CONCEPTS OF PROGRAMMING LANGUAGE

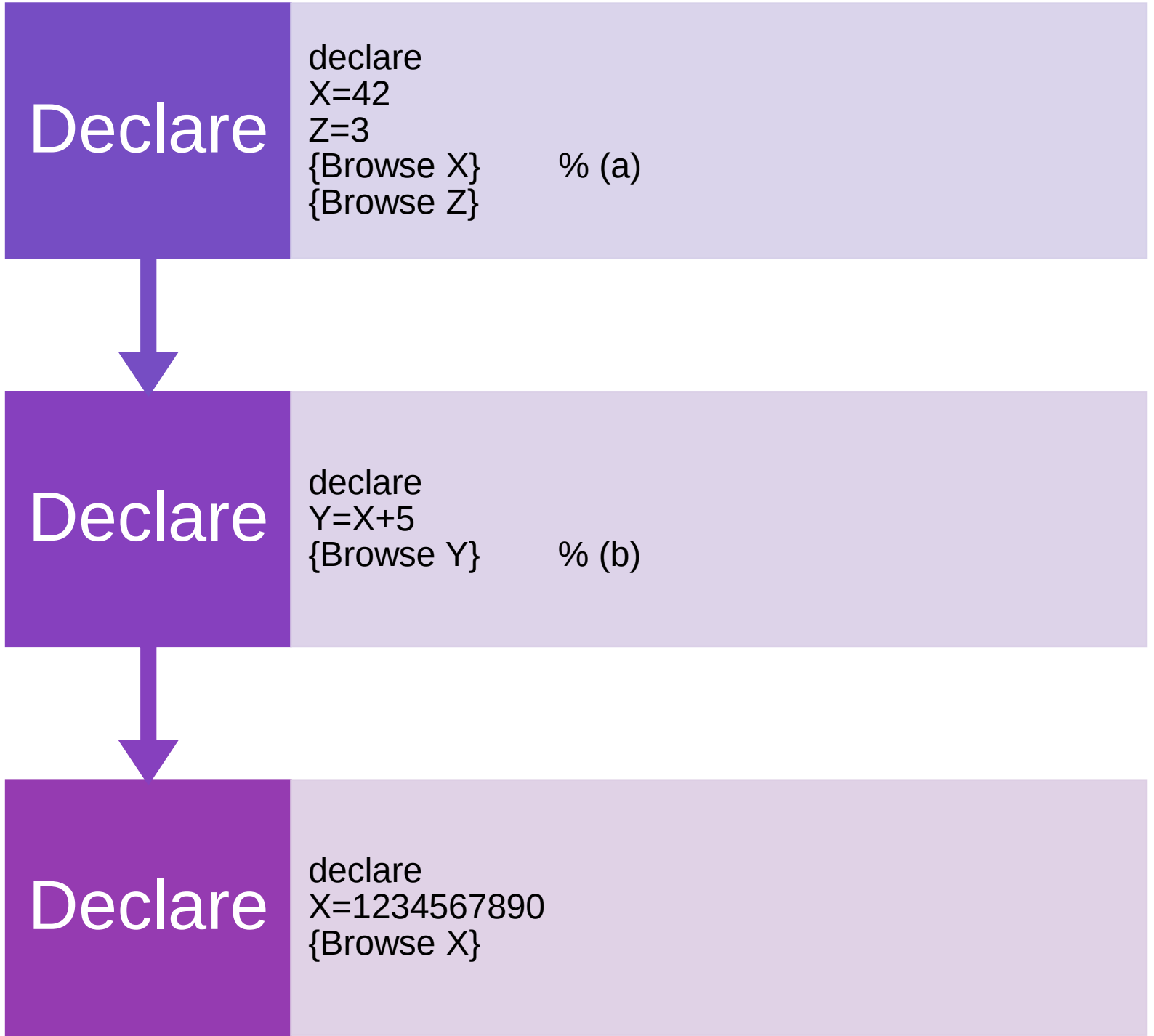
OVERVIEW

Join with Us in Microsoft Team

- With Code

spoc329

Practicle Excercise



Our first paradigm

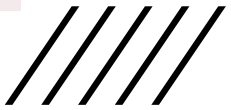
- Functional programming
  - It is one of the simplest paradigms
  - It is the foundation of all the other paradigms
  - It is a form of **declarative programming**

Most Common Paradigms

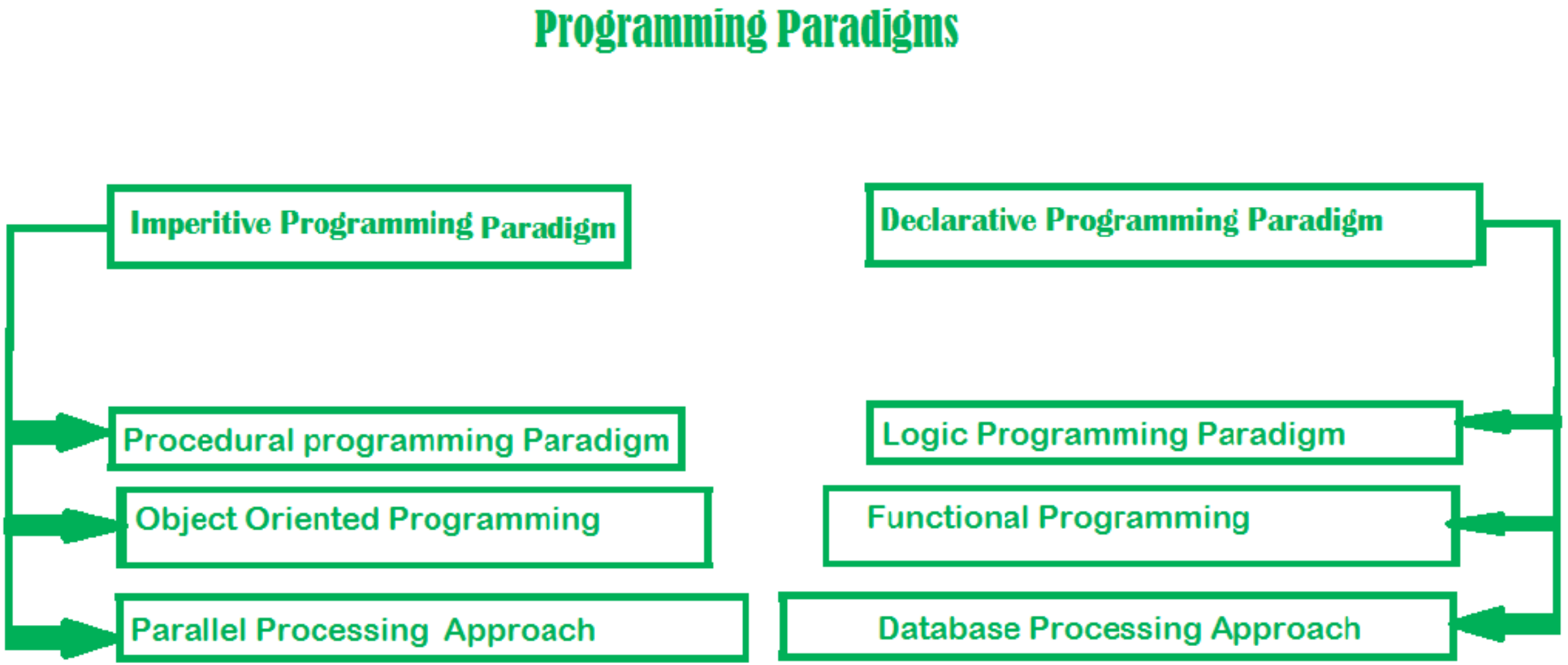
Imperative/ Algorithmic	Declarative		Object-Oriented
	Functional Programming	Logic Programming	
Algol Cobol PL/1 Ada C Modula-3	Lisp Haskell ML Miranda APL	Prolog	Smalltalk Simula C++ Java

Logic Paradigm

```
holi.
cat(bengal).                /* bengal is a cat */
dog(rottweiler).            /* rottweiler is a dog */
likes(Jolie, Kevin).        /* Jolie likes Kevin */
likes(A, Kevin).            /* Everyone likes Kevin */
likes(Jolie, B).            /* Jolie likes everybody */
likes(B, Jolie), likes(Jolie, B).      /* Everybody likes Jolie and Jolie likes everybody */
likes(Jolie, Kevin); likes(Jolie, Ray). /* Jolie likes Kevin or Jolie likes Ray */
not(likes(Jolie, pasta)).      /* Jolie does not like pasta */
```



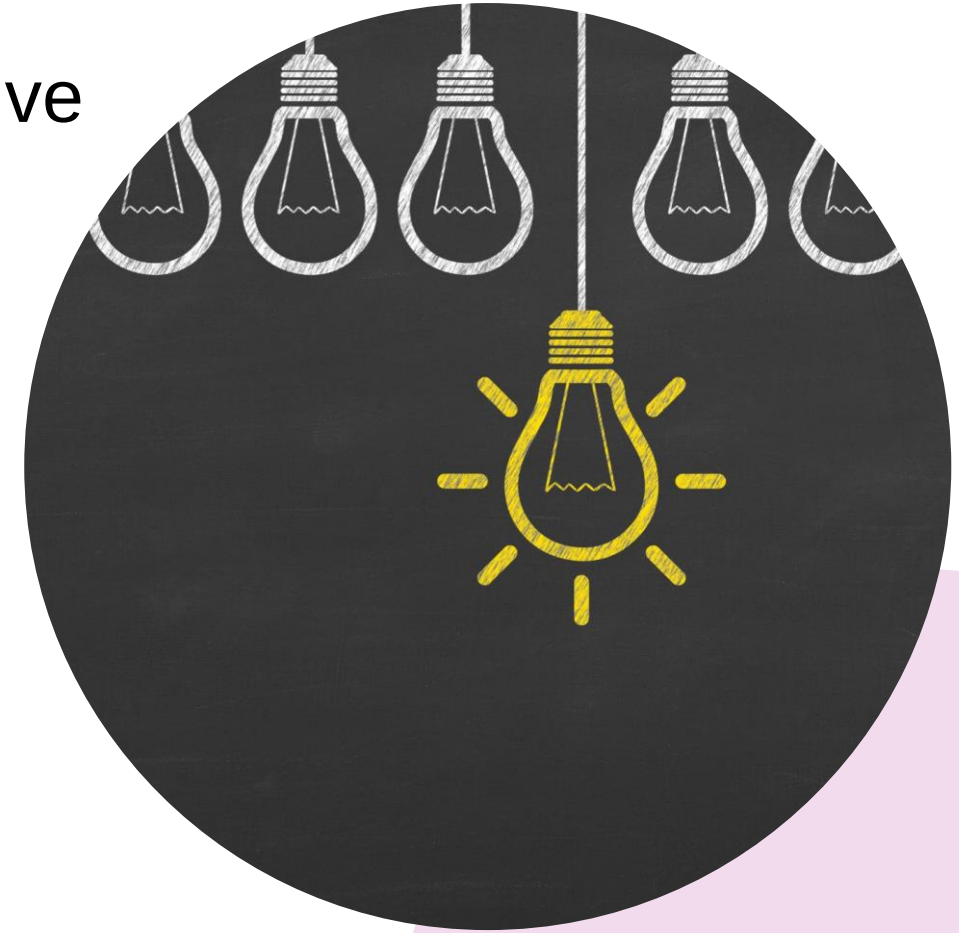
Most Common Paradigms



Difference Between Declarative and Imperative Paradigms

Difference Between Declarative and Imperative Paradigms

- Imperative and declarative programming **achieve the same goals**. They are just **different ways** of thinking about code.
- They have their benefits and drawbacks and there are times to use both.



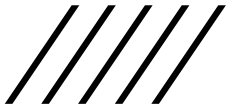
Imperative Paradigm

- As a beginner, you've probably mostly coded in an imperative style: you give the computer a set of instructions to follow, and the computer does **what you want in an easy-to-follow sequence**.
- Imagine we have a list of the world's most commonly-used passwords:

```
const passwords
= [ "123456",
"password",
"admin",
"freecodecamp",
"mypassword123"
, ];
```

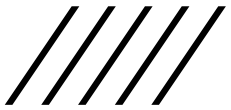
Imperative Paradigm

- Our app is going to check the **user's password** on sign up and not allow them to create a password that is from this list.
- But before we do that, we want to refine this list. We already have code that doesn't allow the user to sign up with a **password less than 9** characters long.
- So we can **reduce** this list to just passwords that are 9 characters or more to speed up our check.
- Imperatively, we would write.

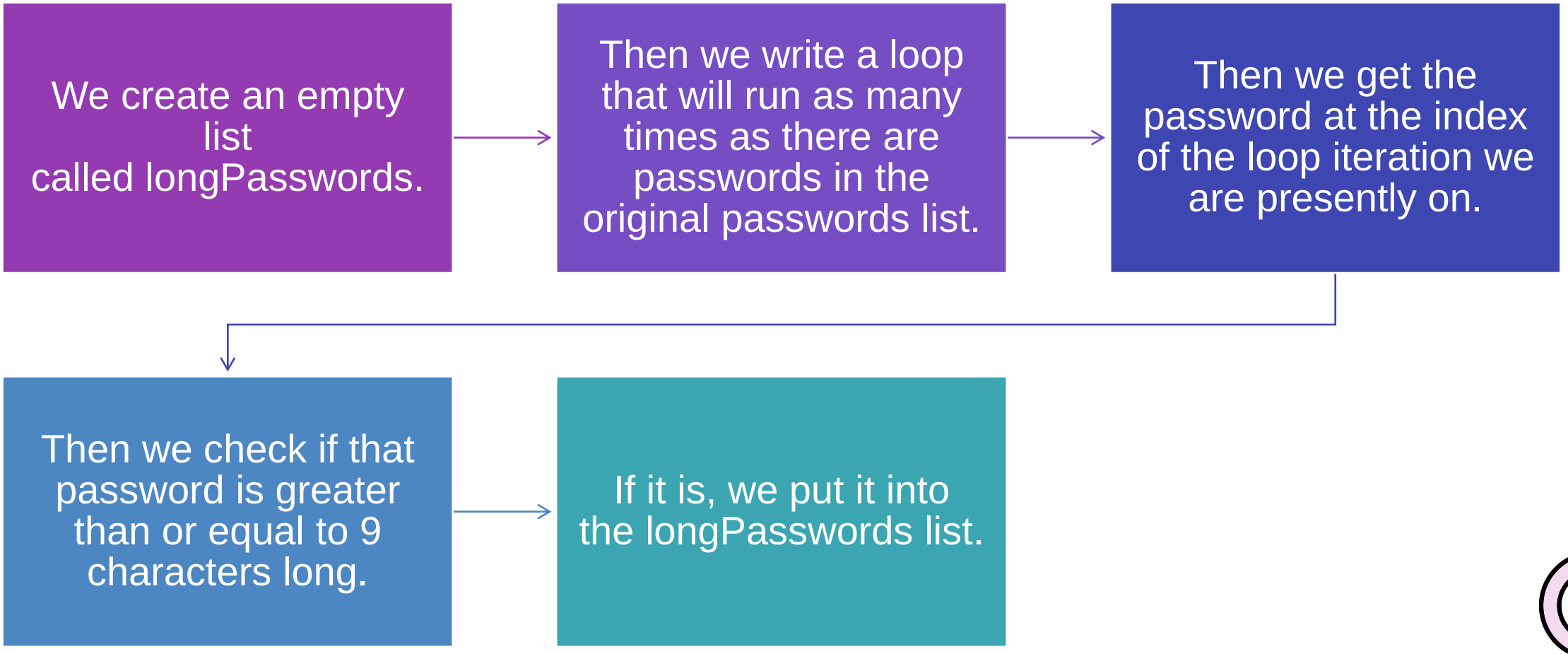


Imperative Paradigm

```
// using the passwords constant from above let
longPasswords = [];
for (let i = 0; i < passwords.length; i++) {
    const password = passwords[i];
    if (password.length >= 9) {
    longPasswords.push(password); }
    }
console.log(longPasswords);
// logs ["freecodecamp", "mypassword123"];
```



Imperative Paradigm



Imperative Paradigm

- One of imperative programming's strengths is the fact that **it is easy to reason** about. Like a computer, we can follow along step by step.

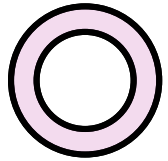
Declarative Paradigm

- But there's another way of thinking about coding – as a process of constantly defining **what things are**. This is referred to as declarative programming.
- Though **imperative** programming is easier to reason about for **beginners**, **declarative** programming allows us to write more **readable code that reflects what exactly we want to see**.
- So instead of giving the computer step by step instructions, we declare what it is we want, and we assign this to the result of some process

Declarative Paradigm

```
// using the passwords constant from above const
longPasswords = passwords.filter(password => password.length >= 9);
console.log(longPasswords);
// logs ["freecodecamp", "mypassword123"];
```

- The list of longPasswords is defined (or declared) as the list of passwords filtered for only passwords greater than or equal to 9 characters.
- The functional programming methods in JavaScript enable us to cleanly declare things.



Declarative Paradigm

- “Passwords” **This is a list of passwords.**
- **“**longPasswords “ **This is a list of only long passwords.** (After running `filter`.)
- **“**password**” This is a list of passwords with ids.** (After running `map`.)
- **“**password.length” **This is a single password.** (After running `find`.)

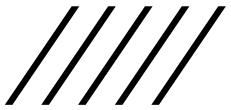
- One of declarative programming’s **strengths** is that it forces us to **ask what we want first**. It is in the naming of these new things that our code becomes expressive and explicit.
- We **encourage** learners to **write declarative code** as often as possible, constantly defining (and refactoring to redefine) **what things are**.
- Rather than hold an entire imperative process in your head, you can hold a more tangible thing in your head with a clear definition.

Declarative programming: the long-term view

- Declarative programming is a vision for the future
  - Just say what result you want (give properties of the result)=Declarative
  - Let the computer figure out how to get there =imperative
  - Declarative versus imperative: properties versus commands
- The whole history of computing is a progression toward more declarative
  - And faster and cheaper

Declarative programming: the short-term view

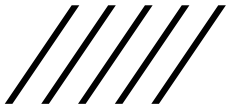
- Declarative programming is the use of mathematics in programming (such as functions and relations)
- A computation calculates a function or a relation
- Use the power of mathematics to simplify programming
- Very common in practice
  - Functional languages: LISP, Scheme, ML, Haskell, OCaml, ...
  - Logic languages (relational): SQL, constraint programming, Prolog, ...
- Also called “programming without state”
- Variables and data structures can’t be updated
- Testing and verification is much simplified
- Declarative versus imperative: stateless versus stateful



Functional Programming	OOP
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “What you are doing”	Focus is on “How you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.
Supports both "Abstraction over Data" and "Abstraction over Behavior".	Supports only "Abstraction over Data".

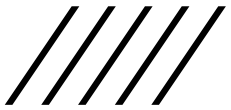
Excercise

- **True Or False?**
  - "Logic programming and functional programming are both a kind of declarative programming." (T)



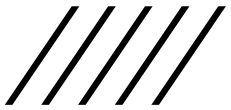
Excercise

- **Complete this sentence:**
  - "Functional programs do not have any internal memory, they are ..."

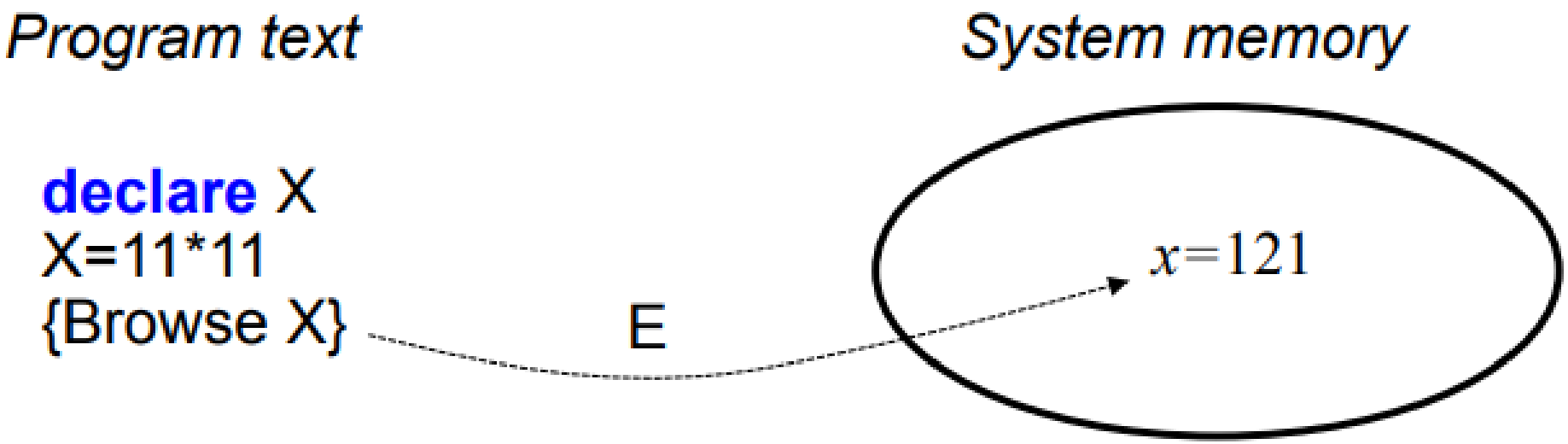


Excercise

- **Complete this sentence:**
  - "Functional programs do not have any internal memory, they are *Stateless* "



Assignment



The assignment instruction X=121 binds the variable x to the value 121

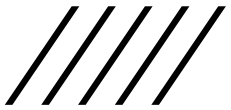
Single assignment

- A variable can only be bound to one value
  - It is called a single-assignment variable
  - Why? Because we are in the functional paradigm!
- Incompatible assignment:

X = 122 signals an error

- Compatible assignment:

X = 121 accepted

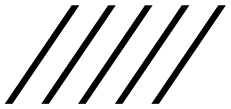


Why single assignment?

- The lack of variables has further implications.
- A language that does not support the ability to change state, also cannot support loops. Why?

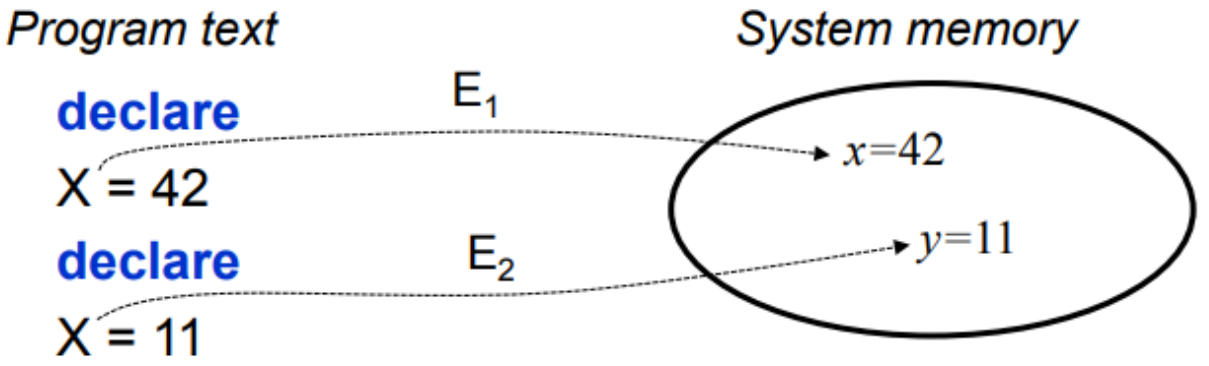
  This is because typical FOR, WHILE, and DO loops require a way of indicating when the loop should stop. Without any state changes, all loops would be **infinite loops!** So, pure functional languages also do not contain loops.

- If we could assign more than once, we could break a correct program
- But how can we program without multiple assignment? Actually, it's easy, as we will see.



Redeclaring an identifier

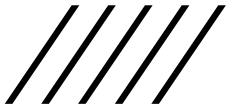
- An identifier can be redeclared
  - The same identifier refers to a different value
  - There is no conflict with single assignment. Each occurrence of X corresponds to a different variable.
- The interactive environment always has the last declaration
  - declare keeps the same correspondance until redeclared (if ever)
  - In this example X will refer to 11



Scope of an identifier occurrence

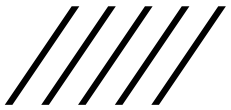
```
local
    X
In
    X = 42 {Browse X}
    Local
        X
    In
        X = 11 {Browse X}
    end
    {Browse X}
end
```

- The instruction *local X in end* declares X between *in* and *end*
- The scope of an identifier occurrence is that part of the program text for which the occurrence corresponds to the same variable declaration
- The scope can be determined by inspecting the program text; no execution is needed. This is called lexical scoping or static scoping.
- **Why is there no conflict between X=42 and X=11, even though variables are single assignment?**
- **What will the third Browse display?**



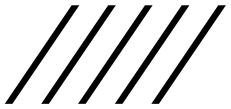
Tips on Oz syntax

- At this point you can see that Oz syntax **is not like most syntaxes** you may have seen before
- The most popular syntax in mainstream languages (**C++, Java) is « C-like** », where identifiers are statically typed (« int i; ») and can start with lowercase, and code blocks are delimited by braces
  - Oz syntax is definitely not C-like!
- Oz syntax is inspired by many languages: Prolog (logic programming), Scheme and ML (functional programming), C++ and Smalltalk (object-oriented programming), and so forth



Why is Oz syntax different?

- It is different because Oz supports many programming paradigms
- The syntax is carefully designed so that the paradigms don’t interfere with each other
- It’s possible to program in just one paradigm. It’s also possible to program in several paradigms that are cleanly **separated in the program text.**
- So it is important for **you not to get confused** by the differences between Oz syntax and other syntaxes you may know
- Let me explain the main differences so that you will not be hindered by them



Main differences in Oz syntax

1. Identifiers in Oz always start with an uppercase letter
   - Examples: X, Y, Z, Min, Max, Sum, IntToFloat.
   - Why? Because lowercase is used for symbolic constants (atoms).
2. Procedure and function calls in Oz are surrounded by braces { ... }
   - Examples: {Max 1 2}, {SumDigits 999}, {Fold L F U}.
   - Why? Because parentheses are used for record data structures.
3. Local identifiers are introduced by local ... end
   - Examples: local X in X=10+20 {Browse X} end.
   - Why? Because all compound instructions in Oz start with a keyword (here « local ») and terminate with end.
4. Variables in Oz are single assignment
   - Examples: local X Y in X=10 Y=X+20 {Browse Y} end.
   - Why? Because the first paradigm is functional programming. Multiple assignment is a concept that we will introduce later.

Questions

- **Why is Oz syntax so different from C-like syntax?**

Questions

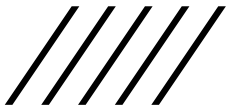
- **Why is Oz syntax so different from C-like syntax?**
  - *Because Oz supports many paradigms that do not interfere with each other*

Functions

- We would like to execute the same code many times, each time with different values for some of the identifiers
  - To avoid repeating the same code, we can define a function
- Function Sqr returns the square of its input:

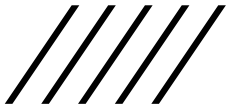
```
declare
fun {Sqr X} X*X
end
```

- The fun keyword identifies the function. The identifier Sqr refers to a variable that is bound to the function.



Numbers

- There are two kinds of numbers in Oz
  - Exact numbers: integers
  - Approximate numbers: floating point
- There is never any automatic conversion from exact to approximate and vice versa
- To convert, we use functions IntToFloat or FloatToInt



Sum of digits function

- Function SumDigits calculates the sum of digits of a three-digit positive integer:

```
declare
fun {SumDigits N}
   (N mod 10) + ((N div 10) mod 10) +
   ((N div 100) mod 10)
end
```

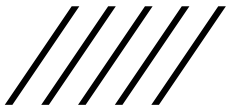
  - **mod** and **div** are integer functions
  - / (division) is a float function
  - * (multiplication) is a function on both floats and integers

SumDigits6

- Sum of digits of a six-digit positive integer

```
fun {SumDigits6 N}
{SumDigits (N div
1000)} + {SumDigits (N
mod 1000)} end
```

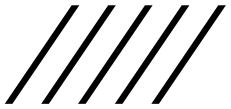
- This is an example of function composition: defining a function in terms of other functions
- This is a key **ability for building large systems**: we can build them in layers, where each layer is built by a different person
- This is the first step toward **data abstraction**



SumDigitsR

```
fun {SumDigitsR N}
 (N mod 10) + {SumDigitsR (N div 10)} end
```

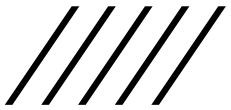
- This function calls itself with a smaller value
- But it never stops: we need to make it stop!



SumDigitsR

```
fun {SumDigitsR N}
    if (N==0) then 0
    else
    (N mod 10) + {SumDigitsR (N div
    10)}
    end
end
```

- This introduces the conditional (if) statement
- This is an example of function recursion: defining a function that calls itself
- This is a key ability for building complex algorithms: we divide a complex problem into simpler subproblems (divide and conquer)



THANKS