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.

[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`

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]. 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),

[6]Cap'n Proto: `https://capnproto.org/`
[7]Flatbuffers: `https://google.github.io/flatbuffers/`
[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/`

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.

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

After invesigating my other options, I finally settled on following the "Crafting Interpreters" book by Robert Nystrom, as my project for winter study. This book is a free textbook available online, divided into chapters, where each chapter describes one aspect of implementing the Java interpreter or the C virtual machine.

I felt that this textbook is reputable for learning about programming languages, because the author wrote the best-selling book "Game Programming Patterns", and has also created a number of programming languages in his free time. Right now, the author works at Google on the Dart programming language.

Furthermore, from reading over the very beginning of the book, I thought that the book was suitable for a winter study course, because the content feels very approachable and not very dense, with the author writing with the mindset of practical over purity. Therefore, I ended up working through this book for the remainder of winter study.

Process

Once I settled on following the Crafting Interpreters book, my goal for winter study was to complete as much of the book as possible. If I had extra time remaining afterwards, I would try to add additional features to the language, to make it my "own", and to experiment with features that may not be present in other languages.

The way the book is laid out, I would work on one or two chapters per day. Working through a chapter involed implementing a feature of the Lox language for that chapter, for which the full code is provided by the book. I made sure to understand each bit of code before writing it out myself. At the end of each chapter, the book provides several exercises, such as improving one of the features in the chapter, or implementing a more advanced features on top of that. I attempted as many of the exercises as I could, making the language I created in the end a more extensive version of Lox than the minimal one presented by the book. In doing so, I gained a deeper understanding of the implementation from the chapter, since I wrote my own original code on top of it.

The Lox Language

The Lox language was defined as a loose specification in one of the first chapters of the crafting interpreters book. Lox is a high-level, dynamically typed, minimal scripting language, with a clean C-inspired syntax.

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

Here is an example Hello World program in Lox:

```
fun main() {
  // Prints "Hello, world!" to the console
  print "Hello, world!";
}
```

Since Lox is dynamically typed, its variables can store values of any type, and there is no compile-time type checking. Lox also has automatic memory management with garbage collection.

Lox contains the following data types:

- Booleans
  - Contains either the value `true` or `false`
- Numbers
  - In Lox, all numbers are double-precision floating point numbers.
  - As an extension, I separated Float and Int into two datatypes, with Float being a double-precision floating point number and Int being a 64-bit signed integer.
- Strings
  - Lox contains Strings to store groups of character.
- Nil
  - Nil represents null or nothing in Lox. By default, uninitialized expessions are nil, and returning from a function without a value returns nil.

Lox contains the following expressions, which work similar to the way they do in other languages:

- Arithmetic expressions (`+`, `-`, `*`, `/`)
- Comparison expressions (`<`, `<=`, `>`, `>=`, `==`, `!=`)
- Logical operators (`!`, `and`, `or`)
- Grouping (`()`)

Lox contains the following statements, which also work similar to the way they do in other languages:

- Print (`print`), which prints out the following expression to the console (similar to `println` in other langauges)
- Control flow statements (`if`, `else`, `for`, `while`)
  - Specifically, Lox supports the C-style 3-part `for` loop, with an initializer, a terminator, and an increment.

Lox contains variables with lexical scoping and blocks.

Lox contains functions that work similarly to functions in other languages. Specifically, Lox funcions are first-class, int that they themselves can are objects in Lox.

Lox also contains object orientation, similar to object orientation in Java, with classes, methods, constructors, inheritance.

Finally, the language definition for Lox has support for a standard library, but not much work in the book is put into implementing one.

This is essentially what the book defines as the Lox language. To make Lox a practical language for use, other features should probably be added, including a much more extensive standard library (with features

including math, input/output, graphics, networking, etc.), ways to import code from other Lox files and libraries, a foreign function interface, documentation describing how things work, development tools such as a debugger or an IDE, etc. These will not be the focus of this winter study, but I may try to add some of these if I further pursue making this language or another language, or if I have extra time at some point.

The Tree-Walk Interpreter

The tree-walk interpreter is the first of two interpreters that I implemented. This implementation was written in Java, since it is a popular language with high-level support such as objects and garbage collection.

This interpreter is divided into several phases. The first phase of the interpreter is the scanner, also commonly known as the lexer. The way this phase works is that it scans the input (either a File or a REPL input) as a String and converts it to a list of tokens. A token is a logical unit of the program's text. For example, a token can be an operator (e.g. `+`), a grouping symbol (e.g. `{`), a keyword (e.g. `if`), an identifier (e.g. `i`), or a literal (e.g. `16`). The token has fields holding the type of the token, the location of the token in the program (for error message purposes), and the value of the token if it's a literal.

After the scanner is done scanning its tokens, the tokens are used as the input to the parser. The parser used in this implementation of Lox is a recursive descent parser. This parser is considered a top-down parser, because it starts from the outermost grammar rule and works its way inwards. The parser translates the grammar's rules into imperative code, such that each rule becomes a function. The parser generates the Abstract Syntax Tree (AST), a tree-form of the program that can be easily ran by the interpreter.

After the parser converts the input to the AST, it is ran thorough the resolver, which records variable invocations to ensure scoping is correct. The resolver also contains some additional features not specified by the book, including giving an error if a `return` or `break` statement are used inappropriately, if a variable is declared and never used, or a variable is used while it is guaranteed to be uninitialized.

After resolution, the program is ready to be execcuted by the interpreter. The interpreter implements the visitor pattern to provide a piece of functionality for each type of node that it visits on the AST. For example, upon visiting an If expression, the interpreter evaluates the condition (which recursively interprets the children node on the tree). If the evaluated expression is truthy, the statement in the true branch is executed. Otherwise, if an else branch exists, that statement is executed. Because this type of execution is analagous to walking through the AST, this type of interpreter is known as a tree-walk interpreter.

The remainder of the work on this interpreter is implementing the many features that exist in the Lox language. This includes implementing state (variables), control flow, functions, lexical scoping, and object oriented features.

In the end, I created a working Lox interpreter in Java with all of the features specified in the informal specification and more. However, one disadvantage of this interpreter is that performance-wise, it is quite slow, due to the fact that it is a tree-walk interpreter (which means that there are many pointer indirections), and that it is built on Java, which means that we have less control over potential optimizations made. This is why, the second half of the book involves implementing a bytecode virtual machine for Lox in C.

The Bytecode Virtual Machine

The bytecode virtual machine is the second of two interpreter that I implemented. This implementation was written in C, to give more control over the memory management (as it include writing a custom garbage collector for Lox), and to optimize for the performance compared to the tree-walk interpreter. Even though it's a virtual machine that reads compiled bytecode, technically speaking it's still an interpreter, because the virtual machine interprets the instructions rather for effects, than converting the instructions to actual machine code.

The way this virtual machine works is that it reads bytecode, executes an instruction based on that, and repeats the process. Bytecode is much more memory efficient than walking over an AST. Recently, many languages have moved to using a Just-In-Time compiler to compile into machine code as the program itself is running, which is usually even faster than a bytecode interpreter, but the cost of that is that it is much

more difficult to write a custom JIT compiler than to write a bytecode VM, so it would likely be out of scope for this project.

The virtual machine implemented is a stack-based VM, which means that values are stored on a stack, and instructions manipulate the top of the stack. For example, the expression `3 + 4` would push 3 and 4 onto the stack and then pop the two values and add them together, pushing the final result of 7 onto the stack again.

Similar to the tree-walk interpreter, this interpreter contains a scanner to convert text into tokens. However, the step after the scanner is now a compiler that compiles tokens into bytecode. The compiler used in the book uses a Pratt parser, an algorithm for top-down operator precedence, to turn its expressions into stack-VM bytecode. This compiler is a single-pass compiler, which works well for a simple language like Lox, and is very efficient performance-wise.

After the bytecode is produced, it is executed by the virtual machine interpreter. The interpreter reads in a bytecode instruciton, manipulates the stack based on the instruction, and repeats itself. This simple mechanism is powerful enough to emulate all of the features in the Lox programming language.

This was the point that I got to with the bytecode virtual machine this semester. The remainder of the book has chapters about adding variable support, functions and function calls, closures and scoping, garbage collection, classes and inheritance, and optimization. I plan on continuing and finishing the implementation of the bytecode virtual machine for Lox, and potentially expanding it into a fully featured custom programming language.

Conclusion

Over this winter study, I learned about and wrote two interpreters for the Lox programming language, one of which was a tree-walk interpreter written in Java and the other of wich was a bytecode virtual machine written in C. Through this process, I learned a great deal about how programming languages are made, including the process, the components, the design decisions, the terminology, and much more. I really enjoyed working on this project during winter study, and I felt that it was definitely worth my time pursuing this project.