Scriptorium

Scriptorium is an original, experimental programming language inspired by the syntax of imperative languages (such as Python or C), equipped with its own parser, lexer, and interpreter based on ANTLR4. It supports variables, functions (with recursion), loops, and flow control. The project serves as a foundation for learning about programming language construction, syntactic and semantic analysis, and as an educational tool.

Installation and Configuration

System Requirements

- Windows
- Python 3.8+

Installing

- Copy **Scriptorium** repository into a folder on your machine
- Create venv and install requirements

```
# Create `venv` folder with virtual environment
$ py -3.8 -m venv venv
# Activate virtual environment
$ source venv/Scripts/activate
# Install all dependencies in virtual environment
$ pip install -r requirements.txt
```

- Add main repository folder path to `SCRIPTORIUM_HOME` environmental variable
- Add `%SCRIPTORIUM_HOME%` into `PATH` environmental variable

Running the Program

1. Create a source file in the Scriptorium language with the `.cr7` extension:

```
touch hello.cr7
```

2. Run the interpreter:

```
scriptorium hello.cr7
```

Example Session

```
$ scriptorium hello.cr7
CULPA: linea 1:18 - missing NL at 'numerus'
```

Ready! Now you can write programs in `.cr7` and run them using your own interpreter

Syntax and Basics

Scriptorium is an experimental programming language inspired by classical Latin. It uses full words as operators and control structures. It relies on indentation (similar to Python), supports functions, conditions, loops, and static typing.

Variables

Variables are declared using a type (`numerus`, `fractio`, `veritas`, `sententia`) and the keyword `esto`:

```
numerus a esto 10.
sententia x esto "salve".
```

Declaration without assignment:

```
fractio var.
```

Output

The keyword `scribere` is used for output. Multiple values can be combined using `et`:

```
scribere "Result: " et a et ".". // Output "Result: 2 ."
```

There is also a way of printing similar to string templating:

```
scribere "x = ${x}.". // Output: "x = 2."
```

Functions

Functions are defined using `munus`, specifying the return type, name, parameters, and action block:

```
numerus munus sum(numerus a et numerus b):
    reddere a adde b.
```

Function call:

```
numerus wynik esto sum(5 et 10).
```

📌 Conditions

The conditional statement `si`, optionally `aliter si`, ended with `aliter`:

```
si a aequalis 10:
    scribere "Ten.".
aliter si a minor quam 10:
    scribere "Less than ten.".
aliter:
    scribere "More than ten.".
```

📌 Loops

🔁 `repetere` — `for` loop

```
repetere i ex 1 ad 5:
    scribere i.
```

🔁 `dum` — `while` loop

```
dum a minor quam 100:
    a esto a adde 1.
```

📌 Operators

Operators in Scriptorium are full words:

Arithmetic

Symbolic	Scriptorium

Symbolic	Scriptorium
`+`	`adde`
`-`	`minue`
`*`	`multiplica`
`/`	`divide`
`//`	`totum`
`%`	`residuum`
`^`	`potentia`

Comparison

Symbolic	Scriptorium
`==`	`aequalis`
`!=`	`inaequale`
`<`	`minor quam`
`<=`	`minor aequalis`
`>`	`maior quam`
`>=`	`maior aequalis`

Logical

Symbolic	Scriptorium
`&&`	`etiam`
`||`	`aut`
`!`	`non`

📌 Special Instructions

- `reddere`: returns a value from a function
- `exire`: breaks a loop (`break`)
- `perge`: continues to the next iteration (`continue`)
- `rogare`: retrieves user input

📌 Comments

Single-line comments start with `//`:

```
// This is a comment
numerus x esto 10.
```

Data Structures

Scriptorium supports **statically typed** variables of four main data types and the null value `nihil`.

Data Types

Type	Keyword	Example	Description
Integer	`numerus`	`numerus a esto 10.`	Corresponds to `int`
Floating-point	`fractio`	`fractio x esto 3,14.`	Uses a comma instead of a dot
String	`sententia`	`sententia s esto "hej".`	Text type (`string`)
Boolean	`veritas`	`veritas v esto verum.`	`verum` or `falsum`
Null value	`nihil`	`nihil munus brak(): ...`	Equivalent to `None` / `void`

Value Assignment

Variables are assigned using `esto`:

```
numerus age esto 25.
fractio pi esto 3,14.
sententia message esto "salve!".
veritas active esto verum.
```

Dynamic Value, Static Type

The variable type is **determined at declaration** and cannot be changed. Incorrect example:

```
numerus x esto 5.
x esto "five".  // ✕ Error – type `sententia` incompatible with `numerus`
```

Input Handling

The `rogare` keyword can be used to read user input. The result is always a `sententia`.

```
sententia name esto rogare "Enter name: ".
```

🔀 Conversions (Explicit)

Type casting is available using `ut` keyword.

```
fractio x esto 4,5 ut numerus.
scribere x. // Result: "4.0"
```

🔐 Variable Scope

Variables are visible only in the **nearest block** (`if`, `for`, `munus`, etc.). Access to variables from higher levels follows the nested scope principle.

```
numerus global esto 5.

munus test():
    scribere global.  // ✔ Access to variable from higher scope
```

There is a possibility to declare variables with same names but in different scopes. There is a `parentes` keyword to access variable at parent scope.

```
numerus x esto 1.
nihil munus func():
    numerus x esto 2.
    scribere x. // Result: "2"
    scribere parentes x. // Result: "1"
    scribere parentes parentes x. // CULPA: linea 6:10 - cannot move 2 scope(s)
ago - no such scope defined
func().
```

🧠 Function Variable (`munus` as an object)

Functions In Scriptorium are represented as variables of type `Func` (internally). They can be called the same way as variables:

```
numerus munus sum(numerus a et numerus b):
    reddere a adde b.

numerus result esto sum(1 et 2).
```

🛑 Error Handling and Messages

Scriptorium is designed to make detecting and diagnosing syntactic and semantic errors as easy as possible. Errors are reported clearly and precisely, indicating the location of the issue in the code.

⚠ Types of Errors

1. **Syntax Errors (SyntaxError)**
   Detected during code analysis by the parser.
   Example messages:
   - `CULPA: linea 3:5 - syntax error at 'esto'`
     (e.g., incorrect use of a keyword or operator)
   - `CULPA: linea 5:10 - missing (")`
     (unclosed string)
   - `CULPA: linea 7:1 - incomplete or incorrect sentence`
     (e.g., missing element in a language construct)

2. **Semantic Errors**
   Detected during variable and function analysis while parsing (using `VariableListener`).
   Examples:
   - `CULPA: linea 4:0 - multiple variable or function "x" declaration (delcared in 2:0)`
     (duplicate declaration of a variable or function in the same scope)
   - `CULPA: linea 8:3 - no variable named "y"`
     (use of an undeclared variable)

3. **Runtime Errors**
   Currently, the language does not support advanced exception mechanisms, but errors such as division by zero or other invalid operations should be handled by the interpreter (as part of extensions).

💬 Error Message Format

```
CULPA: linea <line>:<column> - <error description>
```

- `CULPA` — prefix indicating an error (Latin for "fault").
- `linea <line>:<column>` — location of the error in the source file.
- `<error description>` — brief description of the issue.

⚙ Error Detection Mechanism

- The parser and lexer use a custom `ErrorListener` (an extension of `antlr4`), which captures and formats syntax errors.
- During parsing, the `VariableListener` builds a map of variables and functions, reporting declaration conflicts or references to nonexistent names.
- In `main.py`, all exceptions are caught and displayed, making debugging straightforward.

📝 Error Example

Source code:

```
numerus x esto 5.
numerus x esto 10.  // attempt at double declaration
```

Interpreter output:

```
CULPA: linea 2:0 - multiple variable or function "x" declaration (delcared in 1:0)
```

🔧 Tips

- Ensure the uniqueness of variable and function names within the same scope (function, loop, `if` block).
- Pay attention to properly ending statements with a period `.`.
- Carefully check the format and closure of strings.
- For syntax errors, note the token indicated by the parser that is causing the issue.

🏗️ Project Architecture

The Scriptorium language interpreter project is divided into several key components that work together to interpret source code in the Scriptorium language (with the `.cr7` extension).

🧩 Main Modules

Component	Description
`Scriptorium.g4`	Grammar file — contains parser and lexer rules, defines syntax and tokens.
`main.py`	Entry point of the interpreter. Loads the `.cr7` file, initializes the parser and visitor.
`visitor.py`	Implementation of the visitor, which traverses the syntax tree and interprets the code.
`var.py`	Definitions of variable classes (`Var`, `Func`, `ParamVar`) and mechanisms for scope and value management.
`VariableListener.py`	Listener that builds a map of variables and functions and detects declaration errors.
`requirements.txt`	List of required libraries and tools (e.g., `antlr4-python3-runtime`, `antlr4-tools`, `antlr-denter`).

🔄 Interpreter Workflow

1. **Code Loading**
   The interpreter loads the `.cr7` source code file.

2. **Tokenization and Parsing**

   - The lexer generates tokens based on the rules in `Scriptorium.g4`.
   - The parser builds an abstract syntax tree (AST) according to the grammar.

3. **Variable and Function Analysis (VariableListener)**

   - The listener traverses the tree and creates a map of variables and functions with their scopes.
   - It checks the correctness of declarations and reports errors.

4. **Code Interpretation (Visitor)**

   - The visitor traverses the AST, executing statements, evaluating expressions, calling functions, etc.
   - It manages the program state, e.g., variable values, function recursion levels.

5. **Displaying Results and Error Handling**

   - The interpreter outputs results of `scribere` (print) commands.
   - Errors are caught and reported in a readable form.

🗂 Directory Structure (Example)

```
/Scriptorium
│
├── main.py
├── visitor.py
├── var.py
├── VariableListener.py
├── Scriptorium.g4
├── requirements.txt
└── README.md
```

🛠 Tools and Libraries

- **ANTLR4** — generates lexers and parsers from the `.g4` file.
- **antlr4-python3-runtime** — runtime for parser support in Python.
- **antlr-denter** — lexer extension for handling indentation (indent/dedent) in the style of Python.
- **Python 3.8** — runtime environment.

FAQ — Frequently Asked Questions

1. What file extension does the Scriptorium language use?

Source files for the Scriptorium language have the `.cr7` extension.

2. How do I run a program written in Scriptorium?

After creating a `.cr7` file, run the interpreter with the command:

```
scriptorium program.cr7
```

3. What should I do if I encounter a syntax error starting with "CULPA"?

This is a customized syntax error message in our language.
Check the specified line and column in the file to identify the issue. For example:

```
CULPA: linea 5:10 - syntax error at 'ad'
```

indicates a syntax error on line 5, column 10, related to the token `'ad'`.

4. Does Scriptorium support object-oriented programming or modules?

Currently, Scriptorium is a procedural language without support for OOP, modules, or external libraries.

5. How do I handle variables and data types?

You declare variables by specifying the type before the name, e.g.:

```
numerus x.
```

or with a definition:

```
numerus x esto 5.
```

Available types are: `numerus` (int), `fractio` (float), `sententia` (string), `veritas` (bool), and `nihil` (null).

6. How do I define and call functions?

Functions are defined using the keyword `munus`, e.g.:

```
numerus munus sum(numerus a et numerus b):
    scribere a adde b.
```

Call functions using their name and parentheses:

```
sum(3 et 5).
```

7. How do loops and conditional statements work?

- `dum` loop (while)
- `repetere` loop (for)
- Conditional statements: `si`, `aliter si`, `aliter`

Example:

```
si x maior quam 0:
    scribere "Positive".
aliter:
    scribere "Non-positive".
```

8. Is there support for error handling?

Yes, the interpreter catches syntax and semantic errors, which are reported with precise code locations.

9. How can I add new features or report a bug?

Please submit an issue on GitHub or contact us via email (address in the project documentation).

If you have other questions, check the documentation or contact us!

Summary

Scriptorium is a simple, procedural programming language created for educational purposes and rapid prototyping. With its clear, Latin-inspired syntax and precise error handling, users can quickly identify and fix issues in their code.

The project is based on ANTLR4 and Python 3.8, ensuring easy extensibility and integration.
The introduction of basic data types, functions, loops, and conditions enables the creation of clear and understandable programs.

We hope this documentation will make it easier for you to start working with Scriptorium and make programming a pleasure.

We invite you to experiment and develop the language!