Scriptorium Interpreter Implementation Report

In this document you can find details on how **Scriptorium** interpreter was created.

Scriptorium is a compact, educational programming language designed to demonstrate an end-to-end interpreter pipeline built with **ANTLR 4** and a classic tree-walk evaluator. Its syntax blends Python-style indentation with Latin keywords (e.g. `scribere` for print, `si … aliter` for if/else).

Table of Contents

1. Build & run

Scriptorium has a lexer, parser, visitor and listener auto-generated from Antlr4. To generate new versions use commands:

```
# regenerate with antlr4 after grammar changes
antlr4 -Dlanguage=Python3 ScriptoriumLexer.g4
antlr4 -Dlanguage=Python3 Scriptorium.g4

# execute a program
scriptorium 03-04-2025/program.cr7
```

2. Execution model (two-pass)

The interpreter runs in two distinct passes:

1. **Declaration pass** — in this run `var_map` dictionary is populated with variable, parameter, and function signatures. This dictionary helps keep track of all variables in use.

   > This part is managed in `Scriptorium/ScriptoriumVariableListener.py`.

2. **Evaluation pass** — the visitor updates those `var_map` entries with runtime values, enabling forward references and precise scope resolution.

   > This part is managed in `Scriptorium/ScriptoriumVisitor.py`.

3. Variable Management

The most important part of Scriptorium interpreter is a **variable management system**. There is a plenty of methods/classes created to help with process of defining, writing and reading variables or function invocation.

3.1. Scopes

Every variable and function in the code must be inside some kind of "scope". We decided to define "scope" in similar way like python does - **every indent level is a different "scope"**. But indents cannot be used without a special instruction. So new scope starts with every instruction that ends with `:` and creates indent block for example:

```
si verum:
  // NEW SCOPE HERE
  numerus x esto 2.

// You cannot access `x` from here
```

3.1.1. List of scope tokens

Full list of instructions that create new scope:

- `IfBlockContext` - If block scope
- `IfElseBlockContext` - If else block scope
- `ElseBlockContext` - Else block scope
- `ForLoopContext` - For loop scope
- `WhileLoopContext` - While loop scope
- `FunctionDeclarationContext` - Function scope
- `StartContext` - Global scope

So if there is a `ForLoopContext` token, it is automatically a scope for all variables and functions declared inside its action block.

3.2. Use of `var_map`

To keep track of every variable in Scriptorium we use a dictionary.

3.2.1. Structure of `var_map`

```
var_map: Dict[ctx, Dict[str, Var]]
```

** Where `ctx` is a type of node context object*

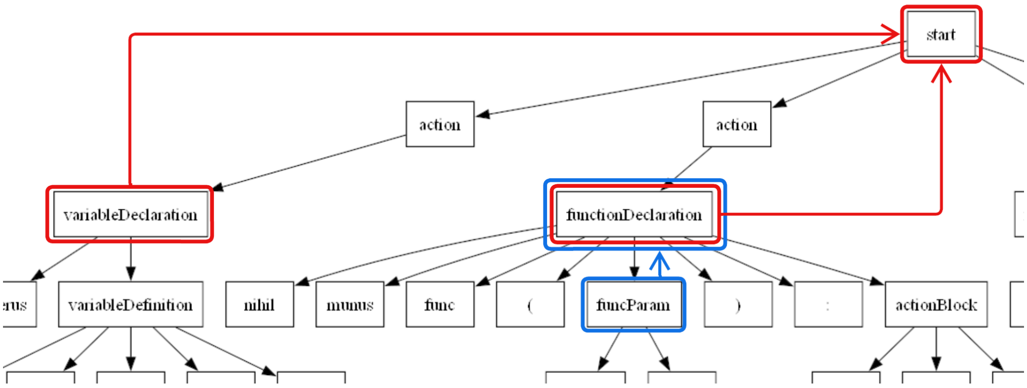
We use **"scope tokens"** as keys in `var_map` dictionary.
Then there is another dictionary where keys are variable names and values are variable/function objects.

```
"var_map" dict
├── KEY: "scope token"
└── VALUE: another dict
    ├── KEY: "name of variable/function"
    └── VALUE: variable/function object
```

3.2.2. Example

```
numerus a esto 5.
nihil munus func(numerus x):
    scribere x.
```

You can see the **"declaration"** nodes of variables, params and functions connected to their **"scope tokens"** nodes would look like this:



And thats why result `var_map` looks like this:

```
{
  <ScriptoriumParser.StartContext object at 0x000001AF6842AC80>: {
    'a': <Var: typeId=23, value=[]>,
    'func': <Func: typeId=5, functionCtx=
<ScriptoriumParser.FunctionDeclarationContext object at 0x000001AF684D5970>,
returnType=29>
  },
  <ScriptoriumParser.FunctionDeclarationContext object at 0x000001AF684D5970>: {
    'x': <ParamVar: typeId=23, value=[]>
  }
}
```

3.3. Storing stack of variable values

Every variable in Scriptorium has it's own stack. We obtain a value from the stack based on recursion level:

```
current_value = some_var.value[recursion_level]
```

3.3.1. How it works

3.3.1.1. Writing

When writing new value to the stack there are only two options

1. There is a value for specified `recursion_level`:

   Value gets overwritten

   ```
   recursion_level = 2
   # var.value -> [0, 1, 2]
   var.change_or_append_value(recursion_level, 10)
   # var.value -> [0, 1, 10]
   ```

2. There is no value for specified `recursion_level`:

   Value is appended at specified `index = recursion_level`. Values at `indices < recursion_level` are set to `None`

   ```
   recursion_level = 2
   # var.value -> [15]
   var.change_or_append_value(recursion_level, 20)
   # var.value -> [15, None, 10]
   ```

3.3.1.2. Reading

If given variable value stack is shorter than `recursion_level` or value on given index is `None`, then there is exception raised:

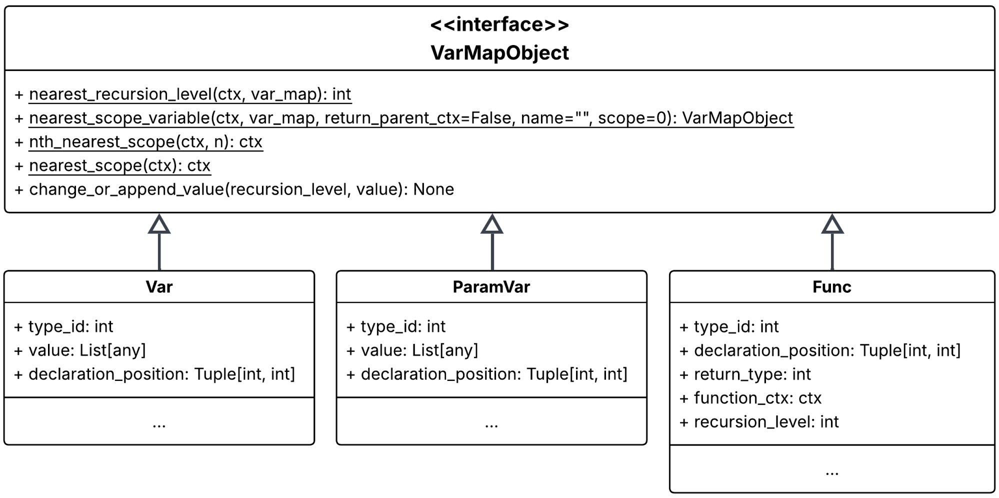
`CULPA: linea xx:yy - variable named "a" is not yet defined`

Otherwise value of `Var.value[recursion_level]` is returned

3.4. Variable classes

Variable system is implemented in such a way that it automatically searches for right scope and finds if variable read instruction is in recurrence run - if it is, it checks what is the `recursion_level` and grabs right value from `value` stack of a variable.

To manage all of that there are following classes implemented:



3.4.1. `Var` class

This is a default variable class. It's used when declaring new variable in any scope and for iterator variable in `for` loop.

3.4.1.1. Attributes

- `type_id` - id of **Scriptorium** type
- `value` - list of variable values in a specific recurrence run
- `declaration_position` - tuple (line, column) for better errors when double declaration occurs

3.4.2. `ParamVar` class

This is a variable class used for function parameters. It's used to distinguish variables defined inside function from function parameters.

3.4.2.1. Attributes

- `type_id` - id of **Scriptorium** type
- `value` - list of variable values in a specific recurrence run
- `declaration_position` - tuple (line, column) for better errors when double declaration occurs

3.4.3. `Func` class

In Scriptorium function is also a `var_map` object. It helps to track right recursion level and has a information about function node context

3.4.3.1. Attributes

- `type_id = 5` - id of **Scriptorium** function type
- `declaration_position` - tuple (line, column) for better errors when double declaration occurs
- `return_type` - id of **Scriptorium** type of value returned from the function
- `function_ctx` - context of function node, used for function invocation
- `recursion_level` - used to determine which value to get from value stacks of other variables inside function scope (for recursion)

3.5. Finding variables

When we need to read value of a variable we use a `VarMapObject.nearest_scope_variable()` static method. It searches through parse tree until it finds a "**scope token**" where there is a variable with specified name or raise an error when there is no variable with that name.

3.5.1. `VarMapObject.nearest_scope_variable()` method

This method finds nearest variable with specified name.

```
def nearest_scope_variable(ctx, var_map, return_parent_ctx=False, name="",
scope=0):
  """
  Searches for the nearest variable definition in the current or parent scopes.

  Args:
      ctx: The current context object, typically representing a scope or block.
      var_map (dict): A mapping of contexts to their defined variables.
                      Each key is a context, and each value is a dictionary of
variable names to their Var objects.
```

```
        return_parent_ctx (bool, optional): If True, returns a tuple containing the
variable value and its parent context.
                                          Defaults to False.
        name (str, optional): The name of the variable to search for. If empty, the
variable name is extracted from `ctx`.
                            Defaults to an empty string.
        scope (int, optional): The number of scopes to go back for the search. Used
for error reporting purposes.
                              Defaults to 0.

    Returns:
        The Var object of the nearest variable matching the name in the current or
parent scopes.
        If `return_parent_ctx` is True, returns a tuple (variable_object,
parent_context).

    Raises:
        Exception: If the variable is not found in the current or parent scopes, an
exception is raised with details
                    about the line and column of the context and the scope depth.
    """
```

Why is parent context useful? Because when searching for function recursion level you can start searching from variable parent context to avoid getting wrong recursion level value.

3.6. Finding scopes

When searching for scopes we look for **"scope tokens"**. There are 2 methods that help us manage scopes throughout development process.

3.6.1. `VarMapObject.nearest_scope()` method

Function that returns a context of nearest **"scope token"**.

```
def nearest_scope(ctx):
    """
    Determines the nearest enclosing scope for a given context.
    This function traverses the parent contexts of the given `ctx` object
    until it finds a context that matches one of the predefined scope types.
    If no matching scope is found, it returns the topmost parent context.
    Args:
        ctx: The current context object.
    Returns:
        The nearest enclosing scope context object that matches one of the
        predefined scope types.
    """
```

3.6.2. `VarMapObject.nth_nearest_scope()` method

Function used with `parentes` keyword. It helps wind variables above certain number of scopes.

```
def nth_nearest_scope(ctx, n):
  """
  Retrieves the nth nearest scope from the given context.

  This function navigates up the scope hierarchy starting from the provided
  context (`ctx`) to find the nth nearest scope. If the requested scope level
  exceeds the available parent scopes, an exception is raised.

  Args:
      ctx: The current context object.
      n (int): The number of scopes to move up from the current context.

  Returns:
      The context object representing the nth nearest scope.

  Raises:
      Exception: If the requested scope level exceeds the available parent
                 scopes, an exception is raised.
  """
```

3.7. Calculating current `recursion_level`

When function is invoked, then a `recursion_level` attribute on `Func` object that holds function context is incremented. When function returns - attribute is decremented. While trying to access any variable we use `VarMapObject.nearest_recursion_level()` static method to search for closest function scope (or start scope for global variables) and get it's `recursion_level` value.

3.7.1. `VarMapObject.nearest_recursion_level()` method

```
def nearest_recursion_level(ctx, var_map):
  """
  Determines the recursion level of the nearest function declaration context.

  This function moves up through the parent contexts of the given `ctx` to locate
the nearest
  function declaration context. Once found, it retrieves the recursion level of
the
  corresponding function variable from the `var_map`.

  Args:
      ctx: The current context object, typically an instance of a parser context.
      var_map: A dictionary mapping scope contexts to variable mappings. Each
variable
            mapping contains function variables with their associated recursion
levels.var_map (dict): A mapping of contexts to their defined variables.
                Each key is a context, and each value is a dictionary of variable
names to their Var objects.
```

```
    Returns:
        int: The recursion level of the nearest function declaration context. If no
function
          declaration context is found, returns 0.
    """
```

4. ANTLR Grammar

4.1. Indents

For indentation mechanic we used a Antlr **addon**: `antlr-denter` [LINK].

We followed instructions and implemented it in our language. It helps us keep track of indentation level and check if it is correct. Antlr rules that use indentation looks like this:

```
block: INDENT statement+ DEDENT
```

4.2. String templating

For string templating we used Antler lexer **modes**. There are 3 modes:

1. `DEFAULT`
2. `IN_STRING` - anything between `"` characters
3. `IN_INTERP` - while in `IN_STRING` mode, anything between `${` and `}`

Modes are only used for string templating because we decided to use them at the and of the project when all the rest was done. Using modes earlier would probably be a good idea.

4.3. Possible value types

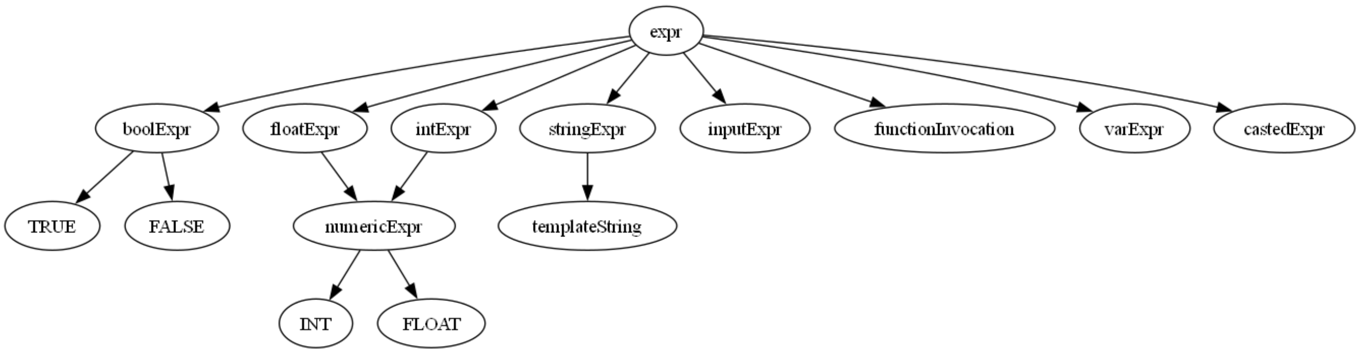
Every value that exists in Scriptorium is a **expr**. There are many possible operations depending on type of variables and/or constants.

Expr can be one of:

1. Standard types:
    - `veritas` - Bool
    - `fractio` - Float
    - `numerus` - Int
    - `sententia` - String
2. Value of other operations:
    - `rogare` - User input
    - `ut` - Casting result
    - Function invocation result
    - Other variable assignment

4.3.1. Diagram of `expr` possible paths

This diagram show possible children of **"expr"** node.



4.3.2. Type casting

4.3.2.1. Automatic type casting

There are few moments when `expr` value is automatically casted to the right type. Those scenarios are:

1. **Variable definition**

   Based on metadata stored in `Var.type_id`

2. **Function invocation**

   Based on metadata stored in `ParamVar.type_id`

3. **Returning from function**

   Based on metadata stored in `Func.return_type`

* *There is also a **numericExpr** - merge of floatExpr and intExpr, that allows user to make operations on both integers and float numbers without casting.*

4.3.2.2. Manual type casting

In other scenarios where there is no metadata found on what is the target type of value. For manual casting there is a `ut` keyword.

```
scribere 2 adde "2" ut numerus. // Result: "4.0"
scribere 2 ut sententia adde "2". // Result: "22"
scribere 2 adde "2". // CULPA: linea 2:19 - syntax error at "2".
```