Winter Study Report - Crafting Interpreters

Markus Feng

Due: 2020-01-31

Introduction

Over this winter study, I followed the book "Crafting Interpreters" by Robert Nystrom[1] and worked on implementing two programming language interpreters for the programming language Lox, as described by the book.

The first interpreter I worked on was a tree-walk interpreter for Lox, written in Java. The way this interpreter worked was that it uses a scanner (lexer) and a parser to convert raw code to an abstract syntax tree (AST), and the interpreter traverses this AST in order to run the program.

The second interpreter I worked on was a bytecode virtual machine for Lox, written in C. This interpreter is implemented with a stack-based bytecode instruction set, so morally it's more similar to implementing a compiler for a bytecode instruction set we devised ourselves.

Note that as of right now, although the tree-walk interpreter is done, the bytecode virtual machine for Lox is not yet complete, as I am still in the middle of working on this. My current progress for the bytecode virtual machine can be found here[2]. This implementation uses Emscripten[3], a C-to-Webassembly transpiler, to compile my bytecode virtual machine to the web. All of the code that I wrote for this winter study can be found on my Github repository here[4].

This paper will discuss my process of implementing these interpreters and what I learned throughout the winter study period. I will further elaborate on the implentation details specific to the tree-walk interpreter and the bytecode virtual machine later on in their individual sections.

Background

At the beginning of Winter Study, I researched several potential projects and topics for me to pursue during the Winter Study period, before deciding on following the Crafting Interpreters book.

Secure Programming Language

One of the first ideas that I had for winter study was to build a type-safe programming language that is secure for remote and distributed systems to use. Doing research on this subject, I found that Google created a remote procedure call (RPC) platform called Protocol Buffers (ProtoBuf) [5]. How this scheme worked is that the user would write a platform-independent `.proto` file, which specifies what data types will be transmitted. ProtoBuf uses this information to automatically generate code in any supported language, so it can help enable type-safe . Furthermore, using external extensions, ProtoBuf can automatically validate that the input provided is valid, since much of validation code is redundant, and the `.proto` file contains much of the information needed to perform validation.

During this research, I also found that because of several flaws in the original ProtoBuf implementation, several competitors have sprung up, namely Cap'n Proto[6] and Flatbuffers (also developed by Google) [7].

[1] Crafting Interpreters book online: `http://craftinginterpreters.com/`
[2] Web demo: `https://markusfeng.com/projects/craftinginterpreters/`
[3] Emscripten: `https://emscripten.org/`
[4] Github repo: `https://github.com/nomoid/CraftingInterpretersStudy/`
[5] ProtoBuf: `https://developers.google.com/protocol-buffers`
[6] Cap'n Proto: `https://capnproto.org/`
[7] Flatbuffers: `https://google.github.io/flatbuffers/`

These competitors have some extra features and fixes some of the issues present in ProtoBuf, but the overall idea is the same.

One topic that I wanted to pursue was to design a programming language that specifically facilitated towards safe remote data handling, such as potentially being designed specifically around supporting one of the aforementioned protocols, with language features, type systems, syntax, and semantics that specifically benefitted from the protocol usage.

However, I realized that simply making a robust programming language was a massive undertaking for the winter study period, particularly because I my only previous experience with implementing a programming language was for my final project for my Programming Languages course, where I came up with the methods mostly on my own, rather than researching about existing programming language implementation methods and learning formally the ways modern languages are written. Therefore, I started researching about and looking into ways I can learn about creating programming languages, and perhaps turn a language I create into one that cares specifically about safe remote data transfer.

GraalVM Truffle Language

When I looked more deeply into modern day programming language technologies, one particular platform stood out to me. GraalVM[8], developed by Oracle originally as a Java Virtual Machine (JVM) alternative, has now developed into a platform with a wide variety of features.

One of the features is the idea of a common runtime for many languages that live on Graal, integrated in a way so that build tools, debugging environments, and so on are all shared among all of the languages, so if someone were to develop a new language on Graal, it would have the advantage of existing supporting tools.

This polyglot (multiple languages) feature allows a program to seamlessly integrate multiple programming languages together. For example, right now Graal has (incomplete) support not only the JVM languages such as Java, Scala, and Kotlin, but also other popular languages including Javascript, Ruby, Python, and R [9].

Furthermore, GraalVM comes with an interpreter-building framework called Truffle[10]. What Truffle does is that it abstracts away many of the necessary components for implementing your own language, by combining all of the repeated machinery and making it easy to make your own modifications.

However, one of the big downsides of Truffle is that overall, it is a very new project, and poorly documented, making it difficult to figure out how the different parts of it work and interact. When I tried to get started, I found myself stuck, not knowing how to proceed, especially with the lack of resources for less advanced users. As a short-term user, it felt like Truffle was more geared towards internal Oracle users rather than the general public, with some of the features behind internal-only or Enterprise Edition exclusive feature sets.

In addition, I did not find many externally-built projects that use GraalVM, so I feel that at the current point in time, this technology is not suitable for general use. Therefore, after spending a few days on pursuing the route of making a language with GraalVM, I decided to look for another option for my winter study project.

LLVM Frontend

The next project I looked as a potential project to pursue is to write an LLVM frontend for a custom programming language. LLVM [11] is a set of compiler infrastructure technologies, with one of the most important parts of LLVM being LLVM Core, a set of libraries that can turn LLVM Intermediate Representation (IR), a machine-code-like language, into optimized platform-specific machine code. Notably, the Clang C compiler, is built on top of the LLVM infrastructure, and competes with other C compilers such as the GNU C Compiler (GCC) as one of the industry standard C compilers.

[8] GraalVM: `https://www.graalvm.org/`
[9] GraalVM Supported languages: `https://www.graalvm.org/docs/`
[10] Truffle: `https://github.com/oracle/graal/tree/master/truffle`
[11] LLVM: `https://llvm.org/`

Therefore, if a programming language compiler generates LLVM IR, it can be compiled onto any platform with LLVM support, without the compiler having to worrying about the specific instruction set of the underlying machine. Thus, many programming languages write what is called a LLVM frontend, including the compilers of Rust, Swift, and Kotlin, and have LLVM do the heavy lifting of ensuring that the resulting executable will work on any LLVM-supported platform. In addition, various LLVM backends have been created that target platforms other than native code, including Emscripten, as mentioned earlier, which turns LLVM IR into Webassembly, and GraalVM, which supports LLVM languages on the virtual machine.

One aspect of LLVM that is excellent is its documentation. LLVM includes a tutorial [12] that explains how to implement a language with LLVM. Furthermore, there is ample documentation as to how LLVM works, along with many existing high-profile projects that currently use LLVM to some extent. Therefore, following the LLVM tutorial seems like a much better choice for a winter study project. Ultimately, I decided to following the tutorial for the Crafting Interpreters book instead, but the LLVM project would not have been a bad idea either.

Crafting Interpreters Book

Process

The Lox Language

The Tree-Walk Interpreter

The Bytecode Virtual Machine

Future Work

Conclusion

[12] LLVM Tutorial: `https://llvm.org/docs/tutorial/index.html`