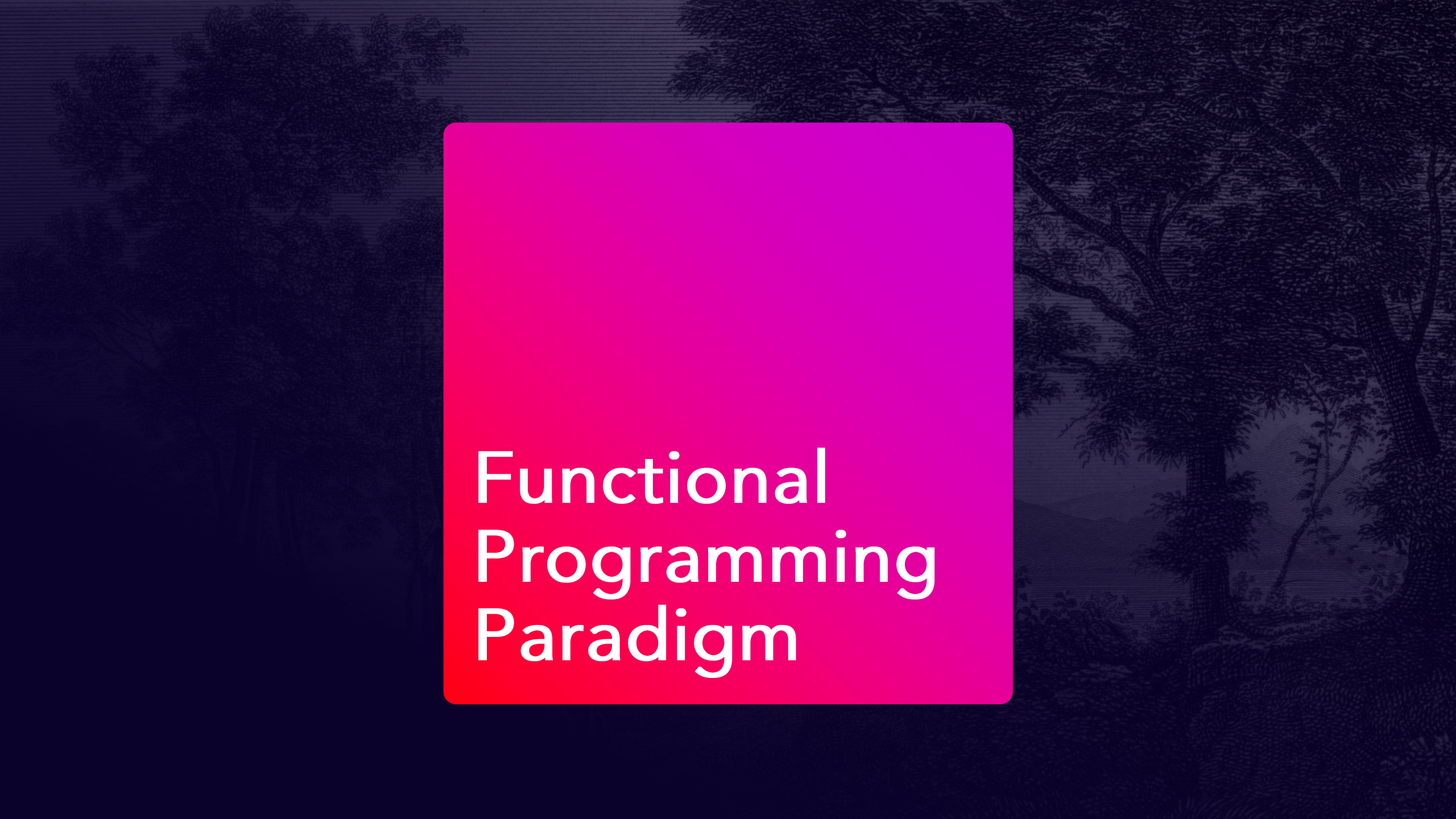

Functional
Programming
Paradigm

Lambda Calculus

During the 1930's a mathematician investigating the foundation of mathematics, **Alonzo Church**, introduced a formal system of expressing computational logic .

Expressions in Lambda Calculus

Let Λ be the set of expressions

1. **Variables**. If x is a variable, then $x \in \Lambda$
2. **Abstractions**. If x is a variable and $\mathcal{M} \in \Lambda$, then $(\lambda x.\mathcal{M}) \in \Lambda$.
3. **Applications**. If $\mathcal{M} \in \Lambda \wedge \mathcal{N} \in \Lambda$, then $(\mathcal{M}\mathcal{N}) \in \Lambda$.

Reductions

Lambda calculus defines 3 ways to reduce/simplify a lambda expression

α equivalence:

α equivalence states that any bound variable, has no inherent meaning and can be replaced by another variable:

$$\lambda x.x \underset{\alpha}{=} \lambda y.y$$

β reductions

β reductions state how to simplify abstractions. This process is similar to applying the function. For example we use the identity function $(\lambda x.x)$ and apply it to some free variable y.

$$(\lambda x.x)y \xrightarrow[\beta]{} y$$

η reductions

η reductions describe equivalencies that arise because of free variables. If x is a variable and does not appear free in $\mathcal{M}$ then:

$$\lambda x.\,(\mathcal{M}\,x) \underset{\eta}{\rightarrow} \mathcal{M}$$

Reduction Example

For example to reduce the following lambda expression, we must first understand what it means.

$$(\lambda x.\, \lambda y.\, (xy))(\lambda x.\, \lambda y.\, (xy))$$

In the outermost level, the expression is the application of $\lambda x.\, \lambda y.\, (xy)$ to itself. It might be tempting automatically apply a β reduction by itself:

$$(\lambda x.\, \lambda y.\, (xy))(\lambda x.\, \lambda y.\, (xy)) \quad \underset{\beta}{\rightarrow} \lambda y.\, ((\lambda x.\, \lambda y.\, (xy))y)$$

$$\underset{\beta}{\rightarrow} \lambda y.\, \lambda y.\, (yy)$$

Reduction Example

But this reduction is actually incorrect because the although x and y appear on both lambda expressions, these variables don't have the same meaning . The x and y variables inside the left lambda expression are **bound** inside this lambda expression.

Reductions Example

The x and y variables outside the left lambda expression (inside the right lambda expression) are **free** in its context, therefore, even though they look the same, it is incorrect to interchange the two variables.

Reduction Example

Two avoid confusion with similarly named variables, it is advisable to apply an α equivalency, to give them different names:

$$(\lambda x.\, \lambda y.\, (xy))(\lambda x.\, \lambda y.\, (xy)) \underset{\alpha}{\equiv} (\lambda x.\, \lambda y.\, (xy))(\lambda u.\, \lambda v.\, (uv))$$

The correct reduction in this case is as follows:

$$\begin{aligned}(\lambda x.\, \lambda y.\, (xy))(\lambda u.\, \lambda v.\, (uv)) &\underset{\beta}{\rightarrow} \lambda y.\, ((\lambda u.\, \lambda v.\, (uv))y) \\ &\underset{\beta}{\rightarrow} \lambda y.\, \lambda v.\, (yv)\end{aligned}$$


the functional
paradigm

Reimagined functions

Lambda calculus evolved from a system of logic foundation with deep roots to computation theory into something that became a basis for programming language design.

Reimagined Functions

Around 1950's programming languages patterned around the framework of lambda calculus started to emerge.

Common Lisp

Haskell

Treating functions differently

One of the biggest difference between your classic imperative programming languages like C and Java and a functional programming language, is how it treat its function

Comparing functions (definition)

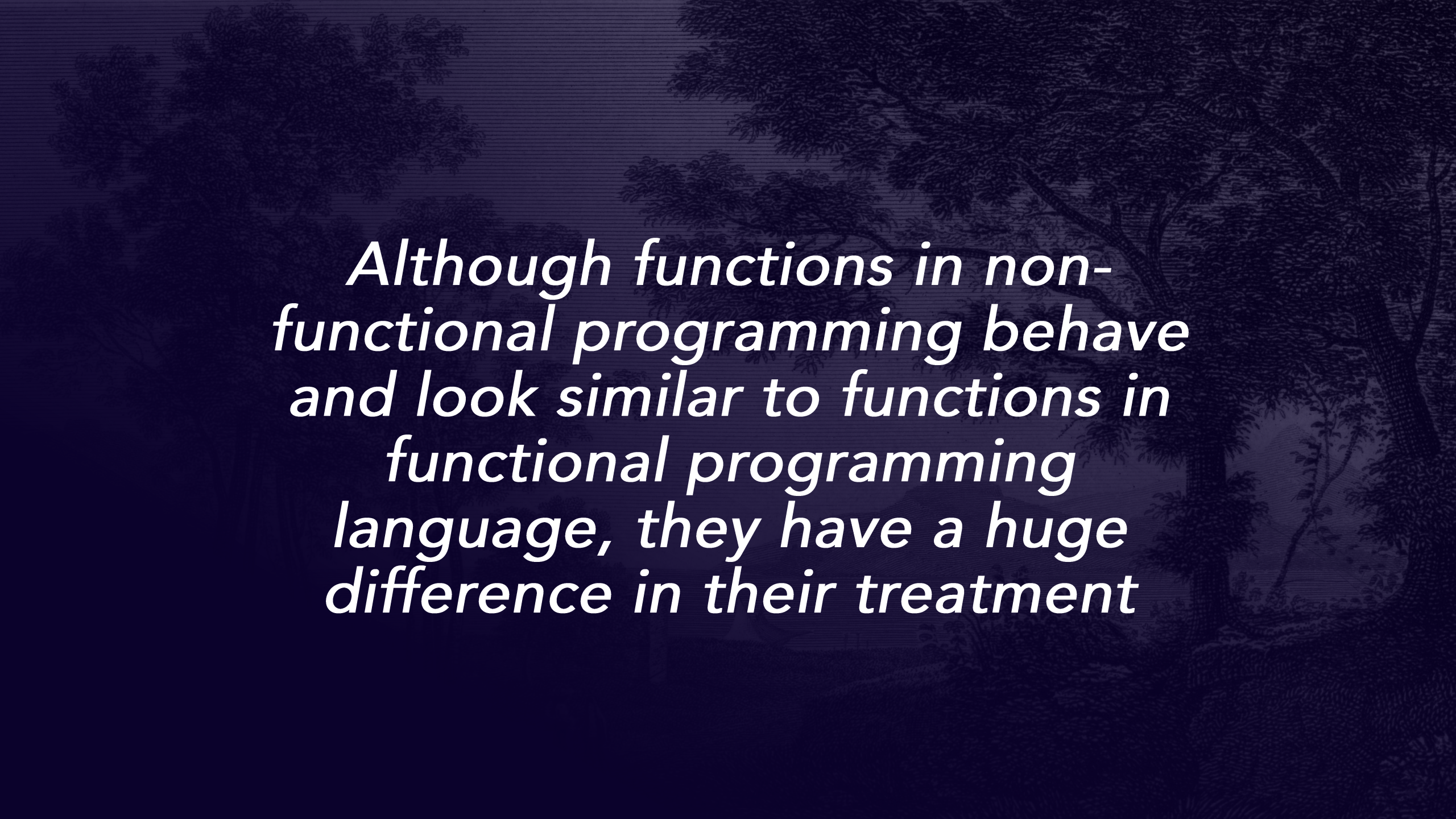
```c
int square(int x){
    return x*x;
}
```

```haskell
square x = x * x
```

Comparing functions (invocation)

```
square(5);
```

```
square 5
```


Although functions in non-functional programming behave and look similar to functions in functional programming language, they have a huge difference in their treatment

Imperative Functions (Procedures or Methods)

A function in C is treated differently from other types of data.

Canonical value types like integers, characters, and arrays can’t be passed on functions and can’t be returned as functions.

Imperative Functions (Procedures or Methods)

```
int* add_to_array(int arr*,int x,int
size){
   for(int i=0;i<size;i++){
     arr[i]+x;
   }
   return arr;
}
```

Imperative Functions (Procedures or Methods)

During runtime, non-functional programming languages interpret the expression `square(5)` as "the number 5 squared" while the expression `square` is just some disembodied function name.

Imperative programming functions during runtime are meaningless unless they are directly invoked.

Higher Order Functions

Functional programming languages treat functions the same way it treats values, you can pass them as in functions and you can return them as well.

```
s:: Int -> Int
func x = x + 1

p::Int -> Int
func x = x - 1

applytwice:: (Int-> Int) -> Int -> Int
applytwice f x = f (f x)
```

Higher Order Functions

The third function is what we call a higher order function. A higher order function is a function that either accepts a function as a parameter or returns a function parameter or both.

```
applytwice:: (Int-> Int) -> Int ->
Int
applytwice f x = f (f x)
```

Higher Order Functions

By defining a function like t
his we can do something lik
e this during runtime:

```
ghci> applytwice s 3
5
ghci> applytwice p 3
1
```


Why is this
even a feature?

Higher order functions

This is not some arbitrary added feature as well since this is directly patterned from lambda calculus:

$$\begin{aligned} \text{let } S &= \lambda n.\, \lambda s.\, \lambda z.\, (s(nsz)) \\ T &= \lambda f.\, \lambda x.\, (f\,(f\,x)) \\ \overline{5} &= TS\overline{3} \end{aligned}$$

Higher Order Functions

In lambda calculus an abstraction and an application does not restrict anyone from the type of expressions bound to variables.

Higher Order Functions

On the other side of the coin, a function, in functional programming will also let you return functions the same way you return any other kind of data.

Higher Order Functions

Suppose we have different functions that when applied to an integer, produces that integer plus a certain integer.

```
addTwo::Int -> Int
addTwo = x = x + 2

addThree::Int -> Int
addThree x = x + 3

addFour:: Int -> Int
addFour = x + 4
```

Higher Order Functions

We can generalize these functions into a *function-maker* function

```
addMaker::Int -> (Int->Int)
addMaker x = (\y -> x + y)
```

Higher Order Functions

This is a representation of an expression we already know from lambda calculus. addMaker is basically an implementation of the lambda expression below.

$$\lambda y.\,\mathrm{add}\ x y$$

Higher Order Functions (lambda expressions)

addMaker produces a **lambda expression**, which essentially behaves exactly like a function. This allows you to create functions during runtime

```
ghci> addSix = addMaker 6
```

Higher Order Functions (lambda expressions)

Simply writing the expression addSix on your terminal will yield you an error, because printing addSix doesn't really have a meaning outside the world of lambda calculus.

Higher Order Functions (lambda expressions)

But since addSix is a lambda that behaves exactly like a function, you can apply addSix to an integer and it will give you a meaningful answer.

```
ghci> addSix 3
9
```

Higher Order Functions (lambda expressions)

In fact you can even omit the part where you bind the value returned by addMaker to a name, and instead use it directly.

```
ghci> (addMaker 7) 4
11
```

Higher Order Functions (lambda expressions)

This nifty trick right here is the reason why lambda expressions are also called **anonymous functions** since these expressions on their own don't have a name.

Higher Order Functions (closure)

A higher order function is not only producing the lambda inside its definition. It produces a construct called a **closure** which is the function definition described by the lambda and the environment of the function call.

Higher Order Functions (closure)

The extra data, called environment, is the reason why the lambda, `(\y -> x + y)` makes sense outside the context of `addMaker`.

Multiple Parameters and Partial Application

If we look back to lambda calculus you'll notice how abstractions are defined to be:

$$(\lambda x.\mathcal{M}) \in \Lambda$$

Multiple Parameters and Partial Application

Abstractions are defined to have exactly one parameter which is different from functions in programming which can have 0 or more parameters

Multiple Parameters and Partial Application

These functions are just disguised to have multiple parameters.

These functions are just several single parameter functions combined to simulate multiple parameter functions.

Multiple Parameters and Partial Application

As an example: a function add that adds two numbers may look like multiple parameter functions:

```
plus x y = x + y
```

Multiple Parameters and Partial Application

But internally this function is equivalent to two lambda calculus abstractions, nested together to simulate multiparameterness

```
plus = \x -> (\y -> x + y)
```

Multiple Parameters and Partial Application

Here add is a higher-level function that accepts a single argument x and produces a lambda expression.

This expression is a direct implementation of the following lambda calculus abstraction:

$$\text{plus} = \lambda x.\, \lambda y.\, \text{add}\ x y$$

Multiple Parameters and Partial Application

Haskell's -> operator is right associative so you can write the same plus function as:

```
plus = \x -> \y -> x + y
```

You can even omit the first -> and the \ near y and it will me an the same lambda expressi on:

```
plus = \x y -> x + y
```

Multiple Parameters and Partial Application

Now when you want to apply this function we write:

```
ghci> (plus 3) 4
7
```

Multiple Parameters and Partial Application

Function applications in Haskell is also left associative so you can omit the parentheses:

```
ghci> plus 3 4
7
```

Multiple Parameters and Partial Application

All multiple parameter functions and lambdas in Haskell are implemented are nested single parameter abstractions in disguise

```
plus x y = x + y

plus = \x -> (\y -> x + y)
```

Multiple Parameters and Partial Application

This means that the expression(plus 3) will have the same meaning regardless of the way you define plus. The expression (plus 3) has a special name, it is called a **partial application**.

Multiple Parameters and Partial Application

When you apply a function that is supposed to accept n parameters to m values (where $m < n$), instead of getting the value, you get a partial application of that function which will evaluate to a lambda expression.

Multiple Parameters and Partial Application

The process of converting a multi parameter function or lambda to a nested single parameter lambda is called **currying**.

Haskell Brooks Curry

Functional Purity and the Absence of States

One of the most distinguishing aspects of functional programming is its philosophy of statelessness.

Programmers exposed to state and mutation find that the concept of state is natural and maybe even inevitable.

Functional Purity and the Absence of States

Functional paradigm challenges this concept and offers a much safer and mathematically intuitive philosophy.

In the perspective of functional programming, there is no state, and everything is immutable.

Pure Functions

Functions in functional programming languages like Haskell are (arguably) the closest computer representation of a mathematical function. We call these functions, **pure functions**.

$$f : \mathbb{Z} \to \mathbb{Z}$$
$$f(x) = x + 1$$

```
f::Int -> Int
f x = x + 1
```

Impure Functions (side-effect)

```
int square(int x){
    addToExternalLogger
("calculating square");
  return x*x;
}
```

Impure Functions (side-effect)

```
int square(int x){
   addToExternalLogger("calculating square");
   return x*x;
}
```

Impure Functions (mutation)

```c
void increaseArray(int *a, int size){
  for(int i = 0; i < size; i++)
    a[i]+1;
}
```

Impure Functions (relational)

```
String headsortails(){
    if(randInt()%2==0)
        return "heads";
    else
        return "tails";
}
```

Functional Purity

A pure function must satisfy these two:

- A pure function has no side-effects
- A pure functions output must be dependent on the inputs alone

Functional Purity

A good way to test if a function is pure is if you can (theoretically) create an infinitely long lookup table such that looking up the value for a specific input is perfectly identical to calling the function with the same input.

x	f(x)
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100
11	121
12	144
13	169
14	196
15	225
16	256

Absence of mutation

Purely functional programming languages like Haskell, lacks the mechanism to mutate anything.

Using the "=" operator in functional languages binds the value on the right-hand side to the left-hand side.

Absence of mutation

Destructive assignment in Haskell is in fact not allowed

```
x = 0
x = 1

main.hs:2:1: error:
    Multiple declarations of
‘x’
    Declared at: main.hs:1:1
                 main.hs:2:1
```

Consequences of Statelessness

Although Haskell's functions are our best representation of a mathematical function, this does not mean that it is pragmatically better than the classic impure functions of C.

Consequences of Statelessness

Restricting a programming language against side-effects seems like an artificial disadvantage introduced only to faithfully implement mathematical functions.

Programming without state

```
void f(int *x, int y){
  *x = *x + y;
  printf("%d\n",*x);
  return;
}
int main(void) {
  int x = 0;
  f(&x, 3);
  x = x - 2;
  f(&x, 3);
}
```

Programming without state

Building a library of functions perfectly working without worrying about side-effects makes the system easier to understand and more resilient to changes.

Programming without state

On small chunks of code, managing the consequences of having states such as global variables, will be trivial since you can reasonably track which variables are global (*or external in general*) and which functions interact with the global variables.

Programming without state

As the system grows, using functions and variables without double checking for side effects becomes much harder.

Programming without state

Consequently the whole system becomes a nightmare of impure functions on top of impure functions which may unexpectedly affect other parts of the system.

Programming without state

Even with all these disadvantages in the state-full mutable paradigm of imperative programming, one can still code robust and harmonious systems.

Programming without state

Programmers must be extra careful writing your code with discipline, only using globals and side-effects when it is safe and necessary

Programming without state

After all, being able to code with states can be thought of as an extra feature. You can be a C programmer and just treat all your variables as immutable and all your functions as pure.

Disadvantages of Functional Programming

Functional programming has its own set of disadvantages as well, most of them related to this artificial crutch of statelessness and immutability.

Disadvantages of functional programming

Creating new values instead of changing an existing variable has extra overhead in both processing and memory, making functional programming slower and less efficient.

Disadvantages of functional programming

Programming without state can be difficult to do for certain mechanisms (*Like the external logger for example*).

Disadvantages of functional programming

Simulating mechanisms like this may introduce conflict on how you compose your functions.

It's not impossible though, you just have to learn some category theory concepts such as **monads** and functors

Disadvantages of functional programming

For most people who are used imperative programming, recursion, does not feel natural at all

Disadvantages of Functional Programming

Nevertheless, these disadvantages are not insurmountable. In fact, there are a plenty of systems written in functional programming languages running without issues.

Disadvantages of Functional Programming

Functional programming has had the reputation of being more conceptual and fancier than the classic imperative programming language.

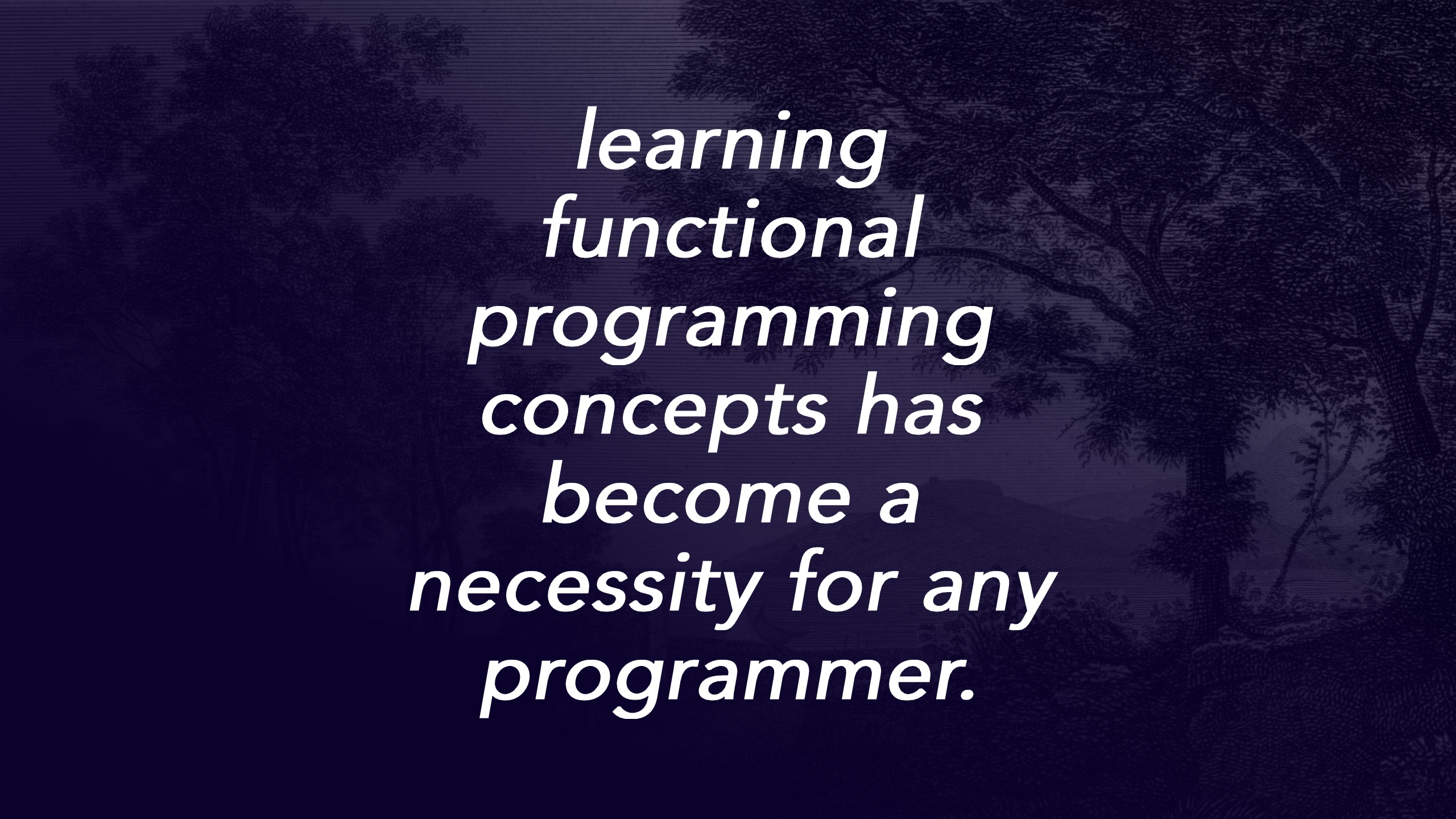
But recently functional programming languages like Haskell, F# and Scala has enjoyed improvements that has elevated it to be as pragmatic as your classic C, C++ or Java

Disadvantages of Functional Programming

Functional programming has gained a rise in popularity to an extent that mainstream programming languages like C#, Python, and JavaScript have started to introduced features patterned from pure functional programming languages (Features such as higher-order functions and lambdas).

Disadvantages of Functional Programming

Functional programming has gained a rise in popularity to an extent that mainstream programming languages like C#, Python, and JavaScript have started to introduced features patterned from pure functional programming languages (Features such as higher-order functions and lambdas).


learning
functional
programming
concepts has
become a
necessity for any
programmer.

References

Church, A. (1932). *A Set of Postulates for the Foundation of Logic. Annals of Mathematics*,33 (2), second series, 346-366. doi:10.2307/1968337

Steele, A. (1996). *The Evolution of Lisp. History of programming languages*, 233-330. doi:10.1145/234286.1057818

References

Common Lisp Logo from https://common-lisp.net/ by Lisp under CC BY-SA

Haskell Logo from Thompson-Wheeler logo on the haskell wiki under the haskell wiki copyright license