Introduction

The first sentence of this introduction was supposed to be this one: “Interpreters are magical”. But one of the earliest reviewers, who wishes to remain anonymous, said that “sounds super stupid”. Well, Christian, I don’t think so! I still think that interpreters *are* magical! Let me tell you why.

On the surface they look deceptively simple: text goes in and something comes out. They are programs that take other programs as their input and produce something. Simple, right? But the more you think about it, the more fascinating it becomes. Seemingly random characters - letters, numbers and special characters - are fed into the interpreter and suddenly become **meaningful**. The interpreter gives them meaning! It makes sense out of nonsense. And the computer, a machine that’s built on understanding ones and zeroes, now understands and acts upon this weird language we feed into it - thanks to an interpreter that translates this language while reading it.

I kept asking myself: *how does this work?* And the first time this question began forming in my mind, I already knew that I’ll only be satisfied with an answer if I get to it by writing my own interpreter. So I set out to do so.

A lot of books, articles, blog posts and tutorials on interpreters exist. Most of the time, though, they fall into one of two categories. Either they are huge, incredibly heavy on theory and more targeted towards people who already have a vast understanding of the topic, or they are really short, provide just a small introduction to the topic, use external tools as black boxes and only concern themselves with “toy interpreters”.

One of the main sources of frustration was this latter category of resources, because the interpreters they explain only interpret languages with a really simple syntax. I didn’t want to take a shortcut! I truly wanted to understand how interpreters work and that included understanding how lexers and parsers work. Especially with a C-like language and its curly braces and semicolons, where I didn’t even know how to start parsing them. The

academic textbooks had the answers I was looking for, of course. But rather inaccessible to me, behind their lengthy, theoretical explanations and mathematical notation.

What I wanted was something between the 900 page book on compilers and the blog post that explains how to write a Lisp interpreter in 50 lines of Ruby code.

So I wrote this book, for you and me. This is the book I wish I had. This is a book for people who love to look under the hood. For people that love to learn by understanding how something really works.

In this book we're going to write our own interpreter for our own programming language - from scratch. We won't be using any 3rd party tools and libraries. The result won't be production-ready, it won't have the performance of a fully-fledged interpreter and, of course, the language it's built to interpret will be missing features. But we're going to learn a lot.

It's difficult to make generic statements about interpreters since the variety is so high and none are alike. What can be said is that the one fundamental attribute they all share is that they take source code and evaluate it without producing some visible, intermediate result that can later be executed. That's in contrast to compilers, which take source code and produce output in another language that the underlying system can understand.

Some interpreters are really small, tiny, and do not even bother with a parsing step. They just interpret the input right away. Look at one of the many Brainfuck interpreters out there to see what I mean.

On the other end of the spectrum are much more elaborate types of interpreters. Highly optimized and using advanced parsing and evaluation techniques. Some of them don't just evaluate their input, but compile it into an internal representation called bytecode and then evaluate this. Even more advanced are JIT interpreters that compile the input just-in-time into native machine code that gets then executed.

But then, in between those two categories, there are interpreters that parse the source code, build an abstract syntax tree (AST) out of it and then

evaluate this tree. This type of interpreter is sometimes called “tree-walking” interpreter, because it “walks” the AST and interprets it.

What we will be building in this book is such a tree-walking interpreter.

We’re going to build our own lexer, our own parser, our own tree representation and our own evaluator. We’ll see what “tokens” are, what an abstract syntax tree is, how to build such a tree, how to evaluate it and how to extend our language with new data structures and built-in functions.

The Monkey Programming Language & Interpreter

Every interpreter is built to interpret a specific programming language. That’s how you “implement” a programming language. Without a compiler or an interpreter a programming language is nothing more than an idea or a specification.

We’re going to parse and evaluate our own language called Monkey. It’s a language specifically designed for this book. Its only implementation is the one we’re going to build in this book - our interpreter.

Expressed as a list of features, Monkey has the following:

- C-like syntax
- variable bindings
- integers and booleans
- arithmetic expressions
- built-in functions
- first-class and higher-order functions
- closures
- a string data structure
- an array data structure
- a hash data structure

We’re going to take a detailed look at and implement each of these features in the rest of this book. But for now, let’s see what Monkey looks like.

Here is how we bind values to names in Monkey:

```
let age = 1;
let name = "Monkey";
let result = 10 * (20 / 2);
```

Besides integers, booleans and strings, the Monkey interpreter we're going to build will also support arrays and hashes. Here's what binding an array of integers to a name looks like:

```
let myArray = [1, 2, 3, 4, 5];
```

And here is a hash, where values are associated with keys:

```
let thorsten = {"name": "Thorsten", "age": 28};
```

Accessing the elements in arrays and hashes is done with index expressions:

```
myArray[0]       // => 1
thorsten["name"] // => "Thorsten"
```

The `let` statements can also be used to bind functions to names. Here's a small function that adds two numbers:

```
let add = fn(a, b) { return a + b; };
```

But Monkey not only supports `return` statements. Implicit return values are also possible, which means we can leave out the `return` if we want to:

```
let add = fn(a, b) { a + b; };
```

And calling a function is as easy as you'd expect:

```
add(1, 2);
```

A more complex function, such as a `fibonacci` function that returns the Nth Fibonacci number, might look like this:

```
let fibonacci = fn(x) {
  if (x == 0) {
    0
  } else {
```

```
    if (x == 1) {
      1
    } else {
      fibonacci(x - 1) + fibonacci(x - 2);
    }
  }
};
```

Note the recursive calls to `fibonacci` itself!

Monkey also supports a special type of functions, called higher order functions. These are functions that take other functions as arguments. Here is an example:

```
let twice = fn(f, x) {
  return f(f(x));
};

let addTwo = fn(x) {
  return x + 2;
};

twice(addTwo, 2); // => 6
```

Here `twice` takes two arguments: another function called `addTwo` and the integer `2`. It calls `addTwo` two times with first `2` as argument and then with the return value of the first call. The last line produces `6`.

Yes, we can use functions as arguments in function calls. Functions in Monkey are just values, like integers or strings. That feature is called “first class functions”.

The interpreter we’re going to build in this book will implement all these features. It will tokenize and parse Monkey source code in a REPL, building up an internal representation of the code called abstract syntax tree and then evaluate this tree. It will have a few major parts:

- the lexer
- the parser
- the Abstract Syntax Tree (AST)
- the internal object system

- the evaluator

We're going to build these parts in exactly this order, from the bottom up. Or better put: starting with the source code and ending with the output. The drawback of this approach is that it won't produce a simple "Hello World" after the first chapter. The advantage is that it's easier to understand how all the pieces fit together and how the data flows through the program.

But why the name? Why is it called "Monkey"? Well, because monkeys are magnificent, elegant, fascinating and funny creatures. Exactly like our interpreter.

And why the name of the book?

Why Go?

If you read this far without noticing the title and the words "in Go" in it, first of all: congratulations, that's pretty remarkable. And second: we will write our interpreter in Go. Why Go?

I like writing code in Go. I enjoy using the language, its standard library and the tools it provides. But other than that I think that Go is in possession of a few attributes that make it a great fit for this particular book.

Go is really easy to read and subsequently understand. You won't need to decipher the Go code I present to you in this book. Even if you are not an experienced Go programmer. I'd bet that you can follow this book along even if you've never written a single line of Go in your life.

Another reason is the great tooling Go provides. The focus of this book is the interpreter we are writing - the ideas and concepts behind it and its implementation. With Go's universal formatting style thanks to `gofmt` and a testing framework built-in, we can concentrate on our interpreter and not worry about 3rd party libraries, tools and dependencies. We won't be using any other tools in this book other than the ones provided by the Go programming language.

But I think much more important is that the Go code presented in this book maps closely to other and possibly more low-level languages, like C, C++ and Rust. Maybe the reason for this is Go itself, with its focus on simplicity, its stripped-down charm and lack of programming language constructs that are absent in other languages and hard to translate. Or maybe it's because of the way I chose to write Go for this book. Either way, there won't be any meta-programming tricks to take a shortcut that nobody understands anymore after two weeks and no grandiose object-oriented designs and patterns that need pen, paper and the sentence "actually, it's pretty easy" to explain.

All of these reasons make the code presented here easy to understand (on a conceptual as well as a technical level) and reusable for you. And if you, after reading this book, choose to write your own interpreter in another language this should come in handy. With this book I want to provide a starting point in your understanding and construction of interpreters and I think the code reflects that.

How to Use this Book

This book is neither a reference, nor is it a theory-laden paper describing concepts of interpreter implementation with code in the appendix. This book is meant to be read from start to finish and I recommend that you follow along by reading, typing out and modifying the presented code.

Each chapter builds upon its predecessor - in code and in prose. And in each chapter we build another part of our interpreter, piece by piece. To make it easier to follow along, the book comes with a folder called `code`, that contains, well, code. If your copy of the book came without the folder, you can download it here:

https://interpreterbook.com/waiig_code_1.7.zip

The `code` folder is divided into several subfolders, with one for each chapter, containing the final result of the corresponding chapter.

Sometimes I'll only allude to something being in the code, without showing the code itself (because either it would take up too much space, as is the case with the test files, or it is just some detail) - you can find this code in the folder accompanying the chapter, too.

Which tools do you need to follow along? Not much: a text editor and the Go programming language. Any Go version above 1.0 should work, but just as a disclaimer and for future generations: when I wrote this book I was using **Go 1.7**. Now, with the latest update of the book, I'm using **Go 1.14**.

If you're using **Go >= 1.13** the code in the `code` folder should be runnable out of the box.

If you're on an older version of Go, one that doesn't support Go modules, I recommend using direnv, which can change the environment of your shell according to an `.envrc` file. Each sub-folder in the code folder accompanying this book contains such an `.envrc` file that sets the `GOPATH` correctly for this sub-folder. That allows us to easily work with the code of different chapters.

And with that out of the way, let's get started!