STUDENTS.md — How the Basil BASIC Interpreter Works

This guide explains, in simple terms, how Basil (a small, modern BASIC-style language) runs your code. You'll see the big pieces and how they fit together, with short examples and a mental model you can keep in your head while you write programs.

If you want the deep, PhD-level details later, read TECHNICAL.md. This file is the gentle on-ramp.

0) Glossary

- AST - Abstract Syntax Tree
- Bytecode - a list of small instructions for the VM to execute
- CGI - Common Gateway Interface
- CGI templating - mixing HTML and Basil in one file to render web pages
- Caching - saving compiled bytecode next to your source in a .basilx file
- Class - a collection of variables and methods
- Compiler - a program that turns the AST into bytecode
- Error message - a message that the VM reports when something goes wrong
- Format version - the version of the bytecode format used by the VM
- Function - a named block of code that can be called from other code
- Instantiate - make an object instance from a class definition
- Interpreter - a program that runs your code
- Keywords - words that have special meaning in Basil
- Lexer - a program that turns characters into tokens
- Line number - the line number in your source code that the VM is currently executing
- Method - a function that belongs to an object
- Object - an instance of a class
- Parser - a program that turns tokens into an AST
- Property - a variable that belongs to an object
- Scope - the set of variables that can be accessed from a given point in your code
- SetLine - tiny marker inserted into bytecode to map source line numbers to bytecode line numbers
- VM - Virtual Machine

1) What is Basil? What is an interpreter?

An interpreter is a program that runs your code. Basil's tool actually does two things:

1. It compiles your Basil source code into a compact "bytecode."
2. It executes that bytecode on a virtual machine (VM).
3. Think of bytecode like a simple set of instructions for a tiny imaginary computer. The VM is the tiny computer.

- Basil is a tiny programming language inspired by BASIC. It's designed to be easy to read and teach, but powerful enough for scripts and small web pages.
- Basil easy to use for data science and machine learning, and for web development.
- Basil designed with AI in mind. It's designed to be easy for AI to write, and produce clear, readable, and editable code for humans to review.
- The Basil interpreter is written in Rust, and is open source.
- Basil can run on Windows, Linux, and macOS, or any device with a Rust compiler.

2) Two ways to run Basil programs

- CLI (Command Line): run a .basil file in your terminal.
- CGI templating (like PHP): mix HTML and Basil in one file to render web pages.

You can keep using the same Basil language in both situations.

2a) Three modes of operation

- Interactive: run a .basil file in your terminal.
- Web template: mix HTML and Basil in one file to render web pages.
- Command line: run a .basil file from the command line.

2b) Three ways to run Basil programs

- Run mode - run a .basil file in your terminal.
- Test mode - run a .basil file from the command line but skip over all inputs with mock keyboard input.
- Lex mode - run a .basil file from the command line and print out the tokens it reads.

3) The journey from source code to output

Here's the pipeline Basil uses whenever you run a .basil file:

1. Optional templating pass (if the file is used as a web template)
2. Lexer: turns characters → tokens (words, numbers, punctuation)
3. Parser: turns tokens → an AST (tree) that represents your program
4. Compiler: turns the AST → bytecode (a list of small instructions)
5. VM (Virtual Machine): executes the bytecode and produces output
6. Cache: saves the bytecode in a .basilx file to make next runs faster

A quick picture in words:

Your .basil text → tokens → AST → bytecode → VM runs it → prints results

4) Lexing (tokenizing)

The lexer reads characters and groups them into meaningful pieces called tokens:

- Numbers: 42, 3.14
- Strings: "Hello"
- Names: PRINT, LET, myVar
- Punctuation: ( ) , ;
- Operators: + - * / == != < <= > >=

Comments and spaces are skipped. Strings support common escapes like \n.

5) Parsing (building a syntax tree)

The parser looks at the token stream and builds a tree (AST) that captures structure and meaning. For example:

```
 LET a = 2 + 3 * 4;
PRINTLN a;
```

The parser knows this means “compute 3*4 first, then add 2,” then “print the result,” not just a flat list of words.

6) Compiling to bytecode

Basil compiles the AST into bytecode, which is a simple instruction list for the VM. Some instructions:

- Load a constant number or string
- Do math (+, -, *, /)
- Compare values (==, <, …)
- Jump for IFs and loops
- Make and access arrays
- Call functions
- Print or return

You don’t see this bytecode directly, but it’s what the VM executes.

7) Running on a stack-based Virtual Machine

The VM is a tiny, predictable computer that uses a stack (imagine a stack of plates). Most instructions:

- push values onto the stack,
- pop values off the stack,
- do an operation, and
- push a result back.

Example mental model for `PRINTLN 1 + 2;` :

- Push 1
- Push 2
- Add (pop 2 and 1, push 3)
- Println (pop 3 and print it)

That’s basically what the VM does, very fast.

8) Variables and types (simple rules)

- Basil values are dynamically typed at runtime (the VM knows what kind of value it is).
- Variable names can have optional suffixes to hint intent and mirror BASIC traditions:
  - `$` means string (e.g., `name$`)
  - `%` often used for integers (e.g., `count%`)
  - `@` is used for object references (e.g., `user@`)
- Common statements:
  - `LET x = 10;`
  - `PRINT "Hi";` or `PRINTLN x;`

The VM supports numbers (ints and floats), strings, arrays, functions, and objects.

9) Arrays (including multi-dimensional)

- Create with `DIM` .
- Arrays are zero-based, and the number you pass is the highest index (inclusive).
  - `DIM a(3);` creates indexes 0..3 (length 4).
- Access with parentheses: `a(0)` , `a(i)` . Assign with `LET a(2) = 99;`
- Multi-dimensional arrays use commas: `DIM grid(2,1);` → indexes 0..2 by 0..1 (3 × 2 elements)
- `LEN(a)` gives the total number of elements.

Example:

```
DIM x(3);
LET x(0) = 1.5;
LET x(3) = 2.5;
PRINTLN "x(0)=", x(0), ", x(3)=", x(3);
```

10) Control flow: IF, loops, functions

- IF/ELSE with optional blocks:

```
LET ans$ = "Y";
IF ans$ == "Y" THEN BEGIN
  PRINTLN "YES";
ELSE
  PRINTLN "NO";
END
```

- Loops: `WHILE … BEGIN … END`, `FOR i% = 0 TO 10 … NEXT`, and `FOR EACH` for iterating collections.
- Functions:

```
FUNC Add(a, b)
BEGIN
  RETURN a + b;
END
```

Call with `PRINTLN Add(2,3);`.

Under the hood, the compiler turns these into jumps and calls in bytecode, which the VM executes.

11) Objects and Classes (beginner-friendly view)

Basil lets you load a “class” from another .basil file and use it as an object. You’ll see variables with an `@` suffix used to hold object references.

Example (from the repo examples):

```
DIM user@ AS CLASS("my_class.basil");
PRINTLN "Initial:", user@.Description$;
LET user@.Description$ = "Updated description";
user@.AddUser("Erik");
PRINTLN "Count:", user@.CountMyUsers%();
```

What’s happening inside:

- The class file is compiled (like any Basil file) into bytecode and cached.
- The VM creates an object instance tied to that compiled code.
- Property gets/sets and method calls become VM bytecode instructions.

You can ship a class as a compiled `.basilx` without sharing the source.

12) Caching: .basilx files

To start your program quickly next time, Basil saves compiled bytecode next to your source in a `.basilx` file. On the next run, if nothing important changed (file size, modified time, format version, etc.), Basil skips re-compiling and runs the cached bytecode immediately.

This is automatic—you don’t have to do anything.

13) Error messages and line numbers

When something goes wrong at runtime (e.g., dividing by zero or calling a function with the wrong number of arguments), the VM reports an error that includes the source line number. That mapping comes from tiny “SetLine” markers the compiler inserts into the bytecode as it compiles your program.

14) Running Basil yourself

From a terminal (Windows PowerShell shown here), you can run examples like:

```
cargo run -q -p basilc -- run examples\hello.basil
```

If you use features that read the keyboard (like INPUT$) on Windows, run in a real terminal window—some IDE consoles don't support raw input well.

For CGI templates (web pages that mix HTML and Basil), see the examples and README for the exact delimiters (e.g., `<?basil … ?>` and the echo shorthand `<?= expr ?>`).

15) What to remember (the short version)

- Your code becomes tokens → a syntax tree → bytecode → the VM runs it.
- The VM is a stack machine: push values, pop values, do work.
- Arrays are `DIM`'d, zero-based, and multi-dimensional if you want.
- Classes let you make objects from other Basil files and call their methods.
- Compiled bytecode is cached (`.basilx`) for faster startup.

That's it! With this mental model, you'll be comfortable reading the deeper docs and exploring the examples in the repo.

16) Where to go next

- Read TECHNICAL.md for detailed internals (bytecode formats, opcodes).
- Read README.md and BASIL.md for language overview and examples.
- Explore `examples\` for runnable code, including arrays, classes, and control flow.