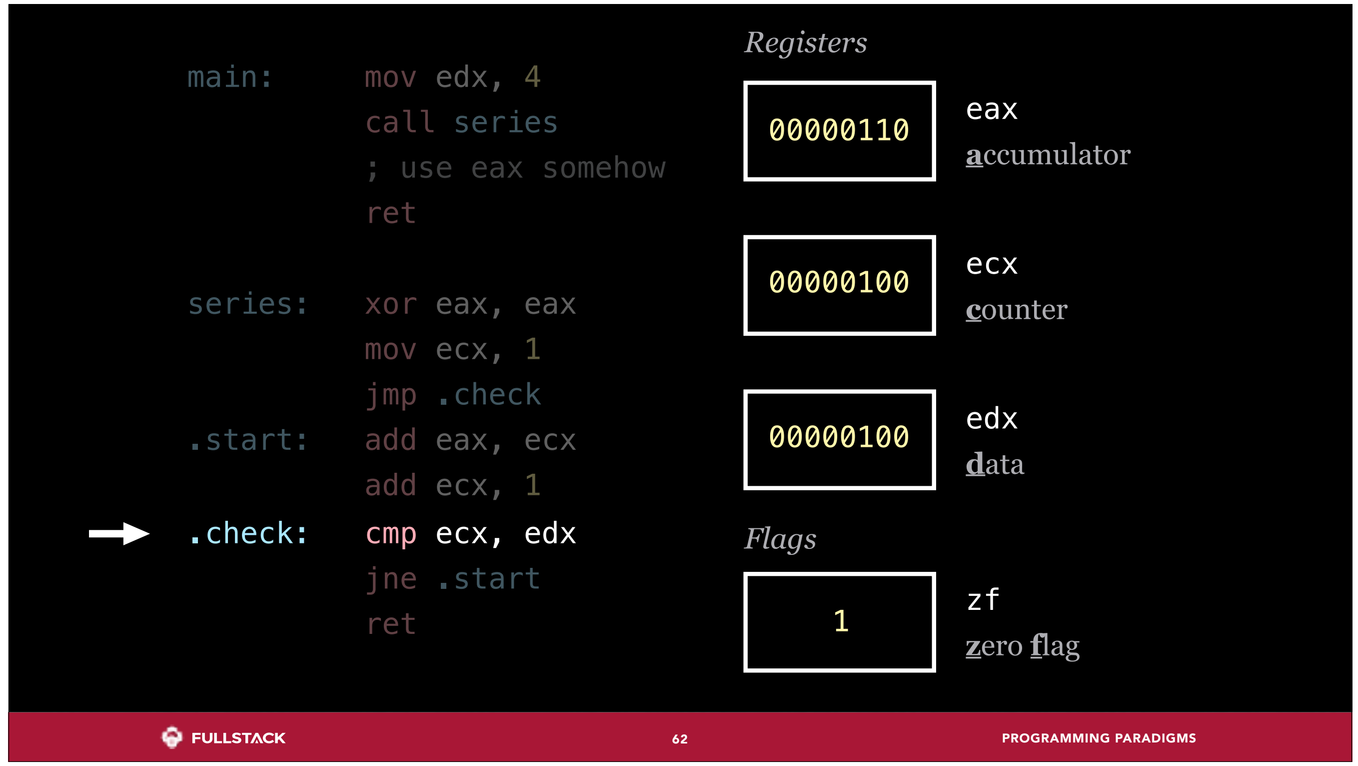
Incrementing i…



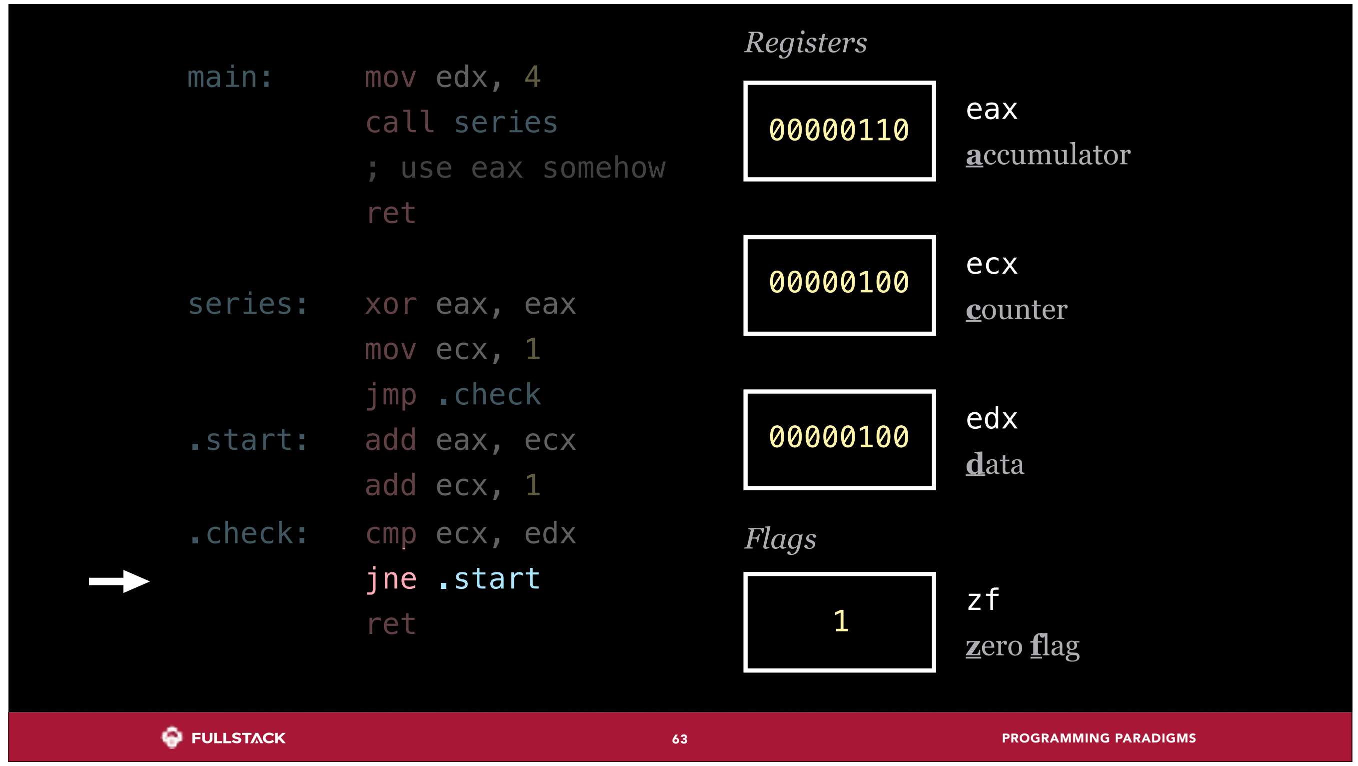
which is now 4.


main: mov edx, 4
call series
; use eax somehow
ret
series: xor eax, eax
mov ecx, 1
jmp .check
.start: add eax, ecx
add ecx, 1
.check: cmp ecx, edx
jne .start
ret
Registers
00000110
eax
accumulator
00000100
ecx
counter
00000100
edx
data
Flags
0
zf
zero flag
FULLSTACK
61
PROGRAMMING PARADIGMS

Uh oh, i == n, so…



…the zero flag is turned on.



This time the jump doesn't occur!

```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
   →      ret
```

Registers

00000110	eax **a**ccumulator
00000100	ecx **c**ounter
00000100	edx **d**ata

Flags

1	zf **z**ero **f**lag

And our series function ends.

```
main:     mov edx, 4
          call series
→         ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

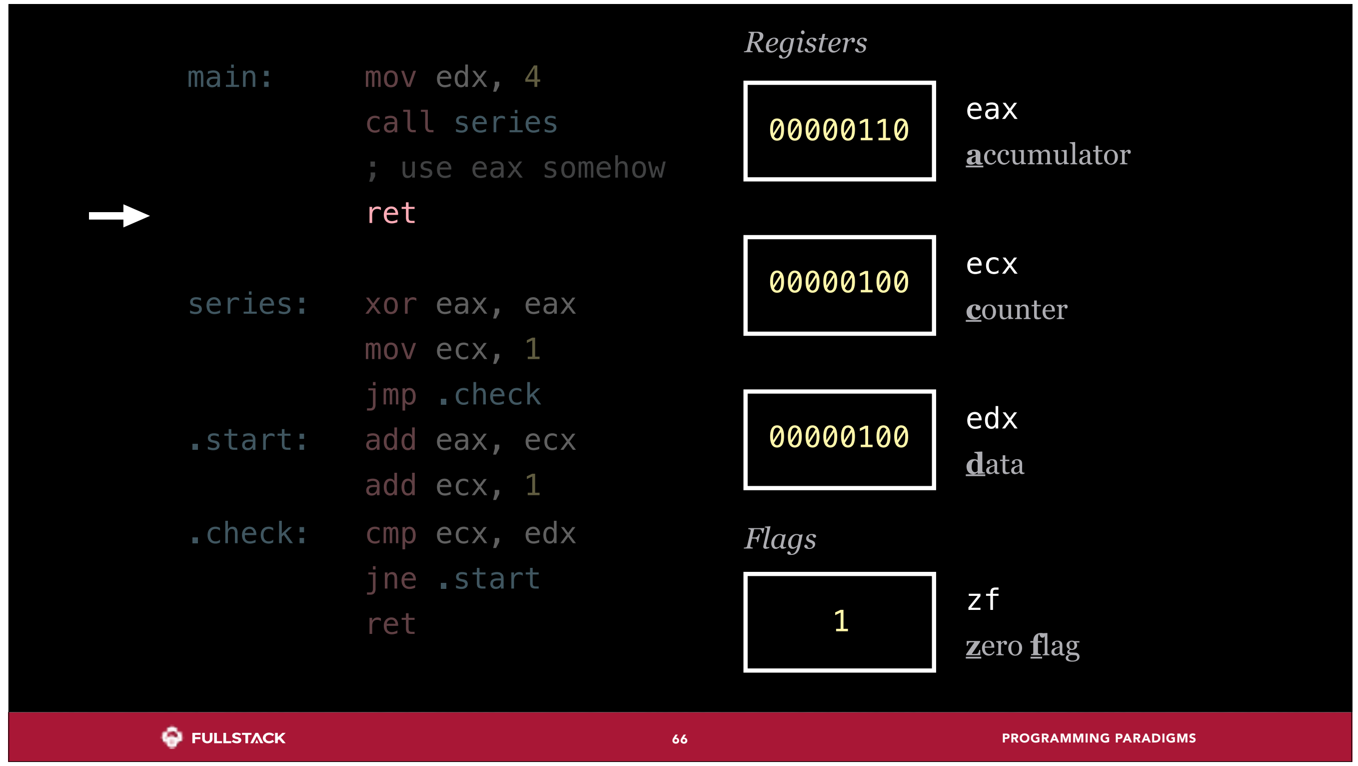
Registers

00000110	eax	**a**ccumulator
00000100	ecx	**c**ounter
00000100	edx	**d**ata

Flags

1	zf	**z**ero **f**lag

We set the accumulator register with the results of the function call, so our calling function can use it. Maybe it'll print it to STDOUT? Who knows.



Eventually, the program terminates, yielding control to the OS.

x86 Assembly

```
series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
main:     mov edx, 4
          call series
          ; use eax somehow
          ret
```

C

```c
int series(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}

int main(void) {
    int res = series(4);
    // use res (= 6) somehow
}
```

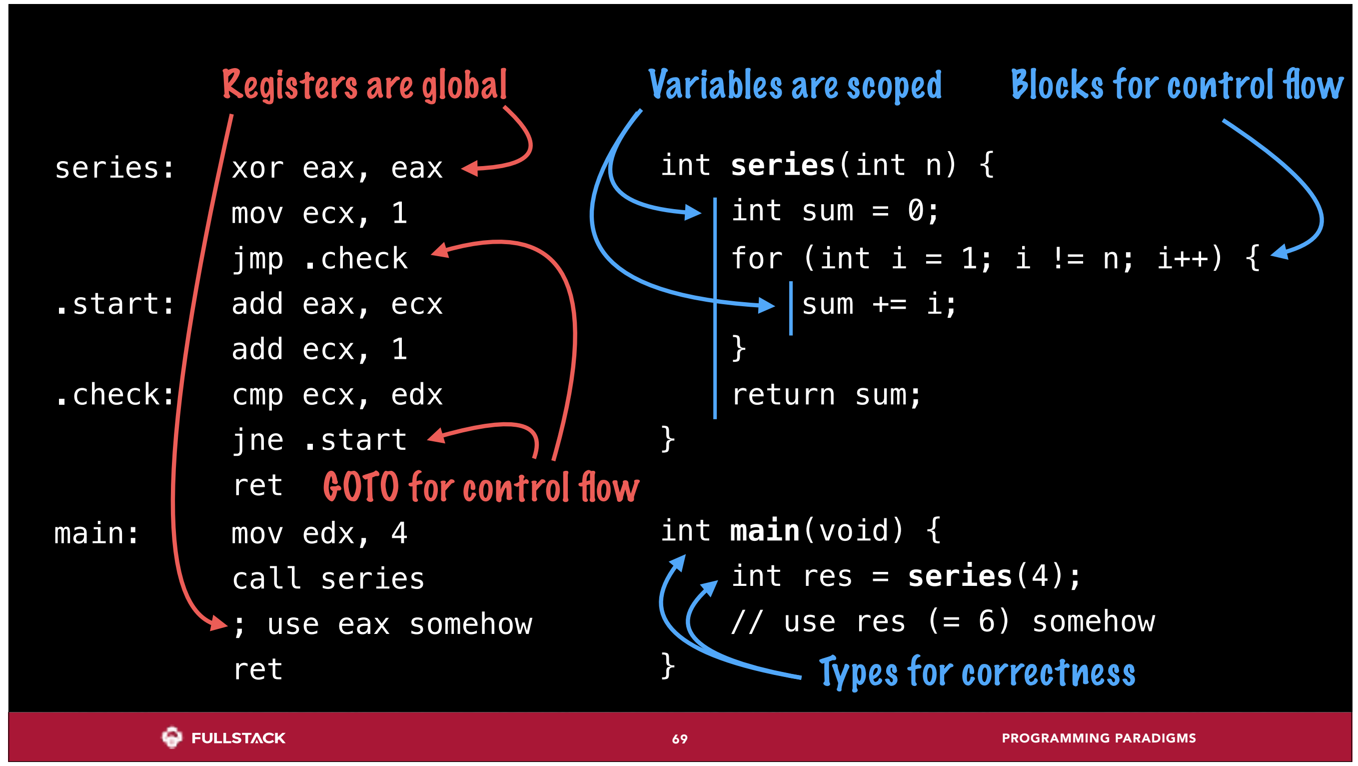
x86 Assembly

```
series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
main:     mov edx, 4
          call series
          ; use eax somehow
          ret
```

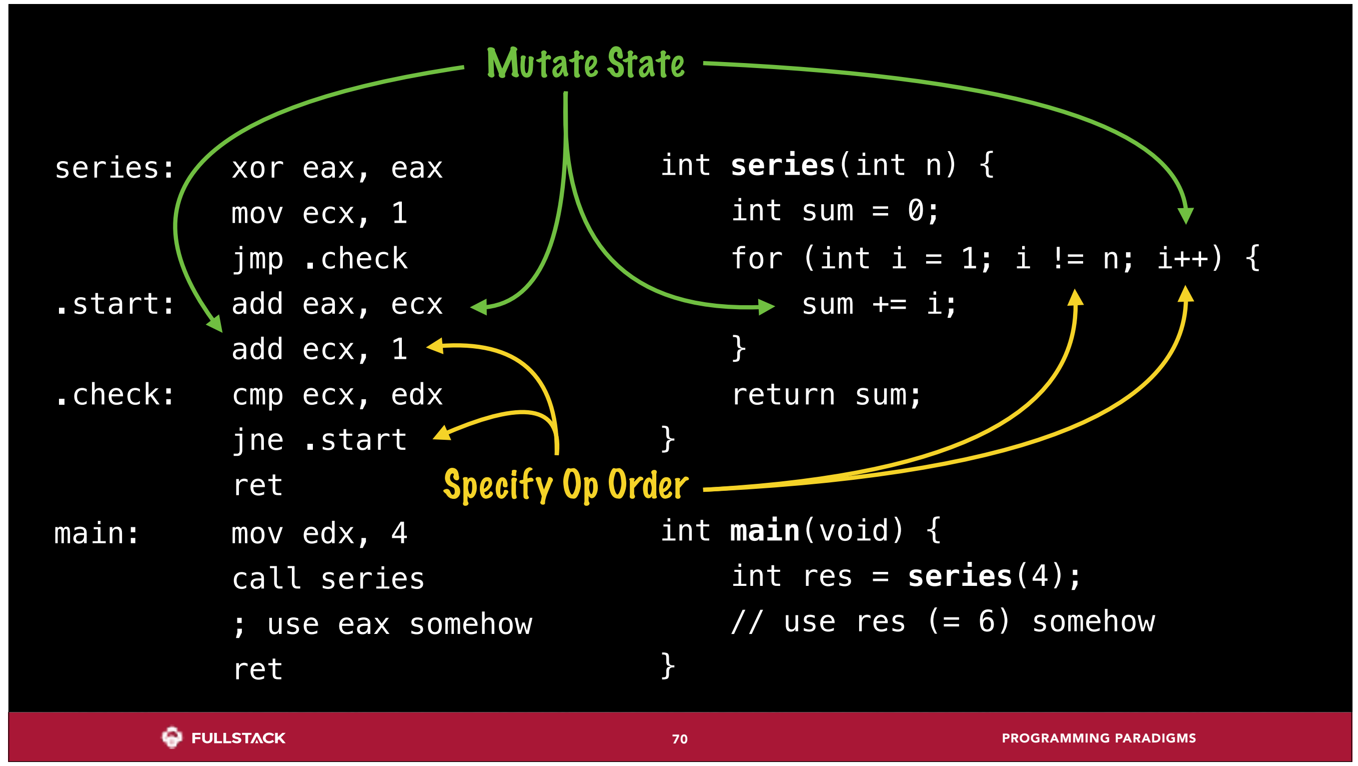
C

```c
int series(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}

int main(void) {
    int res = series(4);
    // use res (= 6) somehow
}
```


Here we color-code how each C statement can be expressed via one or more assembly statements.
* What part of the C code requires the most assembly code to express?
* What are some things C gives you that assembly evidently does not?



Some differences: in assembly, you have access to all the memory, all the time. C introduces scopes which prevent access to out-of-scope variables. C also has types to enforce correctness, and blocks for easier control flow.



Some similarities: in both Assembly and C you can (and do) mutate stateful memory in-place. You also have to do things in a particular order – changing the order of statements affects the meaning of the program since the code is stateful.

EWD249

NOTES ON STRUCTURED PROGRAMMING

by

prof.dr.Edsger W.Dijkstra

August 1969

"Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all 'higher level' programming languages (i.e. everything except —perhaps— plain machine code)."

– EDSGER W. DIJKSTRA, A CASE AGAINST THE GO TO STATEMENT (EWD 215)
(PUBLISHED UNDER "GO-TO STATEMENT CONSIDERED HARMFUL", 1968)

(some disagree: Brian Kernighan & Dennis Ritchie, Linus Torvalds, Steve McConnell)

Dijkstra is credited with very many things in CS, including coining "structured programming" and influencing its adoption. His letter on GOTO was published under the heading "go-to statement considered harmful", the origin of that popular expression in programming.

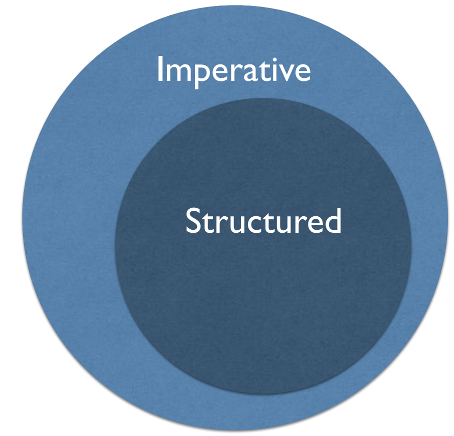
…its title, which became a cornerstone of my fame by becoming a template: we would see all sorts of articles under the title "X considered harmful" for almost any X, including one titled "Dijkstra considered harmful". But what had happened? I had submitted a paper under the title "A case against the goto statement", which, in order to speed up its publication, the editor had changed into a "letter to the Editor", and in the process he had given it a new title of his own invention! The editor was Niklaus Wirth.

Imperative

- 🤖 Instruct the machine what to do
- ▶️ Specify exact order of operations
- 🔥 Mutate state (often direct memory)

Structured

- 🗄️ Blocks for control flow
- ↔️ Branching: if/then/else, switch/case
- 🔃 Iteration: do/while/for



*These are simplifications, and many definitions vary in scope & detail

Procedures

aka Subroutines

- Jumped to / called
- Sequence of instructions
- Imperative, cause effects
- Might set / return value(s)

```
fireTheMissiles()
data = readFromGlobals()
setDefaultsOn(config)
refreshConnection()
result = sendEmail(address)
```

Functions

(when pure)

- Applied to input
- Specify logic
- Internally compute results
- Always return value

```
age = getAge(person)
friends = filter(byCool, people)
top = first(sort(letters))
revenue = fold(add, 0, sales)
returnMsg = storeMsg(secret)
```


Functions are form of procedure. But pure functions are usually contrasted with procedures on the basis of being declarative and effect-free, vs. imperative and effect-causing. Notice that a pure function **must** receive input(s) and **must** return result(s). Whereas, a procedure might not do either, since it can read from and write to the environment.

Paradigm: Declarative

…and beginning to get functional, too…

We are going to take a (very quick!) look at two other paradigms: declarative & object-oriented.

Declare What/Logic (Not How/Sequence)

Layer	Example	Omitted Implementation
HTML	Hello, World	How does this get rendered as formatted text? How is the size / position / color calculated?
HTTP	GET /api/users/1	How transmitted? Where is the data stored? What sequence of steps to retrieve it?
SQL	GET name, age FROM users LIMIT 10	Loop? Filter columns? Where is "users" in memory? How is this optimized?
RegEx	/^.+@.+$/ *(too-simple email regex)*	How does this actually work? What would the code look like that finds this match?
Pure function call	nthFibonacci(99)	What is the algorithm to compute this? Does it use recursion? Loops? Lookup table?
Math expression	− 3 + 7 ∗ 2 / (1 − 9)	Which sub-expressions need to be calculated first? What temp vars are needed?

Declarative languages break completely away from instructions. Declarations instead specify logical relationships or desired results, but none of the implementation details per se.

Every declarative layer has an imperative implementation layer behind it somewhere.

Every declarative layer has an imperative implementation layer behind it somewhere.

Declarative Layer	Imperative Implementation Layer
HTML	Browser's HTML parser → DOM representation
HTTP (e.g. `GET /api/users`)	TCP/IP in Node via C++, Express routes written in JS
SQL	RDBMS in general, see `explain query plan`
Regular Expression	RegEx engine builds a Finite State Machine (!)
pure function call, e.g. `circleArea(3)`	function body, e.g. `Math.PI * radius ** 2`
Mathematical Expression	Parser converts string to tree form, evaluates ops

Human beings are much better at expressing what we want than rigorously figuring out how to make it happen. Declarative languages support that, but only because some system somewhere is capable of digesting the declaration and transforming it into a sequence of steps to perform.

Programs as Evaluations of a Tree

- System <u>figures out</u> order of operations for you
- No direct mutation of state, only descriptions of relationships
- BUT HOW DOES IT WORK? …Someone wrote a fancy compiler, that's how. It all becomes step-by-step instructions in the end.

Example: Hypotenuse(3, 4)

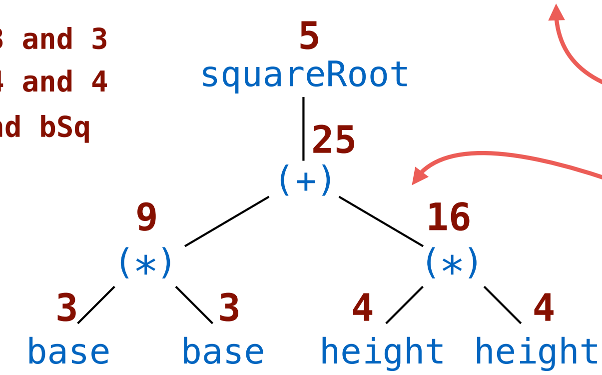
Declarative call

- hypotenuse(base, height) = squareRoot((base * base) + (height * height))

Logical Definition

```
aSq = multiply 3 and 3
bSq = multiply 4 and 4
cSq = add aSq and bSq
c = root of cSq
```

Procedural / Imperative Implementation



Parsed Logic in Tree Form

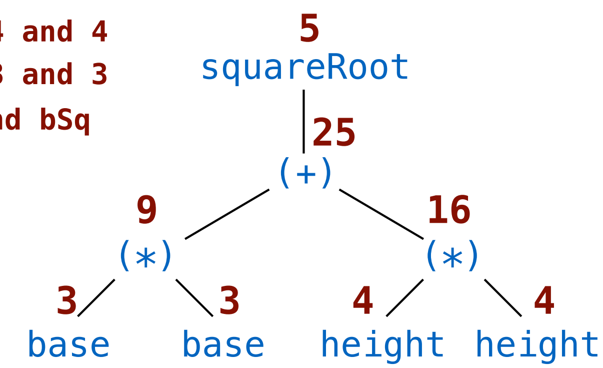
Let's see a quick concrete example.

or…

Example: Hypotenuse(3, 4)

- hypotenuse(base, height) = squareRoot((base * base) + (height * height))

```
aSq = multiply 4 and 4
bSq = multiply 3 and 3
cSq = add aSq and bSq
c = root of cSq
```



MORE DECLARATIVE / FUNCTIONAL ⟷ MORE IMPERATIVE / PROCEDURAL

function call

hypotenuse(3, 4)

function definition

$\mathrm{sqrt}(3^2 + 4^2)$

(parsed as)

```
sqrt
└→add
  ├→mult
  │ ├→3
  │ └→3
  └→mult
    ├→4
    └→4
```

procedural implementation

```
aSq = multiply 4 and 4
bSq = multiply 3 and 3
cSq = add aSq and bSq
c = root of cSq
```

(or)

```
bSq = multiply 3 and 3
aSq = multiply 4 and 4
cSq = add aSq and bSq
c = root of cSq
```

Paradigm: Object-Oriented

(super-duper condensed version)

In Short: Data + Methods = Object

- 📦 Combine state and behavior into a template for values
- 📥 Solves issues of code organization and message passing
- 👩‍👧 Object-Oriented = Objects + Inheritance & Polymorphism
- 📚 Huge field with many sub-fields & variations
- 👩‍💻 Many patterns ("Gang of Four") and best practices / pitfalls
- ⏰ Marketed as reflecting"real world" interactive entities
- 🤔 Traditionally seen as contrary to functional, but partly because OOP is associated with mutable state

We are not going to focus on OOP for this lecture as it is a giant topic which won't really be necessary to appreciate or contrast against upcoming FP topics. A bigger split is FP vs. Imperative, hence why we focused on it.

What about doing this in a (drumroll) FUNCTIONAL way?

Introduction to Functional Programming

Equational Reasoning · Pure Functions · Composition

Today we're going to begin by discussing broad categories or styles of programming languages, called paradigms.

FP in a 🌰 Nutshell

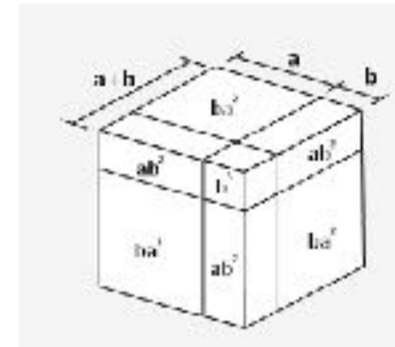
- 🎶 **Functions everywhere** (naturally)
- 🎵 **Composition of functions** (small pieces → larger constructs)
- 💘 **Pure functions only** (input → function → output, no effects)
- 💞 **Equational reasoning / referential transparency** (easier to use)
- 💟 **First-class / higher-order functions** (code uses / produces code)
- 💗 **Currying and partial application** (general-purpose → specific)
- 💎 **Immutable data** (foolproof, supports equational reasoning)
- λ **Mathematical foundations** (lambda calculus, category theory)

We will see more on these, this is to give you a taste of what's to come.
* Composition = seamlessly combining small things into bigger things
* Purity = same output for same input + no effects
* Referential transparency = function call can be replaced with value, no change in meaning
* First-class / HoF = functions are values, and functions can take and/or return functions
* Currying / partial application = give function only some args, returns a "prebaked" function waiting for more
* Immutable = cannot alter, can only generate new versions (which may share data)
* Lambda calc = basis of FP, category theory = wellspring of applicable composition patterns

FP 💡 Motivations

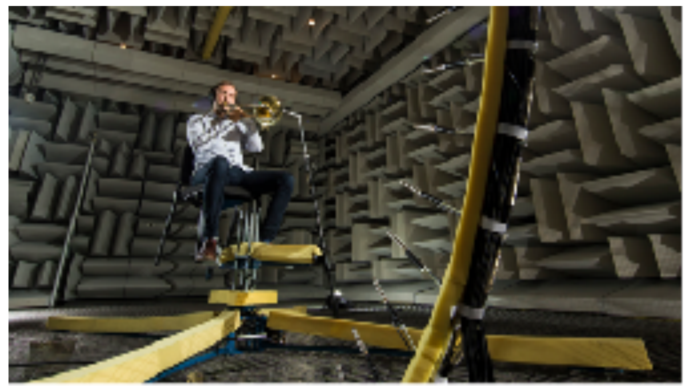
Feature(s)	Benefit(s)
Many functions, composition, higher-order, currying	Seamlessly derive new code from old, maximum interop btw. program pieces
Pure functions, immutability, no side effects, no state mutation	Equational reasoning, reduce mental scope, make bugs impossible, enable optimizations
Mathematical underpinnings (lambda calculus, category theory)	Universal concepts, provable approaches, static analysis tools, clever tech

Now, some fair warning…



$$(a + b)^3$$
$$= a^3 + 3a^2b + 3ab^2 + b^3$$

Most people learn to code imperatively. C, C++, Java, Python, Ruby, JavaScript etc. all support or even emphasize it. FP often therefore seems like a weird “alternative” different from “normal” code. This is due to <u>unfamiliarity</u>, not intrinsic truth. FP in some ways predates imperative code, & many functional languages (e.g. Lisp/Scheme, F#, OCaml, Haskell) are used in practice. With familiarity, FP makes as much (some might say more) sense than imperative code.



FP makes it easier to think about just one problem at a time, because a function only ever take in input and returns output. That makes the function usable everywhere and anywhere, and you can change its implementation at will. FP also makes it easier to assemble programs by combining small pieces together into larger, more complicated pieces.

```
function processEntriesImperative (entries) {
    const csvCopy = entries.slice()

    csvCopy.sort(function (a, b) {
        if (a['Date Created'] === b['Date Created']) return 0
        if (a['Date Created'] > b['Date Created']) return -1
        if (a['Date Created'] < b['Date Created']) return 1
    })

    const seenAlready = {}

    const finalArray = []

    for (let i = 0; i < csvCopy.length; i++) {
        if (!seenAlready[csvCopy[i]['Your Name']]) {
            seenAlready[csvCopy[i]['Your Name']] = true
            finalArray.push(csvCopy[i])
        }
    }

    return finalArray
}
```


Is this code imperative or declarative?

Are there any bugs?

What does this code actually do? Does it sort ascending or descending? Whats' the purpose of `seenAlready`?

```
const R = require('ramda')

const processEntriesFunctional = R.pipe(
    R.sort(R.descend(R.prop('Date Created'))),
    R.uniqBy(R.prop('Your Name'))
)
```


Is this code imperative or declarative?
Are there any bugs?
What does this code actually do?

Add numbers from 1 to n (exclusive?)

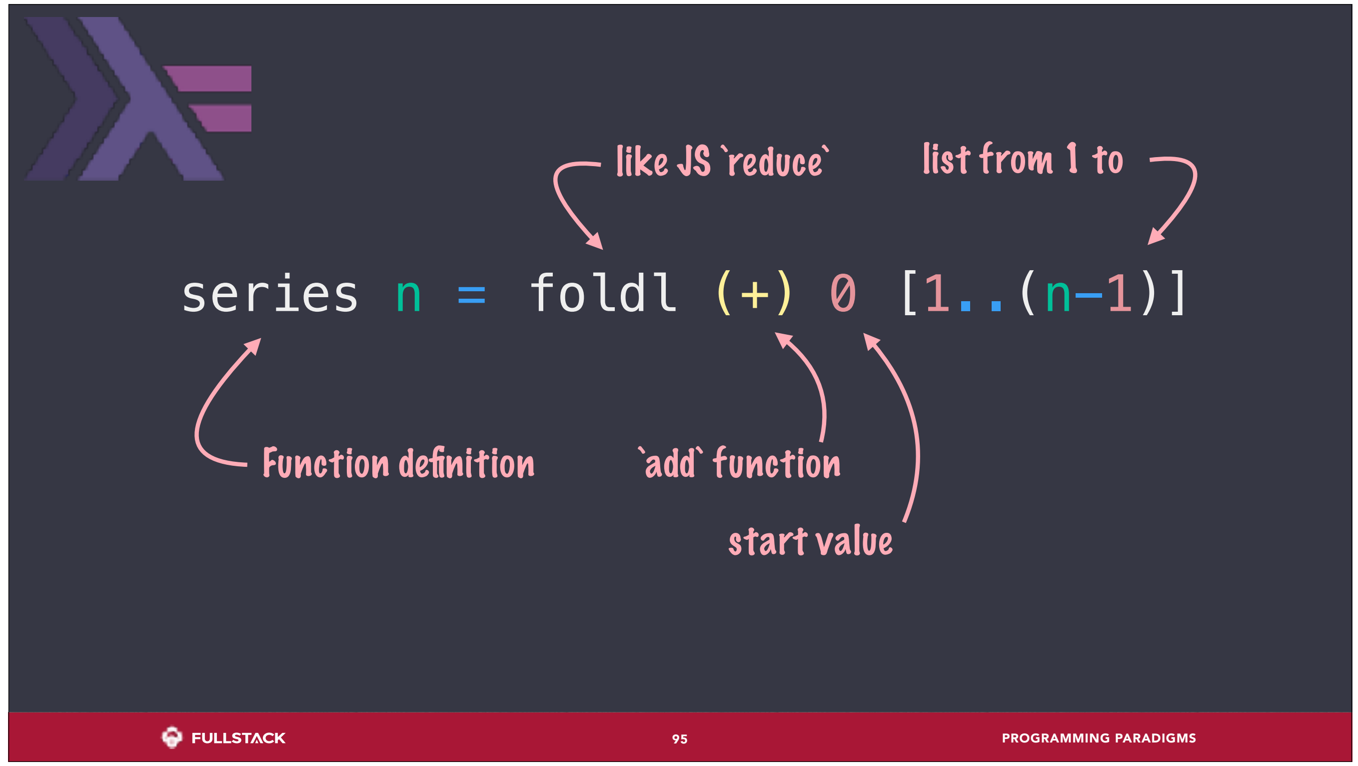
reminder…

O(n): sum = 1 + 2 + 3 + … + (n - 1)

O(1): sum = n * (n - 1) / 2

(we'll do the naive way, for the sake of demonstration)

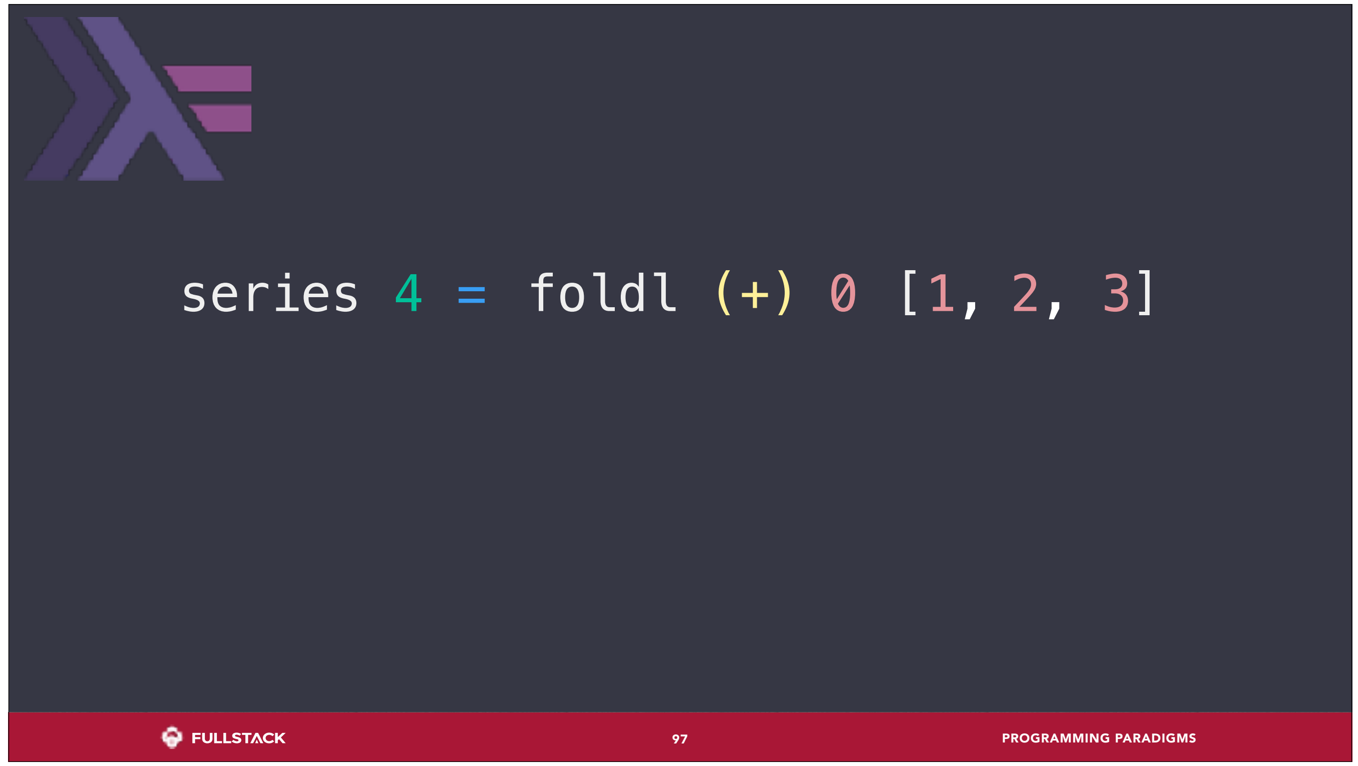
Just as a reminder, we tackled this problem earlier.



Here is an example of the "series" function written in Haskell (a lazily evaluated pure functional language with type inference).

```
series 4 = foldl (+) 0 [1..(4-1)]
```


When we substitute `4` in as our argument, (next page)

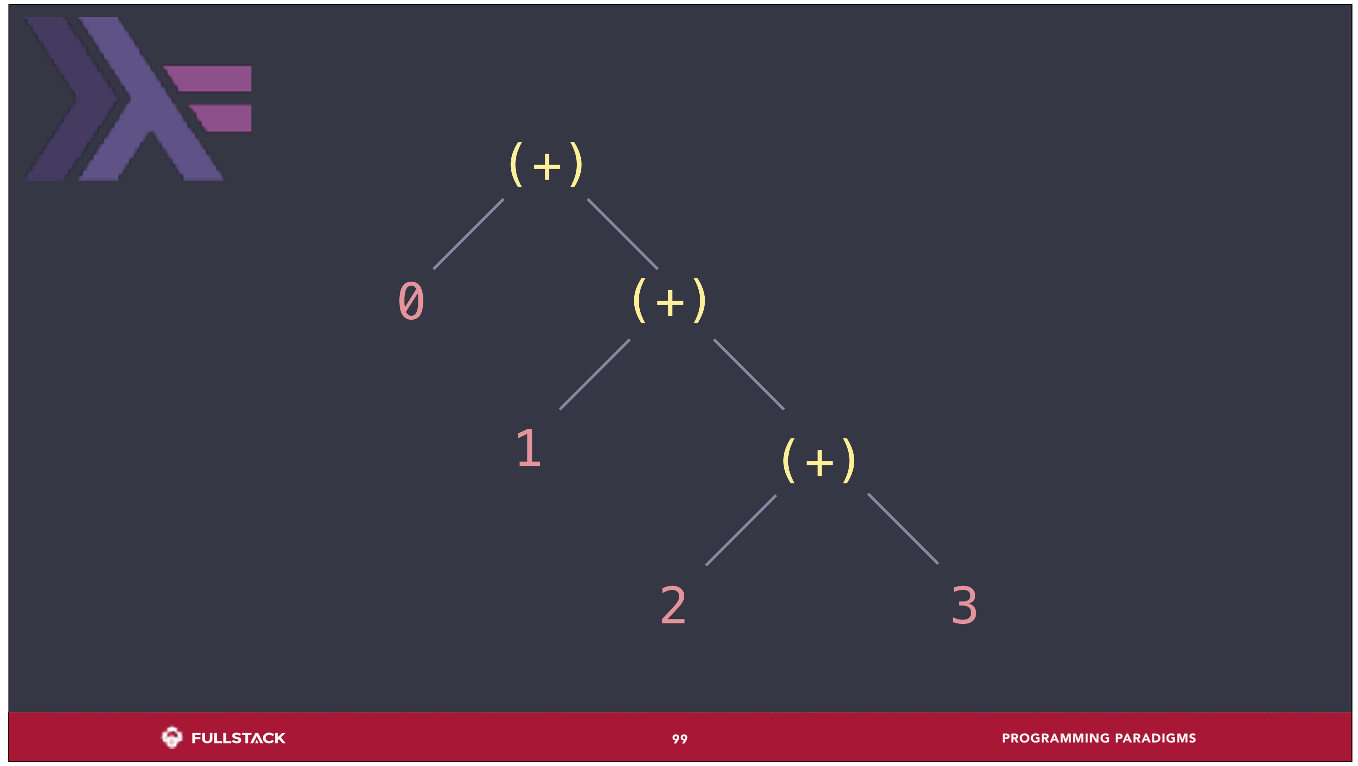


…our list expands…

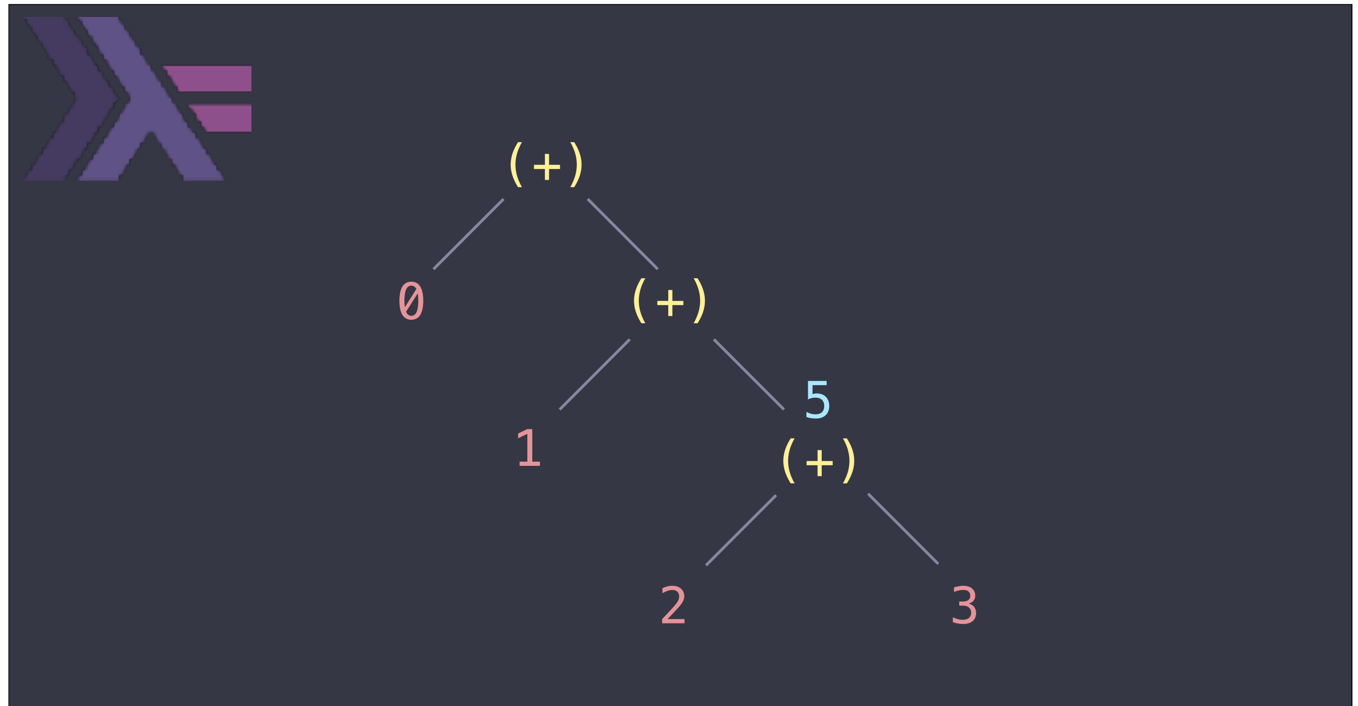
```
foldl (+) 0 [1, 2, 3]
```

FULLSTACK

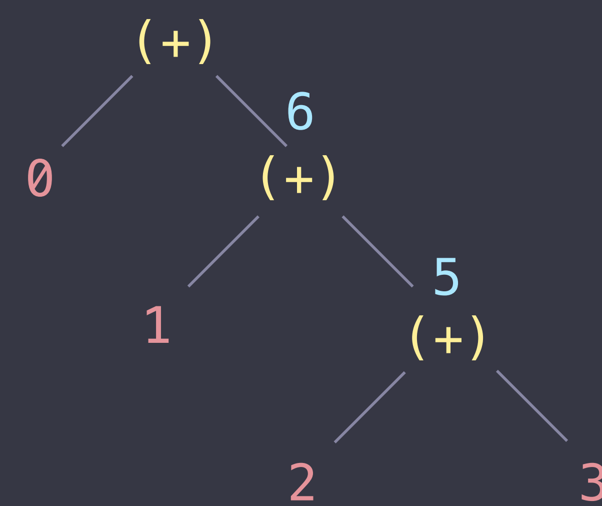
…and our function application looks like this. `foldl` is a left-associative list fold – that is, very similar to JS `reduce`.

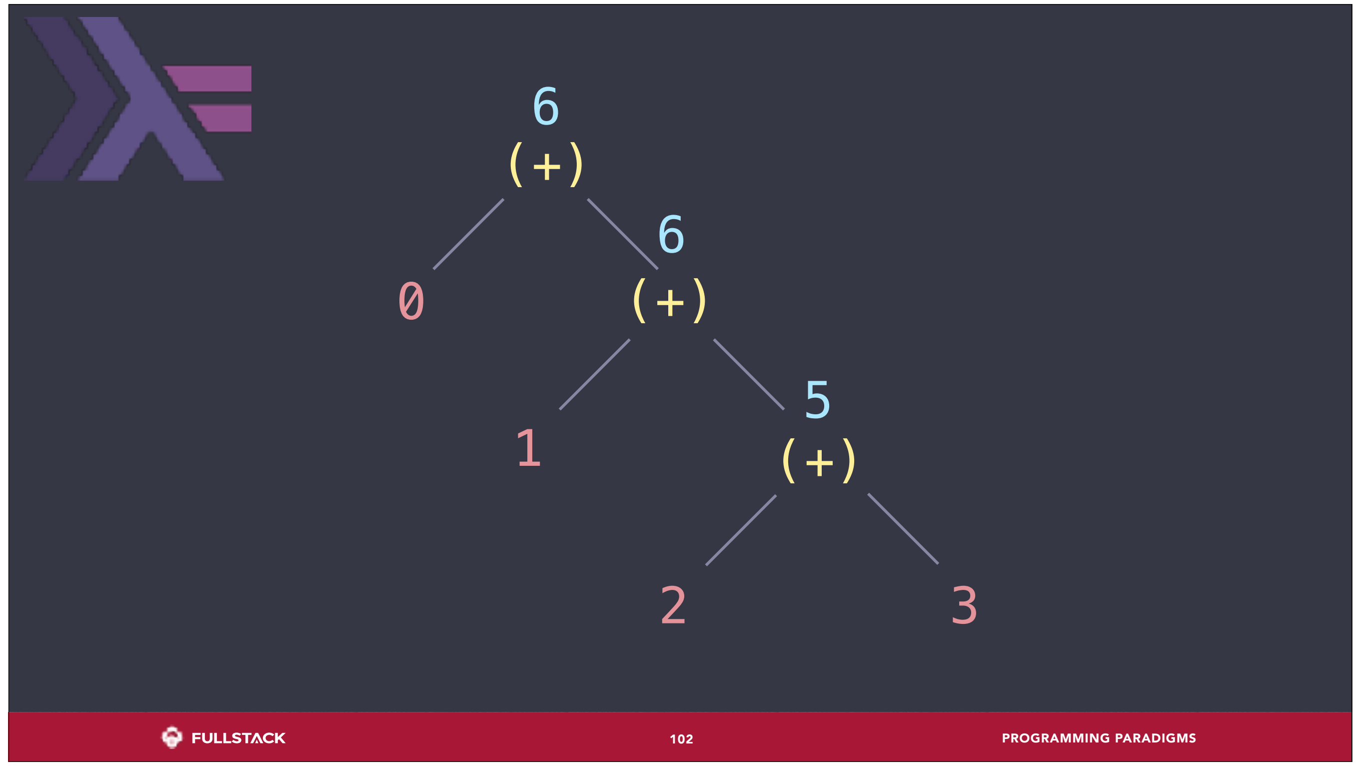


Expanded out, it uses a start value (0) as an accumulation, and recursively applies a function (+) to elements of the list.

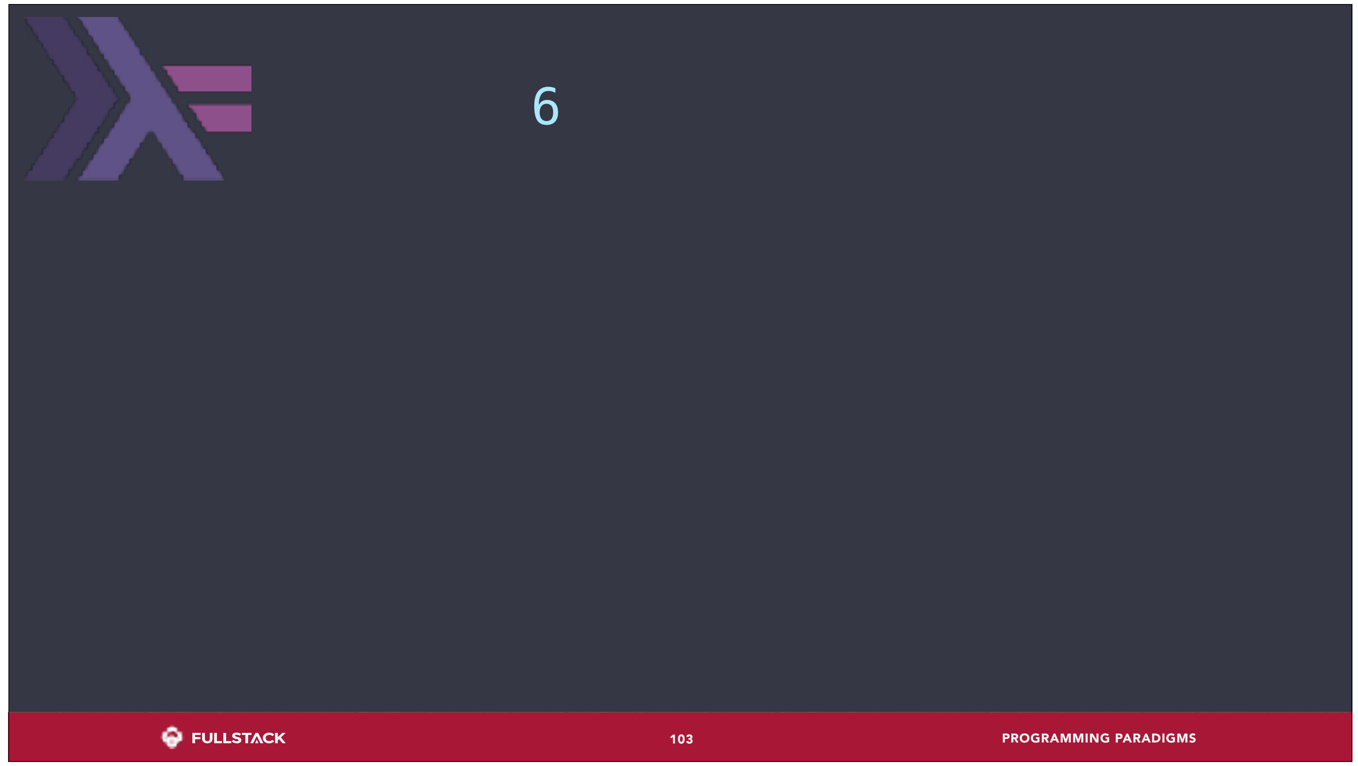


It can then evaluate this tree from the bottom nodes up,


(+)
6
0
(+)
1
5
(+)
2
3



…producing the final value for this expression: 6.



And that's the result of our function call!

Case Study: Mergesort

- Split list in half
- Recursively sort each half
- Merge sorted halves into sorted list
  - Take smaller of the two leading elements
  - Keep doing that until nothing left to take

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Merge sort! You remember this, right? Here it is in Haskell. It's not important to fully understand Haskell syntax, but try to get a sense of the flavor.

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Merge sorting empty list = empty list

Merge sorting singleton = singleton

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Merge sorting anything else =
`merge` sorted `left` with sorted `right`

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
    where (left, right) = splitAt midpoint xs
          midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Any other list

Recursion

`left` and `right` are results of splitting at `midpoint`

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Two return values, in a tuple

Built-in, but not hard to define

and `midpoint` is the length / 2, rounded down

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
  where (left, right) = splitAt midpoint xs
        midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

merging an empty list with 2nd list = 2nd list

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

merging 1st list with an empty list = 1st list

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

merging two lists, each starting w/ some val...

list beginning with some x

list beginning with some y

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

is the smaller val concat'd to merged remainder

construct list beginning with x

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

is the smaller val concat'd to merged remainder

construct list beginning with y

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

sorted = mergesorting this particular list

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

So why show you this snippet of Haskell?

many function applications

& syntactic sugar for function applications

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

Functional programming uses lots of **functions** (obviously). Even some things you may not think of as functions, e.g. operators (==, <) are actually functions.

expressions evaluate to produce values

no such thing as instructions which cause effects

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

Functions **produce values**. They **do not** "do actions" or cause things to change; every expression yields some data, and effects nothing else. This separates **functions** from **procedures**. Even "if-then-else" produces a value, i.e. it's the same as a ternary in JS. Expressions are built out of nested sub-expressions.

defining nouns / relationships, not linear procedures

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

Things are specified more in terms of *what they are* or *their relationships*, less in terms of *instructions to perform*. This gives **functional** programming a lot of overlap with **declarative** programming.

no mutation of state anywhere – all constant

much less specification of order – compiler handles it

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

When everything is defined in terms of its relationships, the order of code doesn't matter nearly as much – the compiler reads the entire source and figures out what order to do things in for you. In pure FP you also never change a value after it is defined; all data is immutable.

so we can take this...

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

So we can modify this…

…and change it to this, or even…

```haskell
mergesort [x] = [x]
mergesort []  = []
mergesort xs  = merge (mergesort left) (mergesort right)
                where midpoint      = length xs `quot` 2
                      (left, right) = splitAt midpoint xs

merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys
merge []     ys     = ys
merge xs     []     = xs

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

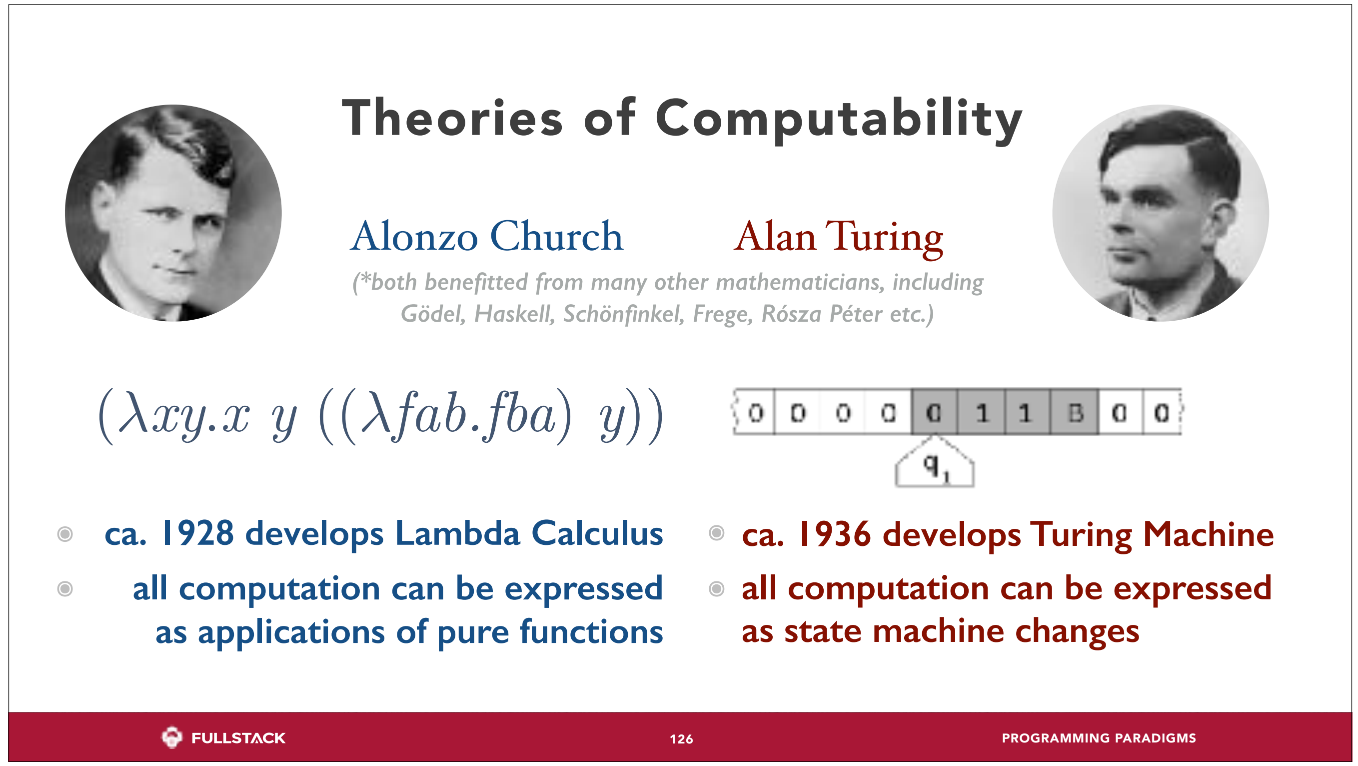
…to this…

```haskell
sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]

merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys
merge []     ys     = ys
merge xs     []     = xs

mergesort [x] = [x]
mergesort []  = []
mergesort xs  = merge (mergesort left) (mergesort right)
                where midpoint      = length xs `quot` 2
                      (left, right) = splitAt midpoint xs
```

…this. Still works!

…and even this, and it all still works perfectly. (Note, *some* order still matters, e.g. arguments & pattern matching – but it is still a *lot less* order).


HISTORY
λ

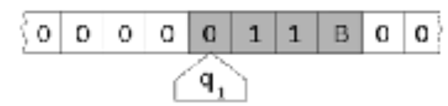


In the 1920s/30s, mathematicians were building on earlier efforts to define the very foundations of rigorous logic. One branch, computability theory, was more or less defined by Alonzo Church and Alan Turing. Church first developed Lambda Calculus, and Turing later published Turing Machines.

Church-Turing Equivalence

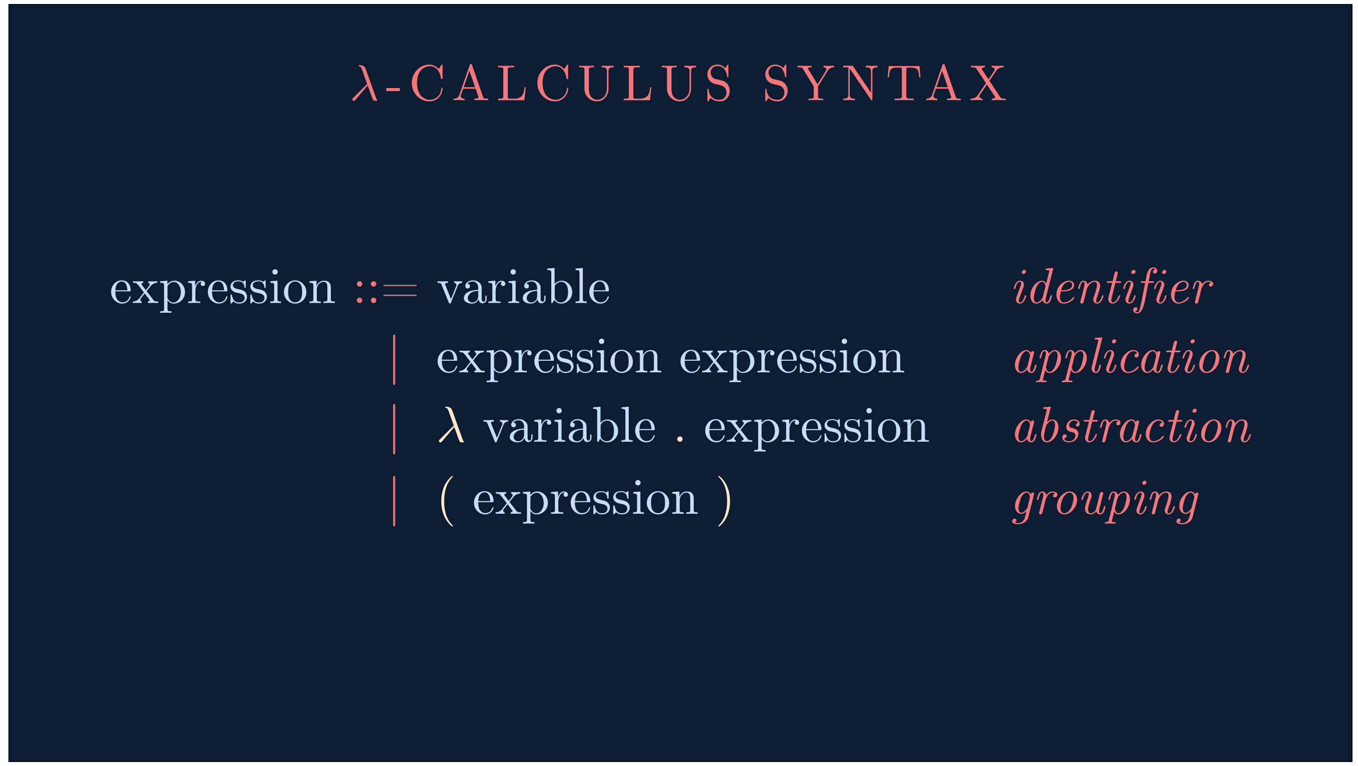
- Turn out to be two different ways to express the same concept
- Everything one can do, the other can

$$(\lambda xy.x\ y\ ((\lambda fab.fba)\ y)) \longleftrightarrow$$



- Exciting because it means code can be entirely abstract
- Exciting because it means we can make real computers

It turned out that both were identically powerful / totally equivalent systems, expressed quite differently. Turing Machines get a lot of press because they are a hypothetical machine which can compute anything; from his work, people developed real computers. LC however is exciting because it has no concept of state; it is entirely abstracted away from any notion of machine.

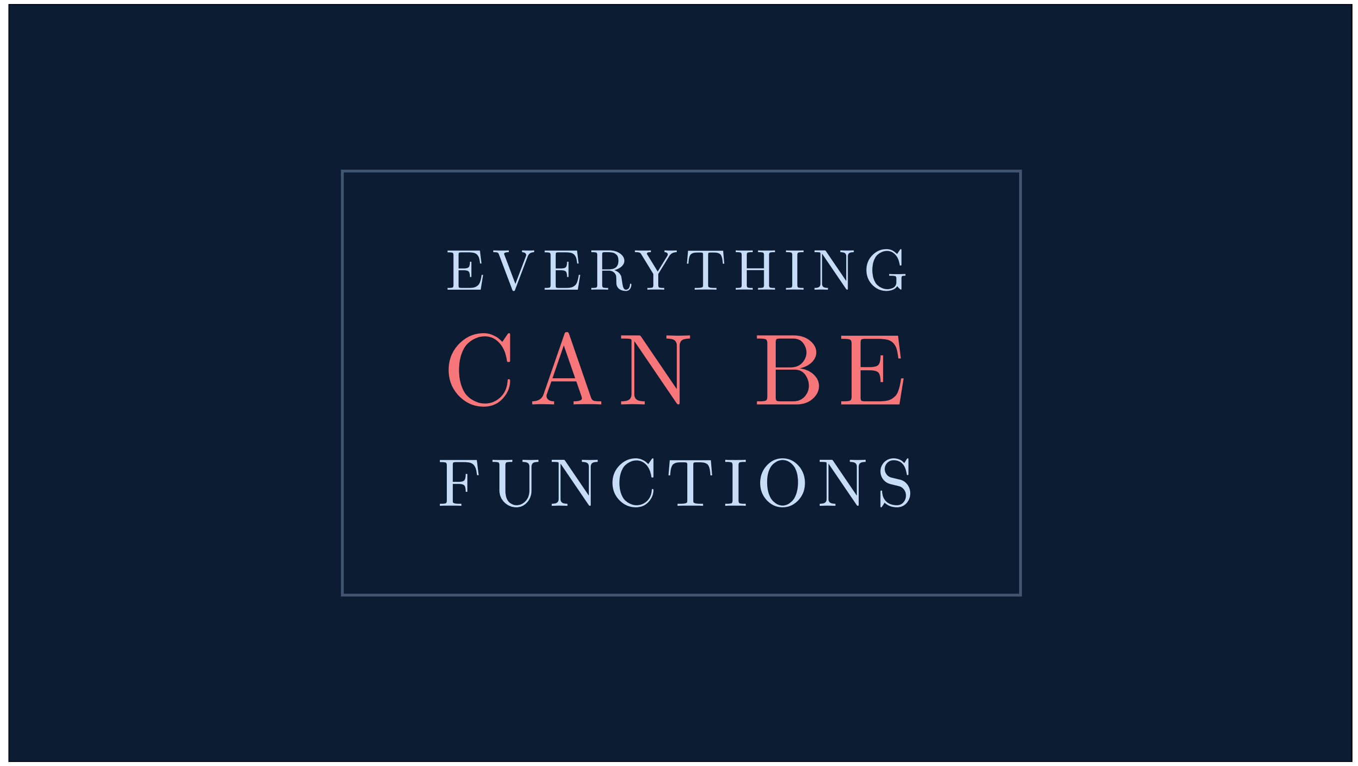


LC has been called the world's smallest programming language. Here it is, in its entirety. This alone is capable of computing anything – including arithmetic and branching logic.

$$(\lambda xy.x)ab$$

```
(x => y => x)(a)(b)
```


Lambda calculus has only a few rules. Everything in it is functions. No numbers, no booleans, no nothing. Only functions (and purely abstract variables, which might stand for functions). A "lambda" really just means a **unary**, **anonymous pure first-class function.**



Gabriel L. says: I have a talk on this which I do sometimes (or it is also recorded on YouTube https://www.youtube.com/watch?v=3VQ382QG-y4).

LISP (Began in 1950s!)

```
(defun csg-intersection-intersect-all (obj-a obj-b)
   (lambda (ray)
     (flet ((inside-p (obj) (lambda (d) (inside-p obj (ray-point ray d)))))
       (merge 'fvector
              (remove-if-not (inside-p obj-b) (intersect-all obj-a ray))
              (remove-if-not (inside-p obj-a) (intersect-all obj-b ray))
              #'<))))
```

- **Later dialects include Common LISP, Scheme, Clojure etc.**

(remaining code was live demo'd)
(slides still under construction)