Concatenative Programming Languages

An Introduction

Agenda

1. What Are Concatenative Languages?
2. Historical Background
3. Key Features
4. Examples
5. Syntax and Execution
6. Strengths and Weaknesses
7. Applications
8. Why Learn Them?
9. Resources

What Are Concatenative Languages?

- **Functional programming paradigm**
- Functions composed by **concatenation**
- Programs built by **concatenating functions**
- **Point-free style** (no explicit arguments, no named variables)
- Execution based on a **stack**

Reverse Polish Notation

- Linear notation of expressions

Standard	RPN
(1 + 2) * 3	1 2 + 3 *
((1 + 2) * 3) / 4	1 2 + 3 * 4 /
-(-(1 + 2) * 3) / 4	1 2 + neg 3 * neg 4 /
f(g(x))	x g f
sin(-pi * cos(x))	pi neg x cos * sin

Historical Background 1

- Originated from **Forth** (Charles Moore, ~1970)
- Popularized by Adobe's **PostScript** (1982)

```
%PDF-1.7
%µí®û
4 0 obj
<< /Length 5 0 R
   /Filter /FlateDecode
>>
stream
```

Historical Background 2

- Modern purely functional languages:
    - **Joy** (2001, Manfred von Thun)
    - **Factor** (2003, Slava Pestov)
    - **Cat** (2006, Christopher Diggins)

Key Features

- **Concatenation** of functions (functions chained linearly)
- **Point-free (tacit)** programming
- **Stack-based data manipulation**
- Emphasis on **functional purity** (no side effects, mostly)

Examples of Concatenative Languages

- **Forth** *(procedural)*
- **PostScript** *(graphics)*
- **Joy** *(theoretical)*
- **Factor** *(practical, modern)*
- **Cat** *(functional, statically typed)*

Syntax and Execution

```
2 3 + 4 *
```

Stack execution:

Operation	Stack
`nop`	`[] <- top of the stack`
`2`	`[2]`
`3`	`[2,3]`
`+`	`[5]`
`4`	`[5,4]`
`*`	`[20]`

Stack Manipulation Built-ins

Here are common stack operators:

Word	Stack Effect	Description	Example
`dup`	(x -- x x)	Duplicate top item	1 dup `<=>` 1 1
`drop`	(x --)	Remove top item	1 2 drop `<=>` 1
`swap`	(x y -- y x)	Swap top two items	1 2 swap `<=>` 2 1
`over`	(x y -- x y x)	Copy 2nd item to top	1 2 over `<=>` 1 2 1
`rot`	(x y z -- y z x)	Rotate top 3 items	1 2 3 rot `<=>` 3 1 2
`nip`	(x y -- y)	Remove 2nd item	1 2 nip `<=>` 2
`tuck`	(x y -- y x y)	Copy top under 2nd	1 2 tuck `<=>` 2 1 2

Stack Manipulation Built-ins - Example

`3 4 over + swap drop`

Stack	Operation
`[]`	
`[3]`	3
`[3, 4]`	4
`[3, 4, 3]`	over
`[3, 7]`	+
`[7, 3]`	swap
`[7]`	drop

Advanced Syntax (Factor)

Define a square function:

```
: square ( n -- n² ) dup * ;
```

Use it:

```
5 square .
```

Output: `25`

Recursion

Define a Factorial recursively:

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

Use it:

```
5 factorial .
```

Output: 120

```
: factorial ( n -- n! )
    dup 1 <=                          // Check if n <= 1
    [ drop 1 ]                        // If true, result is 1
    [ dup 1 - factorial * ]           // If false, n * factorial(n-1)
    if ;
```

Execution for `3 factorial`:

Stack	Operation
`[]`	no operation
`[3]`	`3`
`[3 3]`	`dup`
`[3 3 1]`	`1`

```
: factorial ( n -- n! )
    dup 1 <=                           // Check if n <= 1
    [ drop 1 ]                         // If true, result is 1
    [ dup 1 - factorial * ]            // If false, n * factorial(n-1)
    if ;
```

Stack	Operation
`[3 F]`	`<=` → false
`[3 F [ drop 1 ]]`	`[ drop 1 ]`
`[3 F [drop 1] [dup 1 - factorial *]]`	`[dup 1 - factorial *]`
`[3 [dup 1 - factorial *]]`	`if`
`[3]`	`dup 1 - factorial *`

```
: factorial ( n -- n! )
    dup 1 <=                        // Check if n <= 1
    [ drop 1 ]                      // If true, result is 1
    [ dup 1 - factorial * ]   // If false, n * factorial(n-1)
    if ;
```

Stack	Operation
`[3]`	`dup 1 - factorial *`
`[3 3]`	`1 - factorial *`
`[3 3 1]`	`- factorial *`
`[3 2]`	`factorial *`
...	...

```
: factorial ( n -- n! )
    dup 1 <=                    // Check if n <= 1
    [ drop 1 ]                  // If true, result is 1
    [ dup 1 - factorial * ]     // If false, n * factorial(n-1)
    if ;
```

Stack	Operation
`[3 2 1]`	`factorial *`
`[3 2 1]`	`dup 1 <=` + `*` + `*`
`[3 2 1 1]`	`1 <=` + `branches` + `* *`
`[3 2 1 1 1]`	`<=` + `branches` + `*` + `*`
`[3 2 1 T]`	`branches` + `* *`
`[3 2 1 T [drop 1] [dup 1 - factorial *]]`	`if` + `* *`

```
: factorial ( n -- n! )
    dup 1 <=                      // Check if n <= 1
    [ drop 1 ]                    // If true, result is 1
    [ dup 1 - factorial * ]       // If false, n * factorial(n-1)
    if ;
```

Stack	Operation
`[3 2 1 [drop 1]]`	`* *`
`[3 2 1]`	`drop 1 * *`
`[3 2]`	`1 * *`
`[3 2 1]`	`* *`
`[3 2]`	`*`
`[6]`	result

Another Mini-Program: Fibonacci Sequence

Define Fibonacci recursively:

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

Use it:

```
6 fib .
```

Output: `13`

Explanation of Fibonacci Program

```
: fib ( n -- fib(n) )
    dup 2 <                         ! Check if n < 2
    [ drop 1 ]                      ! Base case: fib(0)=1, fib(1)=1
    [ dup 1 - fib                   ! Recursive call: fib(n-1)
      swap 2 - fib                  ! Recursive call: fib(n-2)
      + ]                           ! Sum two previous results
    if ;
```

Longer Example - Rule 110

```
// examples/rule_110.p
inluce "stdlib.p"

macro N 20 end

mem N 2 - + 1 !8

0 while dup N 2 - < do
    0 while dup N < do
        dup mem + *8 if
            dup mem + N + '*' !8
        else
            dup mem + N + ' ' !8
        end
        1 +
    end

    mem + N + 10 !8 N 1 + mem N + 1 1 syscall3 drop

    mem *8 1 << mem 1 + *8 |

    1 while dup N 2 - < do
        swap 1 << 7 & over mem + 1 + *8 |
        dup2 110 swap >> 1 &
        swap mem + swap !8 swap
        1 +
    end drop drop

    1 +
end drop
```

```
                           *
                          **
                         ***
                        ** *
                       *****
                      **   *
                     ***  **
                    ** * ***
                   ******* *
                  **     ***
                 ***    ** *
                ** *   *****
               *****  **   *
              **   * ***  **
             ***  **** * ***
            ** * **   ***** *
           ******** **   ***
          **      ****  ** *
         ***     **  * *****
        ** *    *** ****   *
       *****   ** ***  *  **
      **   *  ***** * ** ***
     ***  ** **   ******** *
    ** * ******  **      ***
   *******    * ***     ** *
  **     *   **** *    *****
 ***    **  **  ***   **   *
** *   *** *** ** *  ***  **
```

- Written in my language: https://github.com/phatt-23/stack-pl

Strengths ✅

- Elegant, minimal syntax
- Easy function composition and reuse
- Predictable data flow
- Optimization-friendly structure

Weaknesses ⚠️

- Unfamiliar style and syntax (steep learning curve)
- Complexity with managing large stacks
- Limited industry adoption (not mainstream)
- Less mature tooling and libraries
- **Nobody uses it**

Practical Applications

- **Embedded Systems**: Forth in firmware and low-level programming
- **Graphics and Printing**: PostScript for PDF generation, printing tech
- **Research and Education**: Joy, Factor for experimental programming

Why Learn Concatenative Languages?

- Deepens understanding of functional paradigms
- Enhances logical and compositional thinking
- Offers new a perspective into programming
- Great for niche applications and experimentation
- For fun :))))

Resources 📚

- Factor Language
- Joy Language
- Cat Language
- Book: *"Thinking Forth"* by Leo Brodie

Thank you for your attention!