Documentation for the Forth Interpreter Project

December 20, 2024

Contents

1 Introduction **2**

2 Overall Architecture **2**

2.1 Class Inheritance 2

2.2 Function Call Order 2

3 Exception System **3**

3.1 Custom Exceptions 3

3.2 Error Handling Workflow 3

4 Author Notes **3**

1 Introduction

The Forth Interpreter is designed to emulate the behavior of a stack-based programming language, providing efficient execution of commands and stack manipulations. The interpreter supports a variety of operations, including arithmetic, logical operations, and user-defined functions. This document provides an overview of its architecture, exception handling mechanisms, and additional notes for developers.

2 Overall Architecture

The Forth Interpreter is structured into several interacting components. The key classes include:

2.1 Class Inheritance

The core functionality of the interpreter revolves around the abstract `Executable` class, which defines a uniform interface for all executable entities. The inheritance structure is as follows:

- **Executable**: The base class for all executable entities. It defines the `Execute` method, which all derived classes must implement.
  - **VariableCreation**: Represents the creation of variables in the environment. It stores the variable's name, size, and type, and adds it to the environment during execution.
  - **Codeblock**: Encapsulates a sequence of statements (a list of `Executable` objects) and executes them sequentially.
  - **While**: Represents a `while` loop, containing a condition (an `Executable`) and a body (another `Executable`).
  - **For**: Represents a `for` loop structure. It includes a body (an `Executable`) to execute in each iteration.
  - **If**: Implements an `if-else` structure with two parts: `if_part` and `else_part`, both of which are `Executable` objects.
  - **Switch**: Represents a `switch-case` construct. It maps case values to corresponding `Executable` objects.
  - **Operator**: Encapsulates operations (e.g., arithmetic, logical). Operators are identified by their text representation and invoke predefined functions stored in a static map.

2.2 Function Call Order

1. The `Parser` tokenizes the input string into `Lexeme` objects.
2. Each `Lexeme` is passed to the `Grammatical analyzer`, which builds `Executable` tree representation of the program.
3. The `Executable` derived classes invoke functions on the `Environment` to execute the operation (e.g., arithmetic, stack manipulation).

3 Exception System

The interpreter uses a custom exception handling mechanism to ensure robust error management. Key features include:

3.1 Custom Exceptions

- **SyntaxError**: Thrown when invalid input is encountered during parsing.
- **RuntimeError**: Thrown during grammatical analyzing when encountering wrong syntax or when undefined behaviour occurs(user pops empty stack).

3.2 Error Handling Workflow

1. Errors are detected in `grammatical analyzer` or `operator` methods.
2. An exception is thrown, containing a descriptive error message and context information.
3. The exception is caught in the main interpreter loop, which reports the error to the user.

4 Author Notes

- The project demonstrates advanced use of C++ features such as smart pointers (e.g., `std::unique_ptr`) and standard containers (e.g., `std::map`, `std::vector`).
- The implementation of the Trie structure in the `Parser` is primarily for demonstration purposes and could be optimized further for production use.
- The interpreter is extensible, allowing developers to add new operators and customize the environment with minimal changes to the existing codebase.