marp: true theme: default paginate: true class: lead

Slide 1: Title

Hello everyone!
My name is Phat Tran Dai, and today I'm excited to take you into the world of **concatenative programming languages**.

Now, you might be thinking, “Concatenative... what?” — and that’s exactly the point. These languages are often overlooked, even though they offer something truly elegant and conceptually refreshing.

They're minimalist, expressive, and they challenge how we usually think about writing code. So let’s dive into what makes them so interesting.

Slide 2: Agenda

Here’s a quick overview of what we’ll explore:

- First, I’ll explain what concatenative languages actually are.
- Then, we’ll look at where they came from — the historical roots.
- I’ll walk you through their core ideas and features.
- We’ll see simple code examples and then build up to recursive functions.
- I’ll show you a longer real-world-like example — Rule 110.
- After that, we’ll reflect on their pros and cons and talk about where they’re used today.
- Finally, I’ll offer some resources for diving deeper.

Slide 3: What Are Concatenative Languages?

So, what is a concatenative language?

In short, it’s a **functional programming paradigm** where programs are built by **concatenating functions** — just placing them next to each other.

These languages operate in **point-free** style, which means that instead of naming variables or function parameters, we simply compose behavior. That's it — no function calls with parentheses, no named inputs. This may sound restrictive at first, but it actually leads to a very natural flow of logic.

The language works by maintaining a **stack**. You push values onto it and apply functions, which pop arguments off the stack and push results back on.

If you've ever used a calculator in RPN mode — reverse Polish notation — you've already encountered this idea. It's about evaluating expressions in the order they appear, using an implicit context: the stack.

Slide 4: Historical Background

Concatenative programming isn't new.

It goes back to **Forth**, a language created by Charles Moore around 1970. Forth was designed to be compact, efficient, and direct — ideal for embedded systems where resources are tight. It's still used today in places like spacecraft, medical devices, and microcontrollers.

Later, in the 1980s, Adobe introduced **PostScript**, a page description language. It used a concatenative model to describe the layout of text and images on a page. Every time you print a PDF, you're invoking some of this logic.

Fast forward to the early 2000s, and we start to see purely functional concatenative languages like:

- **Joy**, which formalized the theoretical model of concatenative composition.
- **Factor**, which aimed to be a practical, general-purpose language.
- And **Cat**, an academic attempt to bring static typing into the concatenative world.

So, while these languages are rare, their influence has touched both theory and industry.

Slide 5: Key Features

Let's talk about what makes concatenative languages stand out.

First, they rely on **concatenation** — you build programs by simply placing functions and values next to each other. No parentheses, no nesting, no variable declarations.

Second, they use **point-free style**, which means functions are not defined in terms of explicit

arguments. This encourages modular, reusable code.

Third, **stack-based execution** is at the heart of everything. Think of the stack as a conveyor belt. Values come in, functions process them, and new values roll out the other side.

Lastly, many concatenative languages emphasize **functional purity** — no mutation, no global state, and often no side effects. This leads to easier reasoning and better optimization potential.

Slide 6: Examples of Languages

Here are some real examples of concatenative languages.

- **Forth** is still alive today in embedded systems and bootloaders. NASA has used it in mission-critical software.
- **PostScript** is used in nearly every printer and PDF renderer.
- **Joy** inspired theoretical work in functional programming.
- **Factor** is perhaps the most approachable concatenative language for programmers today. It has an IDE, a REPL, and a rich standard library.
- **Cat** was more academic, focusing on static typing and type inference.

Each of these shares the core ideas we've discussed, but with different goals.

Slide 7: Basic Syntax Example (Factor)

Let's look at a basic example in Factor.

We write:

```
2 3 + 4 *
```

What's happening here?

- First, `2` is pushed onto the stack → `[2]`
- Then `3` → `[2, 3]`
- `+` pops those two, adds them → `[5]`
- Then we push `4` → `[5, 4]`

- Finally, `*` pops those two → result is `20`

This is function composition through literal ordering. You don't need to name anything. The stack takes care of context. Once you get used to it, it feels like programming with Lego blocks.

Let's make this more concrete with a small example.

- First, 3 is pushed → stack is [3].
- Then 4 is pushed → [3, 4].
- Next comes over, which copies the second item from the top and pushes it to the top. So over turns [3, 4] into [3, 4, 3].
- Now we do +. This adds the top two numbers: 3 + 4 = 7, so the stack becomes [3, 7].
- Then we do a swap. That flips the top two elements: [3, 7] → [7, 3].
- And finally, drop removes the top value, which is now 3, leaving just [7].
- So the result of the whole expression is 7.

Slide 8: Defining Functions (Factor)

You can define your own functions too.

Here's a function that squares a number:

```
: square ( n -- n² ) dup * ;
```

This means: take the top of the stack, duplicate it, and multiply.

You'd use it like this:

```
5 square .
```

The `.` prints the result. The output is `25`.

The `( n -- n² )` part is a stack comment — it documents the inputs and outputs of the function, which is very helpful when you're working stack-first.

Slide 9: Recursion Example (Factorial)

Let's define a factorial function recursively:

```
: factorial ( n -- n! )
    dup 1 <=
    [ drop 1 ]
    [ dup 1 - factorial * ]
    if ;
```

If the number is 1 or less, we return 1. Otherwise, we recursively compute factorial(n-1), then multiply.

Let's walk through `3 factorial` step by step:

- `[3]` → not less than 1, so call `2 factorial`
- `[2]` → still not less than 1 → call `1 factorial`
- `[1]` → now return 1
- Backtrack: `2 * 1 = 2` , then `3 * 2 = 6`

Result: `6`

This is compact and expressive, and shows the power of stack-based recursion.

Slide 10: Fibonacci Example

Now for Fibonacci:

```
: fib ( n -- fib(n) )
    dup 2 < [ drop 1 ]
    [ dup 1 - fib swap 2 - fib + ] if ;
```

We check if `n < 2` , and if so, return 1. Otherwise, we compute `fib(n-1)` and `fib(n-2)` and sum the results.

Running `6 fib .` produces `13` .

It's elegant but dense — as you can see, you need to think in terms of stack state at every step. That's both the beauty and the challenge.

Slide 11: Longer Example – Rule 110

Let me now show you a longer program.

This is **Rule 110**, a cellular automaton similar to Conway's Game of Life. I wrote it in my own language, inspired by Factor.

The code runs over multiple generations, applying rules to bits, printing each line as a visual pattern. It uses loops, memory, and macros — all stack-based.

This isn't just a toy — it's a demonstration that you can write full, expressive, even artistic programs in this style.

Slide 12: Rule 110 Output

And here's the output.

Each line is a snapshot of the grid over time. Stars represent `1` s, spaces are `0` s. The result is a complex, structured pattern emerging from a very simple rule.

This shows how even minimal concatenative logic can lead to emergent behavior — which is one reason these languages are studied in computer science and complexity theory.

Slide 13: Strengths ✅

To summarize, here are the key strengths of concatenative languages:

- The syntax is minimal, often beautiful.
- Function composition is simple and powerful.
- Stack-based execution offers precise control.
- Programs are often smaller and easier to refactor.
- They can be highly efficient and optimize well.

They reward clarity and modular design.

Slide 14: Weaknesses ⚠️

That said, they're not perfect.

- The point-free style can be **unfamiliar and hard to debug**.
- Managing the stack requires careful thought — one wrong `dup` or `drop` and your logic falls apart.
- Tooling is **limited** compared to Python or JavaScript.
- There's **very little industry adoption** — you won't find Factor jobs on LinkedIn.

So, they're best suited for experimentation, learning, and specialized use.

Slide 15: Real-world Applications

Despite their niche status, concatenative languages do have real-world applications:

- **Forth** is still used in low-level embedded systems — it's small, fast, and close to the hardware.
- **PostScript** powers PDF rendering and printing.
- In research, **Joy** and **Factor** are used to explore new programming models and concepts in functional design.

And increasingly, enthusiasts and language designers are exploring concatenative ideas in new DSLs and tooling.

Slide 16: Why Learn Them?

So why should you care about concatenative programming?

Because it forces you to think differently. It teaches you to compose small, reusable units and manage control flow explicitly. It sharpens your understanding of stacks, recursion, and data flow.

Even if you never write a production system in Factor, the mental models you develop here are valuable — and transferable to other paradigms like functional and reactive programming.

Slide 17: Resources

If this talk has piqued your curiosity, here's where to go next:

- Factorcode.org — the official Factor language site, with an interactive REPL and tutorials.
- Concatenative.org — a wiki collecting information on Joy, Cat, and others.
- *Thinking Forth* by Leo Brodie — a classic that teaches stack-based thinking like no other book.

All highly recommended.

Slide 18: Q&A

Thank you so much for your time and attention.

I hope this presentation gave you a new way of looking at programming — not through variables and classes, but through flow and composition.