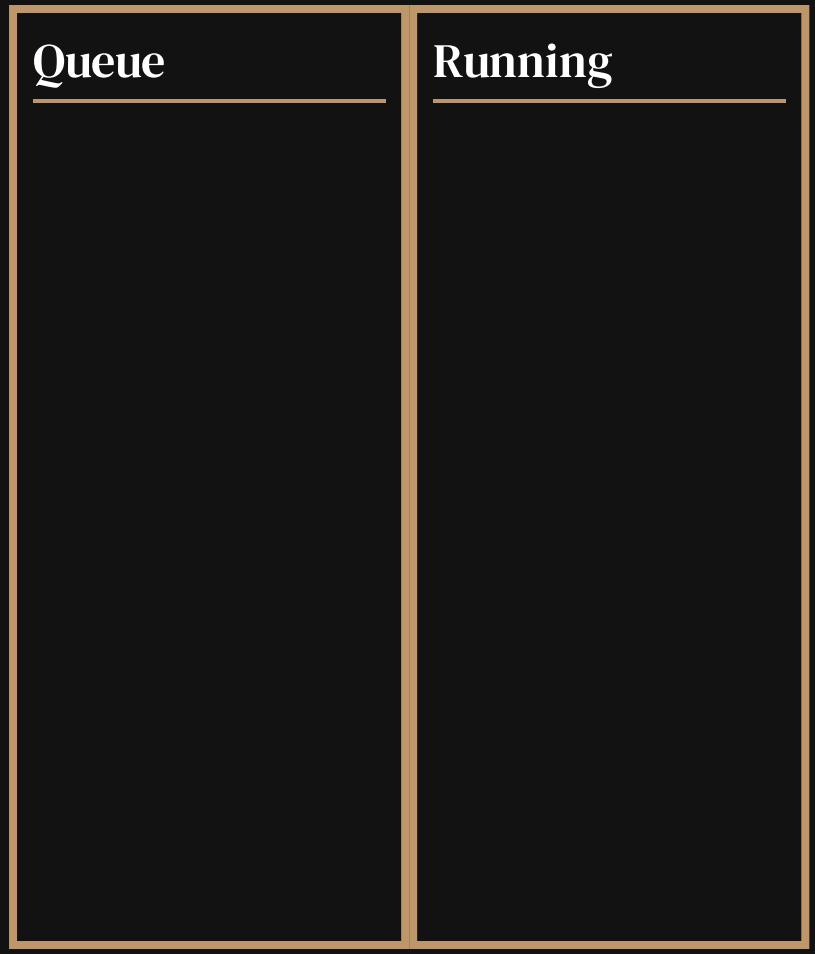
Weaver

Simple scripting language for the joy of coding.

```
"Hello World!"
    ▷ echo()
```


Queue
Running

Overview

Weaver is a simple scripting language that prioritizes readability and simplicity.

Everything is plain objects and functions, most of the code is just concatenating functions together to solve problems.

```
arr := [1, 2, 3, 4]

arr
    ▷ filter(|n| n % 2 == 0)
    ▷ len()
    ▷ echo() // [2, 4]
```

Syntax

Basic syntax and feal of the language.

In this section we will cover the basics of the language, and some of the idioms that make it unique.

Values

```
"Hello World!"          // string
123                     // number
1.23                    // number
true                    // boolean
false                   // boolean
[1, "2", [3]]           // array
{"a": 1, "b": 2}        // object
{a: 1, b: 2}            // object (shorthand)
|a, b| a + b            // lambda
|a, b| { return a + b } // lambda
nil                     // nil (null)
```

There is only one type in Weaver that indicates the absence of a value, it is `nil`.

Note: `{a: 1, b: 2}` is shorthand for `{"a": 1, "b": 2}`.

Binary Operations

```
1   1 + 2            // 3     (number)
2   1.0 + 2          // 3.0   (number)
3   2.3 + 3.4        // 5.7   (number)
4   1 - 2            // -1    (number)
5   1 * 2            // 2     (number)
6   1 / 2            // 0.5   (number)
7   8 % 2            // 0     (number)
8
9   true && false    // false (boolean)
10  true || false    // true  (boolean)
11
12  nil || 1         // 1     (number)
13  error() || "foo" // "foo" (string)
14
15  "hello " + "world" // "hello world" (string)
16
17  add := |a, b| a + b
18  add(1, 2)        // 3     (number)
19  1 ▷ add(2)       // 3     (number)
```

Binary operations are very familiar to other languages, with highlight be operators like pipe operator (`▷`), and lazy evaluation of binray and `&&` and `||` operators that works for booleans and other values also based on if they are "truthy" or not.

Note: `||` returns the first "truthy" value, and `&&` returns the first "falsy" value.

Truthy Values

Weaver boolean operators work with boolean expressions `true` and `false` as well as any other value in the language.

Values that are considered "falsey" are `nil`, `error`, `false`.

```
greet := |name| echo("Hello " + (name || "unknown"));
greet("John") // Hello John
greet() // Hello unknown
```

Type Coercion

Also there is no type coercion, so you must be explicit about the conversion of types, This is a deliberate design decision to avoid mistakes of other languages, like the enfamous javascript examples below.

```
// Weaver
int(true) + int(false)  == 1
12 / int("6")           == 12
"foo" + string(15 + 3)  == "foo18"
```

In other words: What you see is what you get.

Functions

Functions are the core of the language, they are "first class", that is they can be passed around and used as values.

There are no special syntax for functions, you just assign a function value to a variable and call it.

```
add := |a, b| a + b
add(1)  // error: illegal operands number + nil (missing argument)
```

This allows for expressive and concise code.

```
[1, 2, 3, 4]
    ▷ filter(|n| n % 2 == 0)
    ▷ echo() // [2, 4]
```

Control Flow

If Statement

```
if (true) {
    echo("true is truthy!")
}

if (false) {
    echo("this will not run")
}

arr := [1, 2, 3, 4]
if (len(arr) ⩾ 4) {
    arr[0] + arr[3] ▷ echo() // 5
}
```

Ternary Operator

```
n := 1
what := n % 2 == 0 ? "even" | "odd"
echo(what) // "even"
```

Loops

```
// prints 0 to 9
for i := 0; i < 10; i++ {
    echo(i);
}

// prints 0 to 9
for i in 0..9 {
    echo(i);
}

// prints 0 to 9
i := 0;
while (i < 10) {
    echo(i);
    i++;
}
```

Match Statement

Pattern matching is a very powerful feature of Weaver, it allows you to write conditional logic based on the "shape" of the value.

Match cases are evaluated in order, from top to bottom, until a match is found.

```
x := "foo"
match x {
    "bar" ⇒ echo("bar is matched"),
    "nor" ⇒ echo("nor is matched"),
    "foo" ⇒ echo("finally foo is matched"),
    _ ⇒ echo("if nothing else matches"),
}
```

```
arr := [1, 2, 3, 4]
match arr {
    [1, 2] ⇒ {
        echo("arr starts with [1, 2]");
    },
    [2, 3] ⇒ {
        echo("arr starts with [2, 3]");
    },
    _ ⇒ {
        echo("otherwise");
    }
}
```

```
n := 15
match n {
    0..10 ⇒ echo("n is between 0 and 10"),
    11..20 ⇒ echo("n is between 11 and 20"),
    _ ⇒ echo("n is greater than 20"),
}
```

```
students := [
    { name: "Youssef", gpa: 3.5 },
    { name: "John", gpa: 1.5 },
    { name: "Mahmoud", gpa: 2.0 },
];

for student in students {
    match student {
        { name, gpa: 0..1.5} ⇒ echo(name + " is good"),
        { name, gpa: 2..5} ⇒ echo(name + " is really good"),
    }
}

// output:
// Youssef is good
// John is good
// Mahmoud is really good
```

```
match x {
    // matches string "foo"
    "foo" ⇒ {},
    // matches number 123
    123 ⇒ {},
    // matches number 1.4
    1.4 ⇒ {},
    // matches any number between 0 and 10
    0..10 ⇒ {},
    // matches any number less or equal to 10
    ..10 ⇒ {},
    // matches any number greater or equal to 5
    5.. ⇒ {},
    // matches array with two elements, where each element matches the pattern
    [<pattern>, <pattern>] ⇒ {},
    // matches object with "key" matching the pattern and "other" matching the pattern
    { key: <pattern>, other: <pattern> } ⇒ {},
    // matches any string and puts it in the variable s
    string(s) ⇒ {},
    // matches any number and puts it in the variable n
    number(n) ⇒ {},
    // matches error with the first pattern for the error message and the second pattern for the error details
    error(<pattern>, <pattern>) ⇒ {},
    // matches any value and puts it in the variable foo
    foo ⇒ {},
}
```

Match Patterns

Patterns can be as nested as you want.

```
match x {
    [1, { someArray: [a, b, c] }] if a > b && b > c ⇒ echo("MATCH!"),
    _ ⇒ echo("NO MATCH!"),
}
```

Match Guards

Match guards are a way to add additional conditions to a match case.

```
match x {
    [..10, ..10] ⇒ {},
    // same as above
    [a, b] if a ⩽ 10 && b ⩽ 10 ⇒ {},
}
```

Error Handling

Weaver has a unique approach to error handling. Errors are values, just like numbers or strings. When a function returns an error, it's *automatically propagated* up the call stack unless explicitly handled. This is different from languages like JavaScript that use exceptions and `try/catch` blocks.

```
// Example: Automatic error propagation
divide := |a, b| {
    return b == 0 ? error("Division by zero", {divisor: b}) | a / b;
};

result := divide(10, 0)
echo(result) // This line will NOT execute
```

In the example above, `divide(10, 0)` returns an error. Because we didn't handle it explicitly, the error is automatically returned, and the `echo(result)` line is never reached.

You can opt-out of automatic propagation using the `!` operator:

```
// Example: Opting out of automatic propagation
result := divide(10, 0)!
echo("This line WILL execute")
echo(result) // Prints the error value
```

By adding `!` after the function call, we tell Weaver that we want to handle the potential error ourselves. `result` will now contain the error value.

We can then use pattern matching to handle the error:

```
// Example: Handling errors with pattern matching
result := divide(10, 0)!
match result {
    error(msg, data) ⇒ {
        echo("Error: " + msg);          // Prints "Error: Division by zero"
        echo("Divisor: " + string(data.divisor)); // Prints "Divisor: 0"
    },
    n ⇒ echo("Result: " + string(n)), // This won't execute in this case
}
```

Here's a more realistic example, fetching data from a URL:

```
// Example: Real-world HTTP request
response := http:get("https://example.com/api/data")!
match response {
    error(msg, data) ⇒ {
        echo("HTTP request failed: " + msg);
        echo("Status code: " + string(data.statusCode));
    },
    res ⇒ {
        echo("Response body:")
        echo(res.body)
    }
}
```

This approach makes error handling explicit and integrates seamlessly with Weaver's pattern matching.

Comparison with `try/catch` (JavaScript):

```javascript
// JavaScript try/catch example
try {
  let response = await fetch("https://example.com/api/data");
  let data = await response.json();
  console.log(data);
} catch (error) {
  console.error("An error occurred:", error);
}
```

```
response := http:get("https://example.com/api/data")!
match response {
    error(msg, { status }) ⇒ {
        echo("An error occurred: " + msg);
    },
    { body } ⇒ {
        echo(body)
    }
}
```