YARTBML: A Modern-Take on the SMoL (Standard Model of Languages)

Abstract

Inspired by Thorsten Ball's seminal work, "Writing An Interpreter In Go", we aim to extend the foundational concepts introduced by Ball. Our initial intention is to reimplement his base language, but draw some inspiration from a few forks that exist out in the wild, such as Tau (module imports, C interoperability, CLI Tool). Moreover, we aim to create a language with solid developer experience by attempting to introduce a syntax highlighter (and possible Language Server) within the most common editors such as Neovim, VSCode, and Sublime.

The initial implementation will cover the basis of the SMoL (Standard Model of Languages) definition, and then extend it with modern developer tooling to help us understand all the components that make up a modern successful language such as Rust, Go, and JavaScript. This document details the design choices, implementation strategies, and technical specifications of our interpreter, demonstrating its capabilities and the rationale behind its development.

1. Introduction

The creation of a programming language interpreter is a profound exercise in understanding computation and language design. Following the guidance of Thorsten Ball, we embark on a journey to not only replicate the foundational aspects of an interpreter as presented in "Writing An Interpreter In Go" but also to expand its capabilities. The goal is to create an interpreter that supports a rich set of features catering to modern programming paradigms.

We aim to support syntax-highlighting and possibly a language server if time permits, alongside a CLI tool to process entire files of YARTBML code.

2. Language Features

2.1 Supported Data Types

Our interpreter supports several data types, crucial for a wide range of programming needs:

- **Integers**: For numerical computations.
- **Booleans**: To support logical operations.
- **Strings**: For text manipulation.
- **Arrays**: To hold collections of values.
- **Hash Maps**: For key-value pairs storage, enabling complex data structures.

2.2 REPL (Read-Eval-Print-Loop)

A REPL environment is implemented to allow interactive programming and immediate feedback. This feature is vital for testing code snippets, debugging, and educational purposes.

2.3 Arithmetic Expressions

The interpreter can evaluate complex arithmetic expressions involving integers, supporting operations such as addition, subtraction, multiplication, and division.

2.4 Let Statements

Variable bindings are facilitated through "let" statements, allowing users to assign and reference values in their programs.

2.5 First-Class and Higher-Order Functions

Functions in our interpreter are treated as first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables. This supports the creation of higher-order functions, enhancing the language's flexibility and expressiveness.

2.6 Built-In Functions

A set of built-in functions is provided out-of-the-box, offering common utilities for string manipulation, numerical calculations, and data structure handling.

2.7 Recursion

Our interpreter supports recursion, enabling functions to call themselves. This is essential for solving problems that naturally fit recursive solutions, like factorial calculation and tree traversal.

2.8 Closures

Closures are implemented, allowing functions to capture and carry with them their surrounding state. This feature is crucial for creating functional programming techniques.

2.9 CLI Tool

The CLI Tool will allow for our language to be executed via files instead of line-by-line with the REPL.

3. Implementation

3.1 Choice of Go

Following Thorsten Ball's lead, Go was chosen as the implementation language due to its simplicity, efficiency, and excellent support for concurrency. Go's garbage collection and straightforward syntax make it an ideal choice for building complex interpreters.

3.2 Parsing and Lexical Analysis

The interpreter uses a handwritten lexer and parser, transforming source code into an abstract syntax tree (AST). This approach provides fine-grained control over syntax and error handling.

3.3 Evaluation Engine

At the heart of the interpreter is an evaluation engine that traverses the AST, performing computations and function calls as dictated by the source code. This engine is designed with extensibility in mind, allowing for the easy addition of new features and data types.

4. Advanced Features and Extensions

Building upon the foundational features, our interpreter includes several advanced mechanisms to enhance its capabilities further:

- **Error Handling**: Robust error handling mechanisms are implemented, ensuring meaningful feedback is provided for syntax and runtime errors.
- **Memory Management**: Efficient memory management techniques are utilized to manage the lifecycle of objects and functions, preventing memory leaks.
- **Optimization**: Various optimizations are applied to improve the performance of arithmetic operations and function calls, ensuring the interpreter remains responsive, even for complex programs.

5. Conclusion

The development of our interpreter, inspired by “Writing An Interpreter In Go,” will result in a powerful and flexible tool that extends the foundational concepts introduced by Thorsten Ball. By supporting a wide range of features such as multiple data types, a REPL, arithmetic expressions, variable bindings, first-class and higher-order functions, built-in functions, recursion, and closures, we will create an interpreter that is both educational and practical for real-world programming. In addition, our developer tooling will be a large part of why our language adoption will be much smoother to develop complex artifacts with, such as the ability to automate build processes via the CLI tool. Lastly, we will highlight how important it is to have a solid developer experience that works consistently across multiple IDEs by showcasing how easy it is to manually read our code with just proper syntax highlighting and a possible LSP.

6. Future Work

Moving forward, we aim to further expand the interpreter’s capabilities, incorporating static typing, concurrency support, and enhanced library functions. In addition, we also aim to provide better error handling and reporting throughout the project, and hope to eventually compile the language instead of interpreting the language via our own virtual machine. The goal is to evolve the interpreter into a comprehensive platform that not only serves as an educational tool but also as a viable option for certain types of software development projects.

References

Ball, Thorsten. “Writing An Interpreter In Go.” This work serves as the primary inspiration for our project, providing a solid foundation in interpreter design and implementation.

Santamaria, Nicolo. “tau: A functional interpreted programming language with a minimalistic design.”, (2020). https://github.com/NicoNex/tau.