

WRITING A COMPILER IN GO

THORSTEN BALL

CONTENTS

[Acknowledgments](#_bookmark0) 3

[Introduction](#_bookmark1) 4

[Evolving Monkey](#_bookmark2) 5

[The Past and Present](#_bookmark3) 5

[The Future](#_bookmark4) 7

[Use This Book](#_bookmark5) 8

1. [Compilers & Virtual Machines](#_bookmark6) 10

[Compilers](#_bookmark7) 11

[Virtual and Real Machines](#_bookmark8) 14

[Real Machines](#_bookmark9) 14

[What Is a Virtual Machine?](#_bookmark10) 19

[Why Build One?](#_bookmark11) 21

[Bytecode](#_bookmark12) 23

[What We’re Going to Do, or: the Duality of VM and Compiler](#_bookmark13) 24

1. [Hello Bytecode!](#_bookmark14) 26

[First Instructions](#_bookmark15) 27

[Starting With Bytes](#_bookmark16) 28

[The Smallest Compiler](#_bookmark17) 32

[Bytecode, Disassemble!](#_bookmark18) 37

[Back to the Task at Hand](#_bookmark19) 41

[Powering On the Machine](#_bookmark20) 43

[Adding on the Stack](#_bookmark21) 49

[Hooking up the REPL](#_bookmark22) 54

1. [Compiling Expressions](#_bookmark23) 57

[Cleaning Up the Stack](#_bookmark24) 57

[Infix Expressions](#_bookmark25) 61

[Booleans](#_bookmark26) 65

[Comparison Operators](#_bookmark27) 69

[Prefix Expressions](#_bookmark28) 76

1. [Conditionals](#_bookmark29) 82

[Jumps](#_bookmark30) 85

[Compiling Conditionals](#_bookmark31) 87

[Executing Jumps](#_bookmark32) 99

[Welcome Back, Null!](#_bookmark33) 102

1. [Keeping Track of Names](#_bookmark34) 110

[The Plan](#_bookmark35) 111

[Compiling Bindings](#_bookmark36) 112

[Introducing: the Symbol Table](#_bookmark37) 114

[Using Symbols in the Compiler](#_bookmark38) 117

[Adding Globals to the VM](#_bookmark39) 120

1. [String, Array and Hash](#_bookmark40) 125

[String](#_bookmark41) 125

[Array](#_bookmark42) 130

[Hash](#_bookmark43) 134

[Adding the index operator](#_bookmark44) 140

1. [Functions](#_bookmark45) 146

[Dipping Our Toes: a Simple Function](#_bookmark46) 146

[Representing Functions](#_bookmark47) 146

[Opcodes to Execute Functions](#_bookmark48) 148

[Compiling Function Literals](#_bookmark49) 150

[Compiling Function Calls](#_bookmark50) 163

[Functions in the VM](#_bookmark51) 166

[A Little Bonus](#_bookmark52) 175

[Local Bindings](#_bookmark53) 176

[Opcodes for Local Bindings](#_bookmark54) 177

[Compiling Locals](#_bookmark55) 180

[Implementing Local Bindings in the VM](#_bookmark56) 191

[Arguments](#_bookmark57) 199

[Compiling Calls With Arguments](#_bookmark58) 200

[Resolving References to Arguments](#_bookmark59) 205

[Arguments in the VM](#_bookmark60) 207

1. [Built-in Functions](#_bookmark61) 216

[Making the Change Easy](#_bookmark62) 217

[Making the Change: the Plan](#_bookmark63) 222

[A New Scope for Built-in Functions](#_bookmark64) 223

[Executing built-in functions](#_bookmark65) 227

1. [Closures](#_bookmark66) 233

[The Problem](#_bookmark67) 234

[The Plan](#_bookmark68) 235

[Everything’s a closure](#_bookmark69) 236

[Compiling and resolving free variables](#_bookmark70) 246

[Creating real closures at run time](#_bookmark71) 257

[Recursive Closures](#_bookmark72) 261

1. [Taking Time](#_bookmark73) 273

[Resources](#_bookmark74) 277

[Feedback](#_bookmark75) 279

[Changelog](#_bookmark76) 280

# ACKNOWLEDGMENTS

I started writing this book one month after my daughter was born and finished shortly after her first birthday. Or, in other words: this book wouldn’t exist without the help of my wife. While our baby grew into the wonderful girl she is now and rightfully demanded the attention she deserves, my wife always created time and room for me to write. I couldn’t have written this book without her steady support and unwavering faith in me. Thank you!

Thanks to Christian for supporting me from the start again with an open ear and encouragement. Thanks to Ricardo for providing invaluable, in-depth feedback and expertise. Thanks to Yoji for his diligence and attention to detail. Thanks to all the other beta-readers for helping to make this book better!

# INTRODUCTION

It might not be the most polite thing to do, but let’s start with a lie: the prequel to this book, *Writing An Interpreter In Go*, was much more successful than I ever imagined it would be. Yes, that’s a lie. Of course, I *imagined* its success. The name on the top of bestseller lists, me showered with praise and admiration, invited to fancy events, strangers walking up to me in the street, wanting to get their copy signed – who wouldn’t imagine that when writing a book about a programming language called Monkey? But, now, in all seriousness, the truth: I really didn’t *expect* the book to be as successful as it was.

Sure, I had a feeling that some people might enjoy it. Mainly because it’s the book I myself wanted to read, but couldn’t find. And on my fruitless search I saw other people looking for the exact same thing: a book about interpreters that is easy to understand, doesn’t take shortcuts and puts runnable and tested code front and center. If I could write a book like that, I thought, there might just be a chance that others would enjoy it, too.

But enough about my imagination, here’s what actually happened: readers really enjoyed what I wrote. They not only bought and read the book, but sent me emails to thank me for writing it. They wrote blog posts about how much they enjoyed it. They shared it on social networks and upvoted it. They played around with the code, tweaked it, extended it and shared it on GitHub. They even helped to fix errors in it. Imagine that! They sent me fixes for my mistakes, all the while saying sorry for finding them. Apparently, they couldn’t imagine how thankful I was for every suggestion and correction.

Then, after reading one email in which a reader asked for *more*, something in me clicked. What lived in the back of my mind as an idea turned into an obligation: I have to write the second part. Note that I didn’t just write “*a* second part”, but “*the* second part”. That’s because the first book was born out of a compromise.

When I set out to write *Writing An Interpreter In Go* the idea was not to follow it up with a sequel, but to only write a single book. That changed, though, when I realized that the final book would be too long. I never wanted to write something that scares people off with its size. And even if I did, completing the book would probably take so long that I would have most likely given up long before.

That led me to a compromise. Instead of writing about building a tree-walking interpreter and turning it into a virtual machine, I would *only* write about the tree-walking part. That turned into *Writing An Interpreter In Go* and what you’re reading now is the sequel I have always wanted to write.

But what exactly does sequel mean here? By now you know that this book doesn’t start with “Decades after the events in the first book, in another galaxy, where the name Monkey has no meaning…” No, this book is meant to seamlessly connect to its predecessor. It’s the same approach, the same programming language, the same tools and the codebase that we left at the end of the first book.

The idea is simple: we pick up where we left off and continue our work on Monkey. This is not only a successor to the previous book, but also a sequel to Monkey, the next step in its evolution. Before we can see what that looks like, though, we need look back, to refresh our memory of Monkey.

### EVOLVING MONKEy

#### THE PAST AND PRESENT

In *Writing An Interpreter In Go* we built an interpreter for the programming language Mon- key. Monkey was invented with one purpose in mind: to be built from scratch in *Writing An Interpreter In Go* and by its readers. Its only oﬀicial implementation is contained in *Writing An Interpreter In Go*, although many unoﬀicial ones, built by readers in a variety of languages, are floating around the internet.

In case you forgot what Monkey looks like, here is a small snippet that tries to cram as much of Monkey’s features into as few lines as possible:

**let** name = "Monkey";

**let** age = 1;

**let** inspirations = ["Scheme", "Lisp", "JavaScript", "Clojure"];

**let** book = {

"title": "Writing A Compiler In Go", "author": "Thorsten Ball",

"prequel": "Writing An Interpreter In Go"

};

**let** printBookName = fn(book) { **let** title = book["title"]; **let** author = book["author"]; puts(author + " - " + title);

};

printBookName(book);

*// => prints: "Thorsten Ball - Writing A Compiler In Go"*

**let** fibonacci = fn(x) {

**if** (x == 🅓) {

🅓

} **else** {

**if** (x == 1) {

**return** 1;

} **else** {

fibonacci(x - 1) + fibonacci(x - 2);

}

}

};

**let** map = fn(arr, f) {

**let** iter = fn(arr, accumulated) {

**if** (len(arr) == 🅓) { accumulated

} **else** {

iter(rest(arr), push(accumulated, f(first(arr))));

}

};

iter(arr, []);

};

**let** numbers = [1, 1 + 1, 4 - 1, 2 \* 2, 2 + 3, 12 / 2]; map(numbers, fibonacci);

*// => returns: [1, 1, 2, 3, 5, 8]*

Translated into a list of features, we can say that Monkey supports:

* integers
* booleans
* strings
* arrays
* hashes
* prefix-, infix- and index operators
* conditionals
* global and local bindings
* first-class functions
* return statements
* closures

Quite a list, huh? And we built all of these into our Monkey interpreter ourselves and – most importantly! – we built them from scratch, without the use of any third-party tools or libraries.

We started out by building the lexer that turns strings entered into the REPL into tokens. The lexer is defined in the lexer package and the tokens it generates can be found in the token package.

After that, we built the parser, a top-down recursive-descent parser (often called a Pratt parser) that turns the tokens into an abstract syntax tree, which is abbreviated to AST. The nodes of the AST are defined in the ast package and the parser itself can be found in the parser package.

After it went through the parser, a Monkey program is then represented in memory as a tree and the next step is to evaluate it. In order to do that we built an evaluator. That’s another name for a function called Eval, defined in the evaluator package. Eval recursively walks down the AST and evaluates it, using the object system we defined in the object package to produce values. It would, for example, turn an AST node representing 1 + 2 into an object.Integer{Value: 3}. With that, the life cycle of Monkey code would be complete and the result printed to the REPL.

This chain of transformations – from strings to tokens, from tokens to a tree and from a tree to

object.Object – is visible from start to end in the main loop of the Monkey REPL we built:

*// repl/repl.go*

**package** repl

**func** Start(in io.Reader, out io.Writer) { scanner := bufio.NewScanner(in)

env := object.NewEnvironment()

**for** {

fmt.Fprintf(out, PROMPT) scanned := scanner.Scan() **if** !scanned {

**return**

}

line := scanner.Text()

l := lexer.New(line) p := parser.New(l)

program := p.ParseProgram()

**if** len(p.Errors()) != 🅓 { printParserErrors(out, p.Errors())

**continue**

}

evaluated := evaluator.Eval(program, env)

**if** evaluated != nil {

io.WriteString(out, evaluated.Inspect()) io.WriteString(out, "\n")

}

}

}

That’s where we left Monkey at the end of the previous book.

And then, half a year later, the [*The Lost Chapter: A Macro System For Monkey*](https://interpreterbook.com/lost)resurfaced and told readers how to get Monkey to program itself with macros. In this book, though, *The Lost Chapter* and its macro system won’t make an appearance. In fact, it’s as if the *The Lost Chapter* was never found and we’re back at the end of *Writing An Interpreter In Go*. That’s good, though, because we did a great job implementing our interpreter.

Monkey worked exactly like we wanted it to and its implementation was easy to understand and to extend. So one question naturally arises at the beginning of this second book: why change any of it? Why not leave Monkey as is?

Because we’re here to learn and Monkey still has a lot to teach us. One of the goals of *Writing An Interpreter In Go* was to learn more about the implementation of the programming languages we’re working with on a daily basis. And we did. A lot of these “real world” languages did start out with implementations really similar to Monkey’s. And what we learned by building Monkey helps us to understand the fundamentals of their implementation and their origins.

But languages grow and mature. In the face of production workloads and an increased demand for performance and language features, the implementation and architecture of a language often change. One side effect of these changes is that the implementation loses its similarity to Monkey, which wasn’t built with performance and production usage in mind at all.

This gap between fully-grown languages and Monkey is one of the biggest drawbacks of our Monkey implementation: its architecture is as removed from the architecture of actual real- world languages as a soapbox car is from a Formula 1 car. It has four wheels and a seat, sure, it can help to learn the fundamentals of steering, yes, but the fact that it’s missing an engine is hard to ignore.

In this book we’re going to reduce this gap between Monkey and *real* languages. We’ll put something real under the hood of our Monkey soapbox car.

#### THE FUTURE

We’re going to turn our tree-walking and on-the-fly-evaluating interpreter into a bytecode com- piler and a virtual machine that executes the bytecode.

That’s not only immensely fun to build but also one of the most common interpreter archi- tectures out there. Ruby, Lua, Python, Perl, Guile, different JavaScript implementations and many more programming languages are built this way. Even the mighty Java Virtual Machine

interprets bytecode. Bytecode compilers and virtual machines are everywhere – and for good reason.

Besides providing a new layer of abstraction – the bytecode that’s passed from the compiler to the virtual machine – that makes the system more modular, the main appeal of this architecture lies in its performance. Bytecode interpreters are fast.

Want numbers? At the end of this book we’ll have an implementation of Monkey that’s three times faster than its predecessor from the first book:

$ ./monkey-fibonacci -engine=eval

engine=eval, result=9227465, duration=27.2🅓4277379s

$ ./monkey-fibonacci -engine=vm

engine=vm, result=9227465, duration=8.876222455s

Yes, three times faster – without low-level tweaking or extensive optimizations. Sound good? Ready to write some code? Perfect. Let’s get the practicalities out of the way and we can start.

## USE THIS BOOK

Like the first one, this book comes with only a tiny number of instructions: you best read it from start to finish and you’ll get the most out of it if you follow along by reading, typing out and playing around with the presented code. That’s it.

These books are *practical* books. They are about writing code and building something. If you want to immerse yourself in the theory of programming-language construction you’re better off by choosing one of the canonical textbooks. That’s not to say that you won’t learn anything here, no. I’ll try my best to guide you along and explain what everything is and how the pieces fit together. It just won’t be as academic as a textbook on compilers. But that’s exactly how I intend it to be.

This book, just like its predecessor, comes with a folder called code. If your copy of the book came without it, you can download it here:

<https://compilerbook.com/wacig_code_1.2.zip>

In this folder you’ll find subfolders for each of the chapters in which we write code. Each contains the codebase as it stands at the end of the corresponding chapter, which can be helpful if you get stuck while following along.

The subfolder 🅓🅓, however, is special and also what differentiates this book from the first one: we don’t start with a clean slate, but build upon the codebase of the previous book. It doesn’t correspond to a chapter of *this* book, but contains the complete codebase as we left it at the end of the previous one. That also means it doesn’t include the macro system from *The Lost Chapter*, but if you’re a fan of the macros, it shouldn’t be too hard to keep the macro-expansion step around.

The code contained in these folders is the focus of this book. I try to show most of it, but sometimes I’ll only refer to something located in the codebase, without actually showing it. Why? Most of the time because it’s a repetition of something we’ve seen before and showing it would take up too much space.

So much about the content of the code folder. Now, let’s talk about tools because I have some good news: you don’t need many. In fact, a text editor and an installation of the Go programming language should be enough. Which version of Go? At least **Go 1.10**, because that’s what I’m using at the time of writing and because we will use a tiny number of features that were only introduced in Go 1.8 and 1.9.

If you’re *not* using **Go >=1.13**, then I also recommend using [direnv](http://direnv.net/) to work with the code folder. *direnv* can change the environment of your shell according to an .envrc file. Whenever you cd into a folder, *direnv* checks whether the folder contains an .envrc file and executes it. Each subfolder in code contains such an .envrc file that sets the GOPATH correctly for this subfolder. That allows us to just cd into a subfolder and execute the code really easily.

But if you *are* on **Go >=1.13** you shouldn’t need to set a GOPATH, since the code folders contain

go.mod files, making them work with the go command out of the box.

And that’s all there is to say about the practicalities of this book. Read it, code along and, most importantly, have fun.

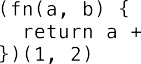
CHAPTER 1

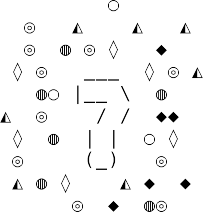
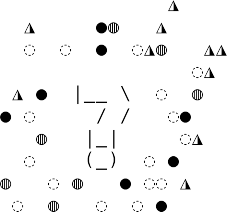
COMPILERS & VIRTUAL MACHINES

For most programmers the word “compiler” has an intimidating ring to it. And even if you’re not intimidated by it, you can’t deny that compilers and what they do is surrounded by an air of mystery and amazement. They produce machine code seemingly nobody mortal can read or write. They do magic optimizations that *somehow* make code run faster. They can take a long time to run – minutes or even tens of minutes. If the rumours are true they sometimes even run for hours. If it takes *that* long, what they’re doing must be extraordinary.

Speaking of rumours: it’s said that compilers are incredibly huge and incredibly complex. In fact, they can often be found listed among the most complex software projects to ever exist – with the claims backed up by numbers. Here’s a taste: the LLVM & Clang projects currently consist of around 3 million lines of code. The GNU Compiler Collection, GCC, is even bigger. 15 million lines of code.

There are not a lot of people that look at these numbers, open their text editor and say “you know what? Let’s build one!” They certainly do not evoke the sense that one could write a compiler in an afternoon.







Virtual machines, too, are largely understood to be mystical creatures, roaming through the dark lower layers of software development, seldom seen in light and even less understood. They, too, attract their fair share of rumours and speculations. Some say that virtual machines have something to do with compilers. Others insist that their programming language is, in fact, such a virtual machine. Another group claims that virtual machines allow them to run an operating system inside another operating system.

It doesn’t help that all of it is true.

But here’s the thing. At their core, compilers and virtual machines are ideas – patterns – just like “interpreter” or “web server” are ideas that can have multiple implementations, ranging from tiny to massive. Being intimidated by them after looking at a project like GCC makes as much sense as giving up on building a website after looking at GitHub.

Sure, it’s not a small task to build a compiler for a virtual machine. But it’s also not as insurmountable as reputation suggests. And once we have a better understanding of these core ideas, of what a compiler and a virtual machine essentially are, you’ll see that you can, in fact, build a compiler in an afternoon.

The first step towards that is finding out what “compiling” means.

## COMPILERS

If I’d ask you to name a compiler, you’d probably and without missing a beat would give me a name like GCC, or Clang, or the Go compiler. In any case, it would most certainly be a compiler for a programming language. Chances are that it’s a compiler that produces executables. I would, too, because that’s just what we associate with the word “compiler”.

But compilers come in all shapes and sizes and compile all kinds of things, not just programming languages, including regular expressions, database queries and even HTML templates. I bet you use one or two compilers every day without even realizing it. That’s because the definition of “compiler” itself is actually quite loose, much more so than one would expect. Here is [Wikipedia’s](https://en.wikipedia.org/wiki/Compiler) [version](https://en.wikipedia.org/wiki/Compiler):

A compiler is computer software that transforms computer code written in one programming language (the source language) into another computer language (the target language). Compilers are a type of translator that support digital devices, pri- marily computers. The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

Compilers are translators. That’s vague. And a compiler that translates high-level languages to produce executables is just one special type of compiler? Sounds counter-intuitive, doesn’t it? You’d think that producing executables *is just what a compiler does*: that’s what GCC does, that’s what Clang does, that’s what the Go compiler does. Shouldn’t that be the first line of the definition? How can this be non-essential?

The solution to this riddle is another one: what’s an executable if not source code in a language the computer can natively understand? Hence, “compiling to native code” is the same as com- piling to machine code. Yes, really, producing executables is just one variation of “translating source code”.

You see, compilers are fundamentally about translation, because translation is how they *imple- ment a programming language*.

Let’s make sense of that sentence by taking a step back here. Programming means giving instructions to a computer. We, the programmers, write these instructions in programming languages the computer can understand. There is no point in using any other language. Now, *implementing* a programming language means making the computer understand it. There are two ways to do that: either interpret the language for the computer on-the-fly or translate it into another language, one the computer already understands.

It’s exactly like we, as humans, can help a friend understand a language she doesn’t speak. We can either listen, translate in our head and repeat the translation to her, or we can write the

translation down so she can read and understand it herself. We can either act as an interpreter or as a compiler.

This might sound like interpreters and compilers are opposites. But while their approach is different, they share a lot of things in their construction. They both have a frontend that reads in source code written in the source language and turns it into a data structure. In both, compiler and interpreter, this frontend is usually made up of a lexer and a parser that together generate a syntax tree. So here, in the front part, they have a lot of similarities. After that, when they both traverse the AST, that’s when their paths diverge.

Since we already built an interpreter we know what it does when traversing the AST: it evaluates it. That is, it executes the instructions encoded in the tree. If a node in the tree represents the source language’s statement puts("Hello World!"), then the interpreter would print “Hello World!” when evaluating the node.

A compiler, on the other hand, wouldn’t print anything. Instead, it would generate source code in another language, called the target language. The source code would contain the target language’s equivalent of the source language’s puts("Hello World!"). The resulting code could then be executed by the computer and "Hello World!" would appear on the screen.

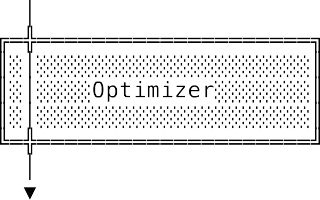
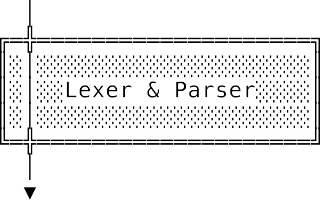
This is where things get really interesting. In which target language does the compiler generate source code? Which language does the computer understand? And how does the compiler generate code in this language? As text, or in a binary format? In a file, or in memory? But much more importantly: what exactly does it generate in this target language? What if the target language doesn’t have an equivalent of puts? What should the compiler generate instead?

Generally speaking, we have to give the same answer to every one of these questions. Soft- ware development’s number one answer, the only hard, capital-T Truth in programming: “it depends.”

Sorry to let you down there, but the answers to these questions depend on a multitude of variables and requirements: the source language, the architecture of the machine that executes the target language, how the output is going to be used (is it executed directly? compiled again? interpreted?), how fast the output needs to run, how fast the compiler itself needs to run, how big the generated source code can be, how much memory the compiler is allowed to use, how much memory the resulting program can use, and how…

The variety between compilers is so high that we can’t make a lot of universal statements about their architecture. That being said, we can ignore the details for a minute now and sketch out the architecture of something like the archetype of a compiler:





This shows the life cycle of source code that’s being translated to machine code. Here’s what happens.

First, the source code is tokenized and parsed by the lexer and the parser. We’re familiar with this part from our interpreter. It’s called the frontend. The source code is turned from text into an AST.

After that a component called the “optimizer” (or sometimes also called “compiler” – I know, I know, …) might translate the AST into another internal representation (IR). This additional IR might just be another syntax tree, or maybe a binary format, or even a textual format. The reasons for this additional translation into another IR are manifold, but the major one is that the IR might lend itself better to optimizations and translation into the target language than the AST would.

This new IR then goes through an optimization phase: dead code is eliminated, simple arith- metic is pre-calculated, code that doesn’t need to be in the body of a loop is moved out of it,

… A ton of possible optimizations exists.

Finally, the code generator, also called the backend, generates the code in the target language. This is where the compilation happens. Here’s where the code hits the file system, so to say. After that, we can execute the result and see the computer perform what we instructed it to in the original source code.

This is how the *simplest* compilers work. And even here a thousand variations are possible. For example, the optimizer could do multiple “passes” over the IR, meaning that it traverses the IR multiple times and each time does a different optimization: removing dead code in one pass, inlining function calls in another one, for example. Or maybe the compiler doesn’t do

optimizations on the IR at all, but only on the source code in the target language. Or only on the AST, or on both. Or it doesn’t do any optimizations ever. Or maybe it doesn’t even have another IR besides the AST. And maybe it doesn’t output machine code, but assembly language, or another high-level language. Or it has multiple backends that can generate machine code for multiple architectures. It all depends on the specific use case.

And then again, a compiler doesn’t even have to be a tool you run on the command line, that reads in source code and outputs code in a file, like gcc or go. It can just as well be a single function that takes in an AST and returns a string. That’s also a compiler. A compiler can be written in a few hundred lines of code or have millions of them.

But underlying all of these lines of code is the fundamental idea of translation. Compilers take in source code in one language and generate source code in another one. The rest – again – “depends”, with the majority on the target language. What the target language is capable of and by which machine it can be executed shapes the design of a compiler like nothing else.

Now, what if we didn’t have to *choose* a target language, but could invent our own? And what if we don’t leave it at that and also dream up the machine that executes this language?

## VIRTUAL AND REAL MACHINES

You probably associate “virtual machine” with software like VMWare or Virtualbox. These are programs that emulate a computer, including a disk drive, hard drive, graphics card, etc. They allow you to, for example, run a different operating system inside this emulated computer. Yes, these are virtual machines. But that’s not what we’re here to talk about. That’s the *other* type of virtual machine.

What we are going to talk about (and later build) are virtual machines that are used to imple- ment programming languages. Sometimes they consist of just a few functions, other times they make up a few modules and on occasion they’re a collection of classes and objects. It’s hard to pin their shape down. But that doesn’t matter. What’s important is this: they don’t emulate an existing machine. They *are* the machine.

The “virtual” is due to the fact they only exist in software, not in hardware, and are thus purely abstract constructs. The “machine” describes their behaviour. These software constructs act like a machine. And not just any machine, no. They mimic the behaviour of their hardware equivalent: computers.

That means, in order to understand and build a virtual machine, we need to learn how a real machine works.

#### REAL MACHINES

“So, how does a computer work?”

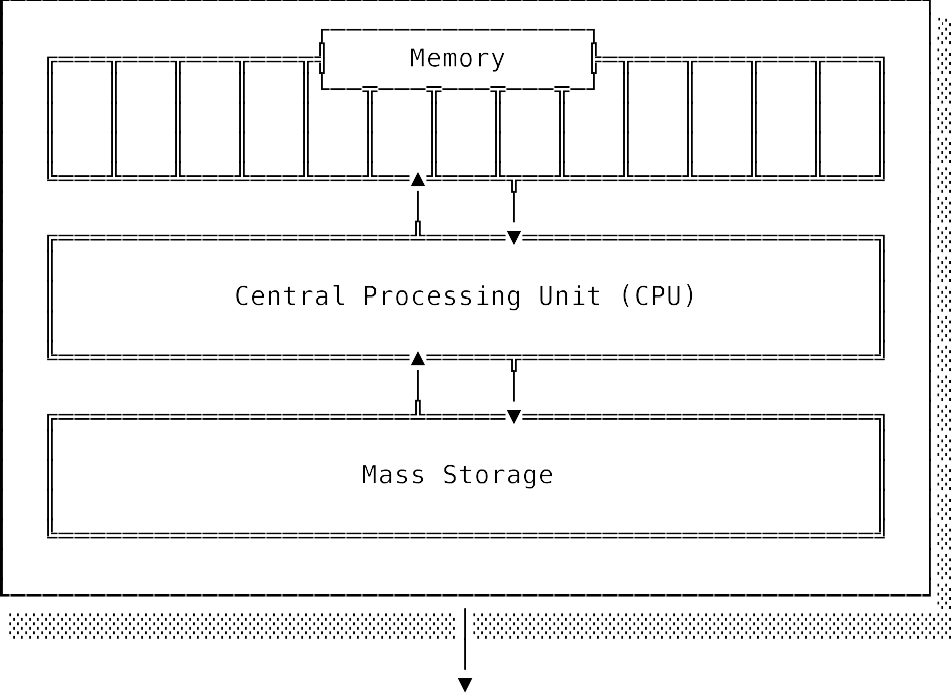
What might sound like an intimidating question can actually be answered in five minutes and with a piece of paper. I don’t know how fast you can read and I’m certain I can’t show you what I draw on a scrap of paper, but let me try anyway.

Nearly every computer in your life is built according to the [Von Neumann architecture](https://en.wikipedia.org/wiki/Von_Neumann_architecture), which describes a way to build a fully-functioning computer with a surprisingly tiny number of parts.

In Von Neumann’s model a computer has two central parts: a processing unit, which contains an arithmetic logic unit (ALU) and multiple processor registers, and a control unit with an instruction register and a program counter. Together they’re called the central processing unit,

often shortened to CPU. Besides that, the computer also contains memory (RAM), mass storage (think: hard drive) and input/output devices (keyboard and display).

CPU, memory, mass storage and I/O. Here’s a rough sketch of such a computer:



As soon as the computer is turned on, the CPU:

* 1. **Fetches an instruction from memory**. The program counter tells the CPU where in memory it can find the next instruction.
  2. **Decodes the instruction**. To identify which operation should be executed.
  3. **Executes the instruction**. This can mean either modifying the contents of its registers, or transferring data from the registers to memory, or moving data around in memory, or generating output, or reading input…

… and then *goto 1* again.

These three steps are called the **fetch-decode-execute cycle**, or instruction cycle. Yes, that’s the cycle known from sentences like “the clock speed of a computer is expressed in cycles per second, e.g. 500 MHz” or “we’re wasting CPU cycles here”.

That was a brief and easy to understand description of how a computer works. But we can make it even easier for us. In this book we don’t care about mass storage components and only tangentially about the I/O mechanisms. What we’re interested in is the interplay between CPU and memory. That means we can make that our focus and safely ignore hard drives and displays.

We start our investigation with this question: how does the CPU address different parts of memory? Or, in other words: how does the CPU know where to store and retrieve things that are located in memory?

We’re given a first hint by how the CPU fetches instructions. The program counter, a part of the CPU, keeps track of where to fetch the next instruction. And “counter” is to be taken quite literally here: the computer simply uses numbers to address different parts of memory. Yes, numbers.

At this point I’m tempted to write “just think of memory as a huge array”, but I’m scared of someone hitting me over the head with a heavy, leather-bound tome called “Things about memory that are *absolutely and without any doubt not like an array, you doofus*”, so I won’t. But, yes, just like we, as programmers, use numbers as indexes to access single elements in an array, the CPU uses numbers as addresses when accessing data in memory.

Instead of “array elements” computer memory is segmented into [“words”](https://en.wikipedia.org/wiki/Word_(computer_architecture)). Now what’s a “word”? It’s the smallest addressable region of memory. It’s the base unit when accessing memory. The size of a word varies and depends on the CPU type among other things. In the computers you and I use, however, 32 and 64 bits word sizes are the standard.

Let’s say we have a fictional computer with a word size of 8 bits and 13 bytes of memory. One word in memory can hold one ASCII character and if we store the string Hello, World! to memory it looks like this:





The letter “H” has the memory address 0, “e” has 1, the first “l” has 2, “W” has 7 and so on. We could access every single letter of the string Hello, World! by using the memory addresses 0 to 12. “Hey CPU, fetch the word at memory address 4” would result in the CPU fetching the letter “o”. Straightforward, right? I know what you’re thinking right now and, yes, if we take such a number – a memory address – and save it to another place in memory, we create a pointer.

That’s the *basic* idea behind addressing data in memory and how the CPU knows where to fetch and store data. As always, though, the real world is a mess.

As I already mentioned, the size of a word differs between different computers. Sometimes it’s 8 bits, sometimes 16, 24, 32, or 64 bits. Sometimes the word size used by the CPU is independent of the address size. And if that’s not enough to deal with: some computers do something entirely different and use [byte-addressing](https://en.wikipedia.org/wiki/Byte_addressing) instead of the just described word-addressing.

And if you’re using word-addressing and want to address a single byte (which is not that uncommon) you not only have to deal with different word sizes but also work with offsets. That in turn is expensive and has to be optimized.

On top of that: the idea that we can simply tell the CPU where to store and retrieve data in memory is something like a fairytale. It’s correct on a conceptual level and helpful when learning, but memory access today is abstracted away and sits behind layers and layers of security and performance optimizations. Memory is not the wild west anymore – we can’t just

go around and access any memory location we want. Security rules and a mechanism called virtual memory try their best to stop that from happening.

Let me stop right here, though, before we take a huge detour and end up talking about the inner workings of virtual memory. That’s not why we’re here. What you can take from this excursion is that there’s more to memory access today than passing a number to the CPU. Not only are there security rules in place, but in the last few decades of computing emerged a set of less strict conventions around the use of memory.

One novel aspect of the Von Neumann architecture was that a computer’s memory contains not only data, but also *programs*, which are the CPU instructions that make up a program. To our programmer ears the idea of mixing data and code sounds like a recipe for tears. And to the ears of programmers a few generations ago, it probably sounded like that, too. Because what they did was to establish conventions around the use of memory that stopped that from happening.

While programs are stored in the same memory as any other data, they’re most often not stored in the same locations. Specific areas of memory are used to store specific things. That’s not only by convention, but also dictated by the operating system, the CPU and the rest of the computer architecture.

“Dumb” data, as in “the contents of a text file” or “the response of a HTTP request”, goes in one region of memory. The instructions that make up a program are stored and accessed in another region, where the CPU can easily fetch them. Another region holds the static data that’s used by a program and yet another region is empty and uninitialized, but reserved so that it can be used by a program once it’s running. The instructions belonging to the operating system kernel have their own special region in memory.

By the way: while programs and “dumb data” may reside in different memory locations, again, the important thing here is that they *are* both stored in the same memory. Saying “data and programs are both stored in memory” makes it sound as if they’re distinct, when in fact programs – made up of instructions – are just data too. Instructions only become instructions once the CPU fetches them from memory, decodes them, and finds out that they are, in fact, proper instructions and then executes them. If the CPU attempts to decode data that’s not a valid instruction, well, then the consequences will depend on how the CPU was designed. It could trigger an event and give the program a chance to recover or it could just stop.

For us, the most interesting thing about this is one particular memory region. It’s the memory region that holds the stack. Yes, *the* stack. Drum roll, fanfare, spot light, deep voice: *The Stack*. You might have heard of him. “Stack overflow” is probably his most famous work, followed by the less popular but equally respected “stack trace”.

So, what is it? It’s a region in memory where data is managed in a last-in-first-out (LIFO) manner. The data in it grows and shrinks, you push elements on to the stack and later pop them off. Just like the stack data structure. But unlike this generic data structure, *the stack* is focused on one purpose: it’s used to implement the [*call stack*](https://en.wikipedia.org/wiki/Call_stack).

Yes, let’s stop right there; it *is* confusing. “Stack”, “the stack”, “stack data structure”, “call stack” – not really self-explanatory, is it? It doesn’t help that these names are often used interchangeably and thrown around and mixed together. But, thankfully, if we are careful with the names and pay attention to the “why” behind them, things become much clearer. So let’s try this again and do this step by step.

We have a memory region where the CPU accesses and stores data in a LIFO manner. It does this in order to implement a specialized version of *a stack*, called *a call stack*.

Why does it need *a call stack*? Because the CPU (or maybe: the programmer that wants the CPU to work as intended) needs to keep track of certain information in order to execute a program. The *call stack* helps with that. What information? First and foremost: which function is currently being executed and which instruction to execute next, once the current function is fully executed. This piece of information, which instruction to fetch after the current function, is called the *return address*. It’s where the CPU *returns to* after executing the current function. Without this the CPU would just increment the program counter and execute the instruction at the next higher address in memory. And that might be the absolute opposite of what should happen. Instructions are not laid out in memory in the order of execution, next to each other. Imagine what would happen if all the return statements in your Go code would vanish – *that*’s why the CPU needs to keep track of the return addresses. The *call stack* also helps to save execution-relevant data that’s local to functions: the arguments of the function call and the local variables only used in the function.

The return address, arguments and local variables – we could theoretically save this information in any other accessible part of memory, in any way we see fit. But, as it turns out, using *a stack* for this is perfect, because function calls are often nested. When entering a function, the data is pushed on to the stack. And while executing the current function, there is no need to access the local data of the outer, calling function. It’s enough to just access the top part of the stack, so to say. And once the current function returns, the local data is simply popped off – there’s no need for it anymore. That leaves us with the local data of the outer function on the top of the stack. Neat, right?

So that’s why we need the *call stack* and why it’s implemented as *a stack*. The only question now is: why the notorious name? Why is it *the Stack* and not just “well, I guess, yeah, it’s a stack, huh”? Because using this region of memory to implement *a call stack* is a convention so strongly held and widespread that by now it’s been cast into hardware. Certain CPUs support instructions solely for pushing and popping things on the stack. Every program that’s running on them makes use of *this memory region* in *this way* to implement *this mechanism*. There is no way around it. That’s why it’s *the stack* and not just any stack.

But keep in mind that the concept of *a call stack* is just that, a concept. It’s not bound to a specific implementation with a specific memory region. One could implement a call stack in any other place in memory – but without hardware or operating-system support then. In fact, that’s what we’re going to do. We’re going to implement our own call stack, a virtual call stack. But before we do that and switch over from the physical to the virtual, we need to look at one more concept to be fully prepared.

Now that you know how *the stack* works, you can imagine how often the CPU needs to access this region of memory while executing a program. It’s a lot. That means that the speed with which the CPU can access memory puts a limit on how fast it can execute programs. And while memory access *is* fast (a CPU can access main memory around a million times while you blink an eye) it’s not instant and still has a cost.

That’s why computers have another place where they can store data: processor registers. Reg- isters are part of the CPU and accessing them is *much* faster than accessing the main memory. Naturally, one might ask, why not store everything in registers? Because there are only a few of them and they can’t hold as much data as main memory, typically only one word per register. A CPU of the x86-64 architecture, for example, has 16 general purpose registers, each holding 64 bits of data.

Registers are used to store data that’s small but frequently accessed. The memory address pointing to the top of *the stack*, for example, is commonly stored in a register – “commonly” being an understatement here. This specific usage of a register is so prevalent that most CPUs

have a single, designated register just for storing this pointer, the so called *stack pointer*. The operands and the results of certain CPU instructions may also be stored in registers. If a CPU would need to add two numbers, both of them would be stored in registers and the result of the addition would end up in one too. That’s not all, though. There are many more use cases for registers. Here’s another one: if there’s a large piece of data frequently being accessed in a program it makes sense to store the address of it to a register so it can be accessed by the CPU really fast. The most important one for us, though, is the *stack pointer*. We’ll meet that one again soon.

And now… take a deep breath, lean back and relax, because: that’s it! Now that we’ve mentioned registers and know what a stack pointer is, we’ve covered everything we need to know about how real machines work. It’s time for us to go abstract, from the physical to the virtual.

#### WHAT IS A VIRTUAL MACHINE?

Let me get straight to the point: a virtual machine is a computer built with software. It’s a software entity that mimics how a computer works. I’ll admit, “software entity” doesn’t say much, but I used this phrase on purpose, to drive home one point: a virtual machine can be anything. A function, a struct, an object, a module, or even a whole program. What matters is what it does.

A virtual machine has a run loop that goes through the fetch-decode-execute cycle, just like a computer. It has a program counter; it fetches instructions; it decodes and executes them. It also has a stack, just like a real computer. Sometimes it has a call stack and sometimes even registers. All built in software.

Let me put code where my mouth is. Here is a virtual machine in 50 lines of JavaScript:

**let** virtualMachine = **function**(program) {

**let** programCounter = 🅓;

**let** stack = [];

**let** stackPointer = 🅓;

**while** (programCounter < program.length) {

**let** currentInstruction = program[programCounter];

**switch** (currentInstruction) {

**case** PUSH:

stack[stackPointer] = program[programCounter+1]; stackPointer++;

programCounter++;

**break**;

**case** ADD:

right = stack[stackPointer-1] stackPointer--;

left = stack[stackPointer-1] stackPointer--;

stack[stackPointer] = left + right; stackPointer++;

**break**;

**case** MINUS:

right = stack[stackPointer-1] stackPointer--;

left = stack[stackPointer-1]

stackPointer--;

stack[stackPointer] = left - right; stackPointer++;

**break**;

}

programCounter++;

}

console.log("stacktop: ", stack[stackPointer-1]);

}

It has a programCounter, it has a stack and a stackPointer. It has a run loop that executes as long as there are instructions in the program left to execute. It fetches the current instruction pointed to by the programCounter, decodes it and executes it. One iteration of this loop is one “cycle” of this virtual machine.

We can build a program for it and execute it like this:

**let** program = [ PUSH, 3,

PUSH, 4, ADD, PUSH, 5, MINUS

];

virtualMachine(program);

Can you already recognize the expression encoded in these instructions? It’s this:

(3 + 4) - 5

If you didn’t, that’s fine. You’ll be able to soon enough. The program is not that hard to read once you get used to doing arithmetic on a stack: first PUSH 3 and 4, then ADD the two topmost elements by popping them off the stack, adding them and pushing the result back on to the stack; finally, PUSH 5 to get the 5 on to the stack, then MINUS it with the second element from the top and leave the result on to the stack.

The result that ends up on top of the virtualMachine’s stack is printed once the machine finishes its run loop:

$ node virtual\_machine.js stacktop: 2

Boom.

Now, while that’s an actual working virtual machine, it’s also a rather simple one. As you can imagine, it’s not a showcase for what the whole spectrum of virtual-machine implementations is capable of and looks like. You can write a virtual machine in around 50 lines of code, like we just did, but also in 50 thousand lines or more. And going from *here* to *there* means making all kinds of choices regarding functionality and performance.

One of the most significant design decisions is whether the machine will be a *stack machine* or a *register machine*. It’s so significant that virtual machines are grouped based on this architectural choice, just like programming languages are sorted into “compiled” or “interpreted” buckets. The difference between a stack and a register machine is – put in the most simple terms – whether the machine uses a stack to do its computations (like we did in our example above) or

registers (virtual ones!). The debate’s still open on what’s the better (read: faster) choice, since it’s mostly about trade-offs and which ones you’re prepared to make.

A stack machine and a matching compiler are said to be easier to build. The machine needs fewer parts and the instructions it executes are simpler, since they “only” make use of *the stack*. The problem is that you need to execute a lot of instructions, because you need to push and pop everything on and off the stack in order to get anything done. This puts a limit on how far one can take the cardinal rule of performance optimization: instead of trying to do a lot faster, first try to do less.

Building a register machine is more work, because the registers are *an addition*; it still has a stack. It’s not as prominent as in a stack machine, but it’s still necessary to implement a call stack. The advantage of a register machine is that its instructions can make use of the registers and are thus much denser compared to their stack counterparts. Instead of putting things on the stack, pushing and popping them to get them in the correct order, instructions can refer to the registers directly. Generally speaking, a program needs less instructions on a register machine than on a stack machine. That in turn results in better performance. But then again, writing the compiler that produces such dense instructions takes more effort. As I said: it’s about making trade-offs.

Besides this main architectural choice there’s a myriad of other decisions that go into building a virtual machine. There are the big questions regarding how to use memory and how to represent values internally (a topic which we already touched upon when we built the Monkey object system for our evaluator). But then there are seemingly tiny things that turn out to be endless, twisted rabbit holes you can get lost in. Let’s take a peek into one.

In our example above we used a switch statement do the *dispatching* in the run loop of our machine. *Dispatching* in a virtual machine means selecting an implementation for an instruction before executing it. In our switch statement the implementation of these instructions can be found right next to the case, inline. In case MINUS we subtract two values, in case ADD we add them, and so on. That’s dispatching. And while a switch statement looks like the obvious and *only* choice it’s far from that.

A switch statement is just the opening of the rabbit hole. And when you’re looking for maximum performance, you have to go in all the way. In there, you’ll find dispatching done with [jump](https://github.com/Shopify/go-lua/blob/88a6f168eee0ba102d7d20c5281056a5dd3d7550/vm.go#L306) [tables](https://github.com/Shopify/go-lua/blob/88a6f168eee0ba102d7d20c5281056a5dd3d7550/vm.go#L306), with [computed GOTO statements](https://eli.thegreenplace.net/2012/07/12/computed-goto-for-efficient-dispatch-tables), with [indirect and direct threaded code](https://www.complang.tuwien.ac.at/forth/threaded-code.html). Because, believe it or not, with a suﬀicient number of case branches (a few hundred or more) a switch might be the slowest of these solutions. The goal is to reduce the overhead of dispatching so much that from a performance perspective the fetch-decode part of a fetch-decode-execute cycle disappears. That should give you a taste of how deep the rabbit holes are.

We now have a rough overview of what a virtual machine is and what goes into building one. Don’t worry if you’re still missing some details. Since we’re going to build our own, we’re going to revisit a lot of the topics, ideas and, yes, the rabbit holes.

##### WHy BUILD ONE?

Let’s put what we just learned into perspective. *Why* would you build a virtual machine to implement a programming language? I have to admit that this is the question I’ve carried around with me the longest. Even after I had built a few tiny virtual machines and had read through the source code of larger ones, I still asked myself: why?

When we implement a programming language we want it to be universal. It should be able to execute all possible programs and not just – as one example of many – functions we built into it. Universal computation is what we’re after and computers offer a solid model for it.

If we construct a programming language based on that model, it’s going to have the same computational capabilities as the computer. It’s also one of the fastest ways to execute programs.

But if executing programs like a computer is the best and fastest way, why not, you know, just let the computer execute the programs? Portability. We could write a compiler for our programming language that allows us to execute the translated programs natively on a computer. These programs would be really fast. But we would also have to write a new compiler for every computer architecture we want to run our programs on. That’s a lot of work. Instead, we can translate our programs into instructions for a virtual machine. And the virtual machine itself runs on as many architectures as its implementation language. In the case of the Go programming language that’s *pretty* portable.

And there’s another reason for implementing a programming language with a virtual machine, which I think is the most fascinating one. It’s this: virtual machines are domain specific. That’s what makes them different from their non-virtual counterparts. Computers offer us a universal solution to all of our computational needs and are decidedly *not* domain specific. And that’s okay, that’s exactly what we need from a computer on which we want to run all sorts of programs. But what if we don’t need a machine to be *that* universal? What if we only need a subset of the features a computer has to offer to its programmers?

As programmers we know that no feature comes without cost. Increased complexity and de- graded performance are just two of the prices we regularly pay for them. Computers today have a lot of features. A CPU of the x86-64 family supports between [900 and 4000 instructions,](https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/) [depending on how you count them.](https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/) That includes at least six different ways to do a bitwise XOR on two operands. This is handy and what makes computers so versatile. But it’s not free. The versatility has a cost, just like every other feature. Think back to the switch statement in our tiny VM and take a second to ponder the performance implications of adding 3997 more case branches. If you’re not sure whether it will actually be slower, ask yourself how easy it is to maintain the code or to program for this VM. The good news is that we can turn this around. If we get rid of the features we don’t need, we can go faster. Less complexity, less maintenance, less mass. This is where virtual machines come into play.

A virtual machine is like a custom-built computer. It has custom parts and a custom machine language. It’s tuned to a single programming language. All the unneeded features are stripped off and what’s left is highly specialized. Since you don’t need to be as universal as a general- purpose computer, you can focus. You can concentrate your efforts into making this highly- specialized and custom-made machine work as best and go as fast as possible. This – the specialization, the domain specificity – is as important as getting rid of the unneeded mass.

Why this is so important becomes even clearer when we take a look at the instructions a virtual machine executes, something which we steered clear of up until now. Remember what we fed our tiny virtual machine? Here it is again:

**let** program = [ PUSH, 3,

PUSH, 4, ADD, PUSH, 5, MINUS

];

virtualMachine(program);

Now, what *is* that? What’s PUSH, what’s ADD, what’s MINUS? Here are their definitions:

**const** PUSH = 'PUSH';

**const** ADD = 'ADD';

**const** MINUS = 'MINUS';

PUSH, ADD and MINUS are just constants referring to strings. There’s no magic sauce. What a letdown. Boo! The silver lining of this revelation is that these definitions are as toy like and for illustration purposes only as the rest of the VM. They don’t answer the larger, more interesting question looming here: what *exactly* do virtual machines execute?

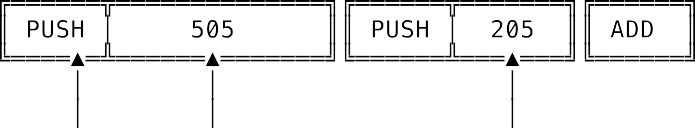
##### ByTECODE

Virtual machines execute bytecode. Like the machine code that computers execute, bytecode is made up of instructions that tell the machine what to do. Push this, pop that, add these, call this function. It’s called bytecode because the opcodes contained in each instruction are one byte in size.

An “opcode” is the “operator” part of an instruction, sometimes also called “op”. The PUSH we’ve seen earlier is such an opcode, except that in our toy example it was a multi-byte string and not just one byte. In a proper implementation PUSH would just be the name that refers to an opcode, which itself is one byte wide. These names, like PUSH or POP, are called mnemonics. They help us programmers remember and talk about opcodes.

The operands (also called arguments or parameters) to these opcodes are also contained in the bytecode. They’re placed alongside each other, with the operands following the opcodes. The operands, though, aren’t necessarily one byte wide. For example, if an operand is an integer and greater than 255, it would take multiple bytes to represent it. Some opcodes have multiple operands, some just one and some don’t have any at all. Whether the bytecode is designed for a register or a stack machine has a huge influence here.

You can imagine bytecode as a sequence of opcodes and operands, laid out in memory next to each other:





This helps to illustrate the general idea, but bytecode is a binary format and not nearly as readable. Meaning that you can’t read it like a text file. The mnemonics, like PUSH, don’t show up in the actual bytecode. They are replaced by the opcodes they refer to, and those are just numbers – bytes. Which numbers exactly is up to the person defining the bytecode. The PUSH mnemonic might stand for the number 🅓 and POP might refer to the number 23.

The operands are also encoded and, again, it depends on its value into how many bytes. In case an operand needs multiple bytes to be accurately represented, the order in which it’s encoded plays a big role. There are two possible orders, called *little endian* and *big endian*. Little endian means that the *least* significant byte of the original data comes first and is stored in the lowest memory address. Big endian is the opposite: the *most* significant byte comes first.

If we, as bytecode designers, were to declare that PUSH refers to 1, ADD to 2 and integers are stored in big endian, we could encode the example from above and lay it out in memory like this:



What we just did – taking a human-readable representation of bytecode and turning it into binary data – is done by programs called assemblers. You’ve probably heard of them in the context of non-virtual machine code. Same thing here. Assembly language is the readable version of the bytecode, containing mnemonics and readable operands, and an assembler turns it into binary bytecode. The reverse, transforming the binary representation into a readable representation, is done by disassemblers.

Alright! So much for the purely technical aspects of bytecode. Any further exploration would get too specific too fast. Bytecode formats are just too diverse and specialized to allow us to make more general statements here. Just like the virtual machine that executes it, bytecode is created with a very specific goal in mind.

Bytecode is a domain-specific language for a domain-specific machine. It’s the bespoke machine language for the custom-built virtual machine. That’s where its power lies. Instead of being universal and supporting every conceivable use case, bytecode can be specialized. It only has to support the features needed by the source language, the language being compiled to bytecode.

And not only that. On top of only allowing a narrow set of instructions, it can contain domain- specific instructions that only make sense in the context of the domain-specific virtual machine. The bytecode for the Java Virtual Machine (JVM), for example, contains these instructions: invokeinterface to invoke an interface method, getstatic to get a static field of a class, new to create a new object of the specified class. Ruby’s bytecode has the putself instruction to load self on the stack, send to send a message to an object, putobject to load any object on to the stack. And Lua’s bytecode has dedicated instructions for accessing and manipulating tables and tuples. You won’t find any of that in the instruction set of a general-purpose x86-64 CPU.

This ability to specialize by using a custom bytecode format is one of the biggest reasons for building a virtual machine in the first place. Not only do compilation, maintenance and debugging get easier but the resulting code is also denser, because it takes less instructions to express something. That in turn makes the code faster to execute.

Now, if all that talk about custom virtual machines, tailor-made machine code, hand-built compilers didn’t wet your appetite, this is your last chance to turn around. We’re about to get our hands dirty.

### WHAT WE’RE GOING TO DO, OR: THE DUALITy OF VM AND COMPILER

Building a virtual machine and a matching compiler requires us first to solve a variation of the chicken-and-egg problem: which one do we build first? The compiler, that outputs bytecode for a machine that doesn’t exist yet? Or the virtual machine, that no one produces any code for?

Here’s the answer I choose for this book: we are going to build both – at the same time.

Building one completely before the other (and it doesn’t matter which one) is frustrating. It’s hard to understand what’s going on and what is the purpose of what you’re doing. If you’re building the compiler and defining the bytecode first, it’s tough to make sense of why things are the way they are without knowing how the virtual machine will later execute it. Building the virtual machine before the compiler comes with its own challenges, because the bytecode needs to be defined first. That’s hard to do without closely looking at the source-language constructs the bytecode aims to represent, which means you’re going to spell out the compiler anyway.

Of course, if you already have experience with building one or the other, you know where you want to end up and can thus choose either option. For us, though, the goal is to learn how to build both from the ground up.

That’s why we’re going to start small. We’re going to build a tiny virtual machine that only supports a tiny number of instructions and a matching tiny compiler that only knows how to output these instructions. That allows us to immediately see why we’re building what we’re building and how all the pieces fit together. We’ll also have a running system right from the start. That gives us fast feedback cycles and allows us to tune, experiment and gradually build up our virtual machine and compiler. It also makes the whole journey a lot of fun.

Now you know what the plan is. And you also know enough about compilers and virtual machines that we don’t get lost along the way. Let’s get to it.

CHAPTER 2

HELLO ByTECODE!

Our goal for this chapter is to compile and execute this Monkey expression:

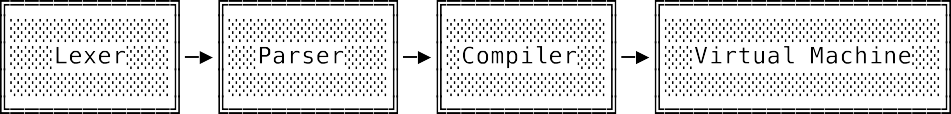
1 + 2

That doesn’t sound like an ambitious goal, but in order to reach it, we will have to learn many new things and build up a lot of the infrastructure we’re going to use in the upcoming chapters. And by choosing the simple expression 1 + 2 we won’t be distracted by the Monkey code itself and how it should work. We can concentrate on our compiler and virtual machine.

At the end of this chapter we want to be able to:

* take the Monkey expression 1 + 2
* tokenize and parse it using our existing lexer, token and parser packages
* take the resulting AST, whose nodes are defined in our ast package
* pass it to the newly-built compiler, which compiles it to bytecode
* take the bytecode and hand it over to the also newly-built virtual machine which will execute it
* make sure that the virtual machine turned it into 3.

The 1 + 2 expression will travel through all the major parts of our new interpreter:



In terms of data structures, you can see that there will be quite a few transformations until we end up with the 3 as our result:



Since we’ll be using a lot of the packages we built in the previous book, we can already handle everything up to the AST. After that we’re entering unchartered territory. We need to define bytecode instructions, build a compiler and construct a virtual machine – just to turn 1 + 2 into 3. Sounds daunting? Worry not, we’ll do this step by step and build from the ground up, as always, and start with the bytecode.

## FIRST INSTRUCTIONS

As I mentioned in the previous chapter, the architecture of the virtual machine is the single- biggest influence on what the bytecode looks like. That means we need to decide what type of machine we’re going to build before we can start to specify bytecode.

So without further ado, let’s pull back the curtain: we’re going to build a **stack machine**! Why? Because stack machines are far easier to understand and to build for beginners than register machines. Less concepts, less moving parts. And all the performance considerations – *is a register machine faster?* – do not play a huge role for us. Our priorities are learning and understanding.

Later on we’ll see more of the implications this decision has, but the immediate and most practical one is that we now have to do stack arithmetic. That means, in order to reach our declared goal of compiling and executing the Monkey expression 1 + 2, we have to translate it to bytecode instructions that make use of a stack. The stack is where a stack machine does its work – we can’t just tell it to add two numbers, without making use of the stack.

Thankfully, we saw a similar example before and already know how to do arithmetic with a stack. We first get the operands 1 and 2 on to the stack and then tell the VM: “add these!”. This “add these!” instruction should then cause the VM to take the two topmost elements from the stack, add them together and push the result back on to the stack. Here’s what stack should look like before and after the instruction:



So in order to fully implement this we need to tell the VM:

* Push 1 on to the stack
* Push 2 on to the stack
* Add the two topmost elements together

We need to create three instructions. As programmers, though, we know that we only need to define two separate instruction types, since pushing 2 on to the stack should be the same as pushing 1, except that the “argument” is different. So, two instruction types in total: one for pushing something on to the stack and one for adding things that are already on the stack.

We’ll implement both in the same way. First we define their opcodes and how they are encoded in bytecode. Then we extend the compiler so it can produce these instructions. As soon as the compiler knows how to do that, we can create the VM that decodes and executes them. And we start with the instructions that tell the VM to push something on to the stack.

##### STARTING WITH ByTES

Here we are. We need to define our first bytecode instruction. How do we do that? Well, since creating definitions while programming is not much more than telling the computer what we know, let’s ask ourselves: what do we know about bytecode?

We know that it’s made up of instructions. And we also know that the instructions themselves are a series of bytes and a single instruction consists of an opcode and an optional number of operands. An opcode is exactly one byte wide, has an arbitrary but unique value and is the first byte in the instruction. Looks like we know quite a lot and the best thing is, that this is precise enough to be turned into code – literally.

As our first oﬀicial practical act of coding in this book, we create a new package, called code, where we start to define our Monkey bytecode format:

*// code/code.go*

**package** code

**type** Instructions []byte

**type** Opcode byte

Instructions is a slice of bytes and an Opcode is a byte. Perfect, they match our descriptions in prose pretty well. But there are two definitions missing here.

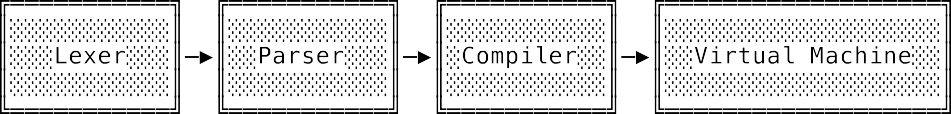
The first one is Instruction – singular. Why didn’t we define it here as []byte? Because it’s far more handy to pass around and work with a []byte and treat it *implicitly* as an instruction than to encode this definition in Go’s type system. You’ll see soon enough how often we’re going to use []byte and how cumbersome type assertions and type casting from and to an Instruction type would be there.

The other missing definition is one for Bytecode. There should at least be some definition of bytecode that tells us it’s made up of instructions, right? The reason for its absence is we’d run into a nasty import-cycle if we were to define Bytecode here in the code package. But it won’t be missing for too long. Once we get to the compiler, we’ll define it there – in the compiler’s package.

Now that we have definitions for Opcode and Instructions, we can define our first opcode, the one that tells the VM to push something on the stack. And here’s a surprise: the opcode won’t have “push” in its name. In fact, it won’t be solely about pushing things. Allow me to explain.

We said earlier that when we compile the Monkey expression 1 + 2, we want to generate three different instructions; two of which tell the VM to push 1 and 2 on to the stack. A first instinct might tell us to implement these by defining a “push”-instruction with an integer as its operand with the idea being that the VM then takes the integer operand and pushes it on to the stack. And for integers that would work fine, because we could easily encode them and put them directly into the bytecode. But what if later on we wanted to push other things contained in Monkey code? String literals, for example. Putting those into the bytecode is also possible, true, since it’s just made of bytes after all, but it would also be a lot of bloat and would sooner or later become unwieldy.

That’s where the idea of *constants* come into play. In this context, “constant” is short for “constant expression” and refers to expressions whose value doesn’t change, is *constant*, and can be determined at *compile time*:





That means we don’t need to run the program to know what these expressions evaluate to. A compiler can find them in the code and store the value they evaluate to. After that, it can reference the constants in the instructions it generates, instead of embedding the values directly in them. And while “reference” sounds like a special data type, it’s far easier than that. A plain integer does the job just fine and can serve as an index into a data structure that holds all constants, often called a constant pool.

And that’s exactly what our compiler is going to do. When we come across an integer lit- eral (a constant expression) while compiling, we’ll evaluate it and keep track of resulting

\*object.Integer by storing it in memory and assigning it a number. In the bytecode instruc- tions we’ll refer to the \*object.Integer by this number. After we’re done compiling and pass the instructions to the VM for execution, we’ll also hand over all the constants we’ve found by putting them in a data structure – our constant pool – where the number that has been assigned to each constant can be used as an index to retrieve it.

Back to our first opcode. It’s called OpConstant and it has one operand: the number we pre- viously assigned to the constant. When the VM executes OpConstant it retrieves the constant using the operand as an index and pushes it on to the stack. Here’s out first opcode definition:

*// code/code.go*

*// [...]*

**const** (

OpConstant Opcode = iota

)

While this looks exactly like the meager three lines of code that they are, this addition is the groundwork for all future Opcode definitions. Each definition will have an Op prefix and the value it refers to will be determined by iota. We let iota generate increasing byte values for us, because we just don’t care about the actual values our opcodes represent. They only need to be distinct from each other and fit in one byte. iota makes sure of that for us.

What’s missing from this definition is the part that says OpConstant has one operand. There’s no technical reason for writing this down, since we could share this piece of knowledge implicitly between compiler and VM. For debugging and testing purposes, though, it’s handy being able to lookup how many operands an opcode has and what its human-readable name is. In order to achieve that, we’ll add proper definitions and some tooling to the code package:

*// code/code.go*

**import** "fmt"

**type** Definition **struct** { Name string

OperandWidths []int

}

**var** definitions = **map**[Opcode]\*Definition{ OpConstant: {"OpConstant", []int{2}},

}

**func** Lookup(op byte) (\*Definition, error) { def, ok := definitions[Opcode(op)]

**if** !ok {

**return** nil, fmt.Errorf("opcode %d undefined", op)

}

**return** def, nil

}

The Definition for an Opcode has two fields: Name and OperandWidths. Name helps to make an

Opcode readable and OperandWidths contains the number of bytes each operand takes up.

The definition for OpConstant says that its only operand is two bytes wide, which makes it an uint16 and limits its maximum value to 65535. If we include 🅓 the number of representable values is then 65536. That should be enough for us, because I don’t think we’re going to reference more than 65536 constants in our Monkey programs. And using an uint16 instead of, say, an uint32, helps to keep the resulting instructions smaller, because there are less unused bytes.

With this definition in place we can now create our first bytecode instruction. Without any operands involved that would be as simple as adding an Opcode to an Instructions slice. But in the case of OpConstant we need to correctly encode the two-byte operand.

For that we’ll now create a function that allows us to easily create a single bytecode instruction that’s made up of an Opcode and an optional number of operands. We’ll call it Make, which gives us the pretty great identifier code.Make in other packages.

And here’s what we’ve been waiting for, the first test of this book, showing what we want Make

to do:

*// code/code\_test.go*

**package** code

**import** "testing"

**func** TestMake(t \*testing.T) { tests := []**struct** {

op Opcode operands []int expected []byte

}{

{OpConstant, []int{65534}, []byte{byte(OpConstant), 255, 254}},

}

**for** \_, tt := **range** tests {

instruction := Make(tt.op, tt.operands...)

**if** len(instruction) != len(tt.expected) { t.Errorf("instruction has wrong length. want=%d, got=%d",

len(tt.expected), len(instruction))

}

**for** i, b := **range** tt.expected {

**if** instruction[i] != tt.expected[i] {

t.Effoff("wrong byte at pos %d. want=%d, got=%d",

i, b, instruction[i])

}

}

}

}

Don’t be put off by tests only containing one test case. We’ll extend it later on when we add more Opcodes to our code vocabulary.

For now, we only pass OpConstant and the operand 65534 to Make. We then expect to get back a []byte holding three bytes. Of these three, the first one has to be the opcode, OpConstant, and the other two should be the big-endian encoding of 65534. That’s also why we used 65534 and not the maximum value 65535: this way we can check that the most significant byte comes first. 65534 will be encoded in big endian as the byte sequence 🅓xFF 🅓xFE and 65535 would be encoded as 🅓xFF 🅓xFF – hard to recognize an order.

Since Make doesn’t exist yet, the test does not fail, but fails to compile, so here’s the first version of Make:

*// code/code.go*

**import** (

"encoding/binary" "fmt"

)

**func** Make(op Opcode, operands ...int) []byte { def, ok := definitions[op]

**if** !ok {

**return** []byte{}

}

instructionLen := 1

**for** \_, w := **range** def.OperandWidths { instructionLen += w

}

instruction := make([]byte, instructionLen) instruction[🅓] = byte(op)

offset := 1

**for** i, o := **range** operands { width := def.OperandWidths[i] **switch** width {

**case** 2:

binary.BigEndian.PutUint16(instruction[offset:], uint16(o))

}

offset += width

}

**return** instruction

}

And that’s how you make bytecode.

The first thing we’re doing here is to find out how long the resulting instruction is going to be. That allows us to allocate a byte slice with the proper length. Note that we don’t use the Lookup function to get to the definition, which gives us a much more usable function signature for Make in the tests later on. By circumventing Lookup and not having to return possible errors,

we can use Make to easily build up bytecode instructions without having to check for errors after every call. The risk of producing empty byte slices by using an unknown opcode is one we’re willing to take, since we’re on the producing side here and know what we’re doing when creating instructions.

As soon as we have the final value of instructionLen, we allocate the instruction []byte and add the Opcode as its first byte – by casting it into one. Then comes the tricky part: we iterate over the defined OperandWidths, take the matching element from operands and put it in the instruction. We do that by using a switch statement with a different method for each operand, depending on how wide the operand is.

As we define additional Opcodes, we soon have to extend this switch statement. For now, we only make sure that a two-byte operand is encoded in big endian. And while it’s not hard to do that by hand, we use binary.BigEndian.PutUint16 from the standard library for this, with the benefit of having the name of the encoding immediately visible.

After encoding the operand, we increment offset by its width and the next iteration of the loop. Since the OpConstant opcode in our test case has only one operand, the loop performs only one iteration before Make returns instruction.

And, would you look at that, our fist test is compiling and passing:

$ go test ./code

ok monkey/code 🅓.🅓🅓7s

We successfully turned OpConstant and the operand 65534 into three bytes. That means we created our first bytecode instruction!

#### THE SMALLEST COMPILER

Now that we have a toolbox called code, we can start working on the compiler. Since we want a system that works from end to end as soon as possible, and not a system that can only be turned on once it’s feature-complete, our goal in this section is to build the smallest possible compiler. It should only do one thing for now: produce two OpConstant instructions that later cause the VM to correctly load the integers 1 and 2 on to the stack.

In order to achieve that, this minimal compiler has to do the following: traverse the AST we pass in, find the \*ast.IntegerLiteral nodes, evaluate them by turning them into \*object.Integer objects, add the objects to the constant pool, and finally emit OpConstant instructions that reference the constants in said pool.

Sounds good? Perfect! Let’s start by defining Compiler and its interface in a new compiler

package:

*// compiler/compiler.go*

**package** compiler

**import** (

"monkey/ast" "monkey/code" "monkey/object"

)

**type** Compiler **struct** { instructions code.Instructions constants []object.Object

}

**func** New() \*Compiler {

**return** &Compiler{

instructions: code.Instructions{}, constants: []object.Object{},

}

}

**func** (c \*Compiler) Compile(node ast.Node) error {

**return** nil

}

**func** (c \*Compiler) Bytecode() \*Bytecode {

**return** &Bytecode{

Instructions: c.instructions, Constants: c.constants,

}

}

**type** Bytecode **struct** { Instructions code.Instructions Constants []object.Object

}

It really is minimal, isn’t it? The Compiler is a small struct with only two fields: instructions and constants. Both are internal fields and will later be modified by the Compile method. instructions will hold the generated bytecode and constants is a slice that serves as our constant pool.

But I bet the thing that caught your eye immediately is the definition we’ve been looking for earlier, in the code package: Bytecode! There it is and it doesn’t need a lot of explanation. It contains the Instructions the compiler generated and the Constants the compiler evaluated.

Bytecode is what we’ll pass to the VM and make assertions about in our compiler tests. Speaking of which, the Compile method is empty and we’re now going to write our first compiler test that tells us what it should do.

*// compiler/compiler\_test.go*

**package** compiler

**import** (

"monkey/code" "testing"

)

**type** compilerTestCase **struct** { input string

expectedConstants []**interface**{} expectedInstructions []code.Instructions

}

**func** TestIntegerArithmetic(t \*testing.T) { tests := []compilerTestCase{

{

input: "1 + 2",

expectedConstants: []**interface**{}{1, 2}, expectedInstructions: []code.Instructions{

code.Make(code.OpConstant, 🅓), code.Make(code.OpConstant, 1),

},

},

}

runCompilerTests(t, tests)

}

**func** runCompilerTests(t \*testing.T, tests []compilerTestCase) { t.Helper()

**for** \_, tt := **range** tests { program := parse(tt.input)

compiler := New()

err := compiler.Compile(program)

**if** err != nil {

t.F6t6lf("compiler error: %s", err)

}

bytecode := compiler.Bytecode()

err = testInstructions(tt.expectedInstructions, bytecode.Instructions)

**if** err != nil {

t.F6t6lf("testInstructions failed: %s", err)

}

err = testConstants(t, tt.expectedConstants, bytecode.Constants)

**if** err != nil {

t.F6t6lf("testConstants failed: %s", err)

}

}

}

What’s happening here doesn’t take long to explain: we take Monkey code as input, we parse it, produce an AST, hand it to the compiler and then make assertions about the bytecode the compiler produced.

We do that by constructing a compilerTestCase in which we define the input, which constants we expect in the constant pool and which instructions we expect the compiler to generate. Then we hand the tests slice with the compilerTestCases to runCompilerTests to run them.

That’s a slightly different approach to constructing tests compared to the first book. The reason for that is Go 1.9, which introduced the wonderful t.Helper method. t.Helper, which we call in runCompilerTests, allows us to remove duplicated logic in test functions by defining test helpers. Think of it as inlining runCompilerTests into TestIntegerArithmetic. That in turn allows to abstract away the common behaviour shared by every compiler test we’re going to write, which greatly reduces the noise in every test function and the page count of this book.

Now, let’s talk about the helpers used in runCompilerTests.

The parse function contains some of the things we built in the first book: the lexer and the parser. We hand it a string and get back an AST:

*// compiler/compiler\_test.go*

**import** (

"monkey/ast"

"monkey/code" "monkey/lexer" "monkey/parser" "testing"

)

**func** parse(input string) \*ast.Program { l := lexer.New(input)

p := parser.New(l)

**return** p.ParseProgram()

}

That’s the prelude. The main part of runCompilerTests revolves around the two fields of the

Bytecode the compiler produced. First, we want to make sure that the bytecode.Instructions

are correct. For that we have the testInstructions helper function:

*// compiler/compiler\_test.go*

**import** (

"fmt"

*// [...]*

)

**func** testInstructions(

expected []code.Instructions, actual code.Instructions,

) error {

concatted := concatInstructions(expected)

**if** len(actual) != len(concatted) {

**return** fmt.Errorf("wrong instructions length.\nwant=%q\ngot =%q", concatted, actual)

}

**for** i, ins := **range** concatted {

**if** actual[i] != ins {

**return** fmt.Errorf("wrong instruction at %d.\nwant=%q\ngot =%q", i, concatted, actual)

}

}

**return** nil

}

As you can see, it uses another helper called concatInstructions:

*// compiler/compiler\_test.go*

**func** concatInstructions(s []code.Instructions) code.Instructions { out := code.Instructions{}

**for** \_, ins := **range** s {

out = append(out, ins...)

}

**return** out

}

We need concatInstructions because the expectedInstructions field in compilerTestCase is

not just a slice of bytes, but a slice of *slices* of bytes. And that’s because we use code.Make to generate the expectedInstructions, which produces a []byte. So in order to compare the expectedInstructions with the actual instructions, we need to turn the slice of slices into a flattened slice by concatenating the instructions.

The other helper used by runCompilerTests is testConstants, which resembles a lot of the test helpers we used in our evaluator package back in the first book:

*// compiler/compiler\_test.go*

**import** (

*// [...]*

"monkey/object"

*// [...]*

)

**func** testConstants( t \*testing.T,

expected []**interface**{}, actual []object.Object,

) error {

**if** len(expected) != len(actual) {

**return** fmt.Errorf("wrong number of constants. got=%d, want=%d", len(actual), len(expected))

}

**for** i, constant := **range** expected { **switch** constant := constant.(**type**) { **case** int:

err := testIntegerObject(int64(constant), actual[i])

**if** err != nil {

**return** fmt.Errorf("constant %d - testIntegerObject failed: %s", i, err)

}

}

}

**return** nil

}

There’s a lot of noise here, but what’s happening here is not complicated. testConstants iterates through the expected constants and compares them with the actual constants the compiler produced. The switch statement is a sign of things to come. We will extend it with new case branches as soon as we expect more than integers to end up in the constant pool. For now it only uses one other helper, testIntegerObject, which is a nearly-identical replica of the testIntegerObject we used in our evaluator tests:

*// compiler/compiler\_test.go*

**func** testIntegerObject(expected int64, actual object.Object) error { result, ok := actual.(\*object.Integer)

**if** !ok {

**return** fmt.Errorf("object is not Integer. got=%T (%+v)", actual, actual)

}

**if** result.Value != expected {

**return** fmt.Errorf("object has wrong value. got=%d, want=%d",

result.Value, expected)

}

**return** nil

}

That’s all there is to TestIntegerArithmetic. The test itself is not complex, but establishes how we will write compiler tests in the future by bringing with it a lot of different test helpers. It looks like a lot of code for such a small test, but I promise you that we’ll get a lot of mileage out of this test setup.

Now, how does the test itself do? Well, not so good:

$ go test ./compiler

--- FAIL: TestIntegerArithmetic (🅓.🅓🅓s)

compiler\_test.go:31: testInstructions failed: wrong instructions length. want="\x🅓🅓\x🅓🅓\x🅓🅓\x🅓🅓\x🅓🅓\x🅓1"

got ="" FAIL

FAIL monkey/compiler 🅓.🅓🅓8s

But considering that we didn’t write any code for the compiler yet, except defining its interface, that’s not so bad, is it? What’s bad though is the output:

want="\x🅓🅓\x🅓🅓\x🅓🅓\x🅓🅓\x🅓🅓\x🅓1"

No one looks at that and goes “Ah, I see…” I know that you’re anxious to get that compiler running and humming, but I can’t let this unreadable gibberish stand. I mean, it’s correct, those *are* the bytes we want, printed in hexadecimal, but it’s just not helpful. And believe me, soon enough this output would drive us nuts. So before we start filling out the compiler’s Compile() method, we’re going to invest in our developer happiness and teach our code.Instructions how to properly print themselves.

##### ByTECODE, DISASSEMBLE!

You can teach types to print themselves in Go by giving them a String() method. That also holds true for bytecode instructions. It’s pretty easy to do, actually, but, as you already know, we wouldn’t print anything without writing a test for it, right?

*// code/code\_test.go*

**func** TestInstructionsString(t \*testing.T) { instructions := []Instructions{

Make(OpConstant, 1),

Make(OpConstant, 2),

Make(OpConstant, 65535),

}

expected := `🅓🅓🅓🅓 OpConstant 1

🅓🅓🅓3 OpConstant 2

🅓🅓🅓6 OpConstant 65535

`

concatted := Instructions{}

**for** \_, ins := **range** instructions { concatted = append(concatted, ins...)

}

**if** concatted.String() != expected {

t.Effoff("instructions wrongly formatted.\nwant=%q\ngot=%q", expected, concatted.String())

}

}

That’s what we expect from the to-be-implemented Instructions.String method: nicely- formatted multi-line output that tells us everything we need to know. There’s a counter at the start of each line, telling us which bytes we’re looking at, there are the opcodes in their human-readable form, and then there are the decoded operands. A lot more pleasant to look at than \x🅓🅓\x🅓🅓\x🅓🅓\x🅓🅓\x🅓🅓\x🅓1, right? We could also name the method MiniDisassembler instead of String because that’s what it is.

The test won’t compile, because the String method is undefined. So here’s the first piece of code we need to add:

*// code/code.go*

**func** (ins Instructions) String() string {

**return** ""

}

Correct, we return a blank string. Why? Because that gives the compiler something to chew on and us the ability to run tests again:

$ go test ./code

--- FAIL: TestInstructionsString (🅓.🅓🅓s) code\_test.go:49: instructions wrongly formatted.

want="🅓🅓🅓🅓 OpConstant 1\n🅓🅓🅓3 OpConstant 2\n🅓🅓🅓6 OpConstant 65535\n" got=""

FAIL

FAIL monkey/code 🅓.🅓🅓8s

Perfect, it fails. That’s a lot more useful to us than an undefined: String compiler error that stops us from running the tests, because we now need to write *another* test and run it.

This other test is for a function that will be the heart of Instructions.String. Its name is

ReadOperands and here’s what we want it to do:

*// code/code\_test.go*

**func** TestReadOperands(t \*testing.T) { tests := []**struct** {

op Opcode

operands []int bytesRead int

}{

{OpConstant, []int{65535}, 2},

}

**for** \_, tt := **range** tests {

instruction := Make(tt.op, tt.operands...)

def, err := Lookup(byte(tt.op))

**if** err != nil {

t.F6t6lf("definition not found: %q\n", err)

}

operandsRead, n := ReadOperands(def, instruction[1:])

**if** n != tt.bytesRead {

t.F6t6lf("n wrong. want=%d, got=%d", tt.bytesRead, n)

}

**for** i, want := **range** tt.operands {

**if** operandsRead[i] != want {

t.Effoff("operand wrong. want=%d, got=%d", want, operandsRead[i])

}

}

}

}

As you can see, ReadOperands is supposed to be Make’s counterpart. Whereas Make encodes the operands of a bytecode instruction, it’s the job of ReadOperands to decode them.

In TestReadOperands we Make a fully-encoded instruction and pass its definition to ReadOperands, along with the subslice of the instruction containing the operands. ReadOperands should then return the decoded operands and tell us how many bytes it read to do that. As you can probably imagine by now, we’re going to extend the tests table as soon as we have more opcodes and different instruction types.

The test fails because ReadOperands is not defined:

$ go test ./code # monkey/code

code/code\_test.go:71:22: undefined: ReadOperands FAIL monkey/code [build failed]

In order to get it to pass we have to implement a ReadOperands function that reverses everything

Make did:

*// code/code.go*

**func** ReadOperands(def \*Definition, ins Instructions) ([]int, int) { operands := make([]int, len(def.OperandWidths))

offset := 🅓

**for** i, width := **range** def.OperandWidths {

**switch** width {

**case** 2:

operands[i] = int(ReadUint16(ins[offset:]))

}

offset += width

}

**return** operands, offset

}

**func** ReadUint16(ins Instructions) uint16 {

**return** binary.BigEndian.Uint16(ins)

}

Just like in Make, we use the \*Definition of an opcode to find out how wide the operands are and allocate a slice with enough space to hold them. We then go through the Instructions slice and read in and convert as many bytes as defined in the definition. And again: the switch statement will be extended soon.

Let me explain why ReadUint16 is a separate, public function. In Make we did the encoding of operands to bytes inline. Here, though, we want to expose the function so it can be used directly by the VM, allowing us to skip the definition lookup required by ReadOperands.

We now have one less failing test and can start to unwind and go back to the failing tests that brought us here. The first one is TestInstructionString, which is still chewing on the blank string:

$ go test ./code

--- FAIL: TestInstructionsString (🅓.🅓🅓s) code\_test.go:49: instructions wrongly formatted.

want="🅓🅓🅓🅓 OpConstant 1\n🅓🅓🅓3 OpConstant 2\n🅓🅓🅓6 OpConstant 65535\n" got=""

FAIL

FAIL monkey/code 🅓.🅓🅓8s

Now that we have ReadOperands, we can get rid of the blank string and properly print instruc- tions:

*// code/code.go*

**import** (

"bytes"

*// [...]*

)

**func** (ins Instructions) String() string {

**var** out bytes.Buffer

i := 🅓

**for** i < len(ins) {

def, err := Lookup(ins[i])

**if** err != nil {

fmt.Fprintf(&out, "ERROR: %s\n", err)

**continue**

}

operands, read := ReadOperands(def, ins[i+1:])

fmt.Fprintf(&out, "%🅓4d %s\n", i, ins.fmtInstruction(def, operands))

i += 1 + read

}

**return** out.String()

}

**func** (ins Instructions) fmtInstruction(def \*Definition, operands []int) string { operandCount := len(def.OperandWidths)

**if** len(operands) != operandCount {

**return** fmt.Sprintf("ERROR: operand len %d does not match defined %d\n", len(operands), operandCount)

}

**switch** operandCount {

**case** 1:

**return** fmt.Sprintf("%s %d", def.Name, operands[🅓])

}

**return** fmt.Sprintf("ERROR: unhandled operandCount for %s\n", def.Name)

}

I don’t think I have to explain to you how this works because we’ve seen variations of this going-through-a-byte-slice mechanism a few times now. The rest is string formatting. But here’s something worth looking at:

$ go test ./code

ok monkey/code 🅓.🅓🅓8s

The tests in the code package now pass. Our mini-disassembler works. We can unwind even further and rerun the failing compiler test that kicked off this ride through the code package:

$ go test ./compiler

--- FAIL: TestIntegerArithmetic (🅓.🅓🅓s)

compiler\_test.go:31: testInstructions failed: wrong instructions length. want="🅓🅓🅓🅓 OpConstant 🅓\n🅓🅓🅓3 OpConstant 1\n"

got ="" FAIL

FAIL monkey/compiler 🅓.🅓🅓8s

Isn’t that beautiful? Alright, granted, beautiful may be a tad too much, but it sure isn’t the eyesore that was want="\x🅓🅓\x🅓🅓\x🅓🅓\x🅓🅓\x🅓🅓\x🅓1".

We just leveled up. With such debuggable output, working on our compiler went from “fumbling in the dark” to “here, let me get that for you”.

#### BACK TO THE TASK AT HAND

Let’s take inventory. We have a lexer, a parser, the dotted outline of a compiler and a failing test that tells us that we need to generate two bytecode instructions. In our toolbox are the definition of an opcode and its operand, the Make function that lets us create bytecode instructions, our object system that we can use to pass Monkey values around, and readable and stunning Instructions.

And here’s a reminder of what our compiler needs to do: walk the AST recursively, find

\*ast.IntegerLiterals, evaluate them and turn them into \*object.Integers, add those to the

constants field, and add OpConstant instructions to its internal instructions slice.

Can you say “walk in the park”?

Let’s start by walking the AST. That’s something we already did in the Eval function we wrote in the previous book and there is no reason to change the approach. Here is how we get to our

\*ast.IntegerLiterals:

*// compiler/compiler.go*

**func** (c \*Compiler) Compile(node ast.Node) error {

**switch** node := node.(**type**) {

**case** \*ast.Program:

**for** \_, s := **range** node.Statements { err := c.Compile(s)

**if** err != nil {

**return** err

}

}

**case** \*ast.ExpressionStatement:

err := c.Compile(node.Expression)

**if** err != nil {

**return** err

}

**case** \*ast.InfixExpression:

err := c.Compile(node.Left)

**if** err != nil {

**return** err

}

err = c.Compile(node.Right)

**if** err != nil {

**return** err

}

**case** \*ast.IntegerLiteral:

*//* **TODO***: What now?!*

}

**return** nil

}

We first go through all the node.Statements in our \*ast.Program and call c.Compile with each of them. That gets us one level deeper in the AST, where we find an \*ast.ExpressionStatement. That’s what represents the 1 + 2 in our test. We then compile the node.Expression of that

\*ast.ExpressionStatement and end up with an \*ast.InfixExpression of which we have to com- pile the node.Left and the node.Right sides.

So far, so recursive. But the TODO is right, what do we do with the \*ast.IntegerLiterals?

We need to evaluate them. That’s safe to do, remember, because literals are constant expressions and their value does not change. A 2 will always evaluate to 2. And even though “evaluate” sounds sophisticated, it means creating an \*object.Integer:

*// compiler/compiler.go*

**func** (c \*Compiler) Compile(node ast.Node) error {

**switch** node := node.(**type**) {

*// [...]*

**case** \*ast.IntegerLiteral:

integer := &object.Integer{Value: node.Value}

*// [...]*

}

*// [...]*

}

Okay, so now we have the result of the evaluation – integer – at hand and can add it to our constant pool. To do that, we’ll add another helper to our compiler, called addConstant:

*// compiler/compiler.go*

**func** (c \*Compiler) addConstant(obj object.Object) int { c.constants = append(c.constants, obj)

**return** len(c.constants) - 1

}

We append the obj to the end of the compilers constants slice and give it its very own identifier by returning its index in the constants slice. This identifier will now be used as the operand for the OpConstant instruction that should cause the VM to load this constant from the constants

pool on to the stack.

We’re now able to add constants and to remember their identifier; time to emit a first instruction. Don’t be put off by the term: “emit” is compiler-speak for “generate” and “output”. It translates to: generate an instruction and add it to the results, either by printing it, writing it to a file or by adding it to a collection in memory. We’re going to do the last one:

*// compiler/compiler.go*

**func** (c \*Compiler) emit(op code.Opcode, operands ...int) int { ins := code.Make(op, operands...)

pos := c.addInstruction(ins)

**return** pos

}

**func** (c \*Compiler) addInstruction(ins []byte) int { posNewInstruction := len(c.instructions) c.instructions = append(c.instructions, ins...) **return** posNewInstruction

}

I’m sure that you understand all of it but I want you to make a mental note of the fact that emit returns the starting position of the just-emitted instruction. Add to this note that we’ll use the return value later on when we need to *go back* in c.instructions and modify it…

In the Compile method we can now use addConstant and emit to make one delicate change:

*// compiler/compiler.go*

**func** (c \*Compiler) Compile(node ast.Node) error {

**switch** node := node.(**type**) {

*// [...]*

**case** \*ast.IntegerLiteral:

integer := &object.Integer{Value: node.Value} c.emit(code.OpConstant, c.addConstant(integer))

*// [...]*

}

*// [...]*

}

One new line, emitting the OpConstant instruction. It’s enough:

$ go test ./compiler

ok monkey/compiler 🅓.🅓🅓8s

It’s strange that it says ok here, because we just turned the status of our first compiler test from

FAIL to woah-what-?-we-did-it-!-we-have-a-compiler-!.

#### POWERING ON THE MACHINE

Let’s do another inventory check. Here’s where we stand. We have one opcode defined, OpConstant. We have a tiny compiler that knows how to walk an AST and emit such an OpConstant instruction. Our tiny compiler also knows how to evaluate constant integer literal expressions and how to add them to its constant pool. And the compiler’s interface allows us to pass around the result of the compilation, including the emitted instructions and the constant pool.

While the instruction set of our bytecode may currently only be able to express “push this constant on to the stack” and not “do something with it”, it’s enough for us to get to work on our VM. Yes, really, it’s time for us to build our machine.

The goal for this section is to build up a VM that we can initialize with the Bytecode produced by the compiler, start and have it fetch, decode, and execute OpConstant instructions. The result of all that should be numbers being pushed on to the VM’s stack.

Sounds like a test? Well, it’s not hard to turn it into one. But before we can do that, we need to prepare by doing something unorthodox. We’ll now copy and paste our parse and testIntegerObject test helpers from our compiler tests to a new vm\_test.go file:

*// vm/vm\_test.go*

**package** vm

**import** (

"fmt" "monkey/ast" "monkey/lexer" "monkey/object" "monkey/parser"

)

**func** parse(input string) \*ast.Program { l := lexer.New(input)

p := parser.New(l)

**return** p.ParseProgram()

}

**func** testIntegerObject(expected int64, actual object.Object) error { result, ok := actual.(\*object.Integer)

**if** !ok {

**return** fmt.Errorf("object is not Integer. got=%T (%+v)", actual, actual)

}

**if** result.Value != expected {

**return** fmt.Errorf("object has wrong value. got=%d, want=%d", result.Value, expected)

}

**return** nil

}

Yes, *yes*, I hear you, duplication is bad, you’re right. But for now, the duplication is the most affordable solution while being easy to understand. It also won’t fall on our feet – trust me, I’ve read this book before.

Then we lay the groundwork for all future VM tests by copying the approach of the compiler tests and making it easy to define and run new test cases with the help of t.Helper:

*// vm/vm\_test.go*

**import** (

*// [...]*

"monkey/compiler"

*// [...]*

"testing"

)

**type** vmTestCase **struct** { input string expected **interface**{}

}

**func** runVmTests(t \*testing.T, tests []vmTestCase) { t.Helper()

**for** \_, tt := **range** tests { program := parse(tt.input)

comp := compiler.New()

err := comp.Compile(program)

**if** err != nil {

t.F6t6lf("compiler error: %s", err)

}

vm := New(comp.Bytecode()) err = vm.Run()

**if** err != nil {

t.F6t6lf("vm error: %s", err)

}

stackElem := vm.StackTop()

testExpectedObject(t, tt.expected, stackElem)

}

}

**func** testExpectedObject( t \*testing.T, expected **interface**{}, actual object.Object,

) {

t.Helper()

**switch** expected := expected.(**type**) {

**case** int:

err := testIntegerObject(int64(expected), actual)

**if** err != nil {

t.Effoff("testIntegerObject failed: %s", err)

}

}

}

The runVmTests function takes care of setting up and running each vmTestCase: lex and parse the input, take the AST, pass it to the compiler, check the compiler for errors and then hand the \*compiler.Bytecode over to the New function.

The New function should return a new instance of the virtual machine, assigned to vm. This is where we get to the heart of each test case. We start the vm with a call to vm.Run and after making sure it ran without an error, we use a method called StackTop to get the object that’s left sitting on top of the VM’s stack. We then pass that to testExpectedObject to make sure that this object matches what we expected in the vmTestCase.expected field.

Whew! Quite a lot of preparation and setup. But, trust me, this is going to make it so easy to

write VM tests in the future. I mean, take a look at this, our first test:

*// vm/vm\_test.go*

**func** TestIntegerArithmetic(t \*testing.T) { tests := []vmTestCase{

{"1", 1},

{"2", 2},

{"1 + 2", 2}, *//* **FIXME**

}

runVmTests(t, tests)

}

Isn’t that wonderful? No noise, no boilerplate. We just write down the Monkey code and what we expect to end up on the stack when the VM executes it.

Please note that we do *not* expect 3 but 2 to be sitting on top of the stack after compiling and executing 1 + 2. Sounds wrong, right? Well, it *is* wrong. At the end of this chapter executing 1

+ 2 should, of course, result in 3. But right now we only have OpConstant defined, which makes the only thing we can test and implement the pushing of constants on the stack. And in this

test case, the 2 is the second integer to be pushed, so that’s what we’re going to test.

The other two test cases, with only the integers 1 and 2 as their input, are sanity checks. They do not test separate functionality. Pushing a sole integer on to the stack is included in, well, pushing two of them. But these test cases do not have a huge cost and don’t take up a lot of space, so I added them to explicitly make sure that a single integer literal in an expression statement ends with an integer being pushed on to the stack.

Currently, though, nothing’s pushed because we haven’t defined VM yet. But we can do that now, since we already know which parts VM needs to have: instructions, constants and a stack.

*// vm/vm.go*

**package** vm

**import** (

"monkey/code" "monkey/compiler" "monkey/object"

)

**const** StackSize = 2🅓48

**type** VM **struct** {

constants []object.Object instructions code.Instructions

stack []object.Object

sp int *// Always points to the next value. Top of stack is stack[sp-1]*

}

**func** New(bytecode \*compiler.Bytecode) \*VM {

**return** &VM{

instructions: bytecode.Instructions, constants: bytecode.Constants,

stack: make([]object.Object, StackSize), sp: 🅓,

}

}

Our virtual machine is a struct with four fields. It holds the constants and instructions generated by the compiler and has a stack. Rather simple for something with such a grand name, isn’t it?

The stack is preallocated to have a StackSize number of elements, which should be enough for us. sp is the stackpointer, which we’ll increment or decrement to grow or shrink the stack, instead of modifying the stack slice itself.

Here’s the convention we’ll use for stack and sp: sp will always point to the *next* free slot in the stack. If there’s one element on the stack, located at index 🅓, the value of sp would be 1 and to access the element we’d use stack[sp-1]. A new element would be stored at stack[sp], before sp is incremented.

Knowing this, we define the StackTop method we use in our VM tests:

*// vm/vm.go*

**func** (vm \*VM) StackTop() object.Object {

**if** vm.sp == 🅓 {

**return** nil

}

**return** vm.stack[vm.sp-1]

}

Now the only thing that’s keeping us from running the tests is the missing Run method of the

VM:

$ go test ./vm # monkey/vm

vm/vm\_test.go:41:11: vm.Run undefined (type \*VM has no field or method Run) FAIL monkey/vm [build failed]

The Run method is what turns the VM into a virtual machine. It contains its heartbeat, the main loop, the fetch-decode-execute cycle:

*// vm/vm.go*

**func** (vm \*VM) Run() error {

**for** ip := 🅓; ip < len(vm.instructions); ip++ { op := code.Opcode(vm.instructions[ip])

**switch** op {

}

}

**return** nil

}

That’s the first part of the cycle, the “fetch”. We iterate through vm.instructions by incre- menting the instruction pointer, ip, and fetch the current instruction by directly accessing vm.instructions. Then we turn the byte into an Opcode. It’s important that we do not use code.Lookup here to get from a byte to an Opcode. That would be far too slow. It costs time to move the byte around, lookup the opcode’s definition, return it and take it apart.

I know that this doesn’t sound like my usual chant of “we’re here to learn, not to build the fastest thing ever”, but we’re in *the* hot path here; everything we can get rid of, we should throw out. Using code.Lookup here would be like putting a sleep statement in the loop. And

in contrast to a generic method that wants to lookup an opcode (like our mini-disassembler in Instructions.String) we have to encode our knowledge about the instructions into the VM’s Run method anyway. We can’t just delegate the execution away and treat every instruction the same.

Alas, fast as it may be, the “fetch” part alone is not enough:

$ go test ./vm

--- FAIL: TestIntegerArithmetic (🅓.🅓🅓s) vm\_test.go:2🅓: testIntegerObject failed:\

object is not Integer. got=<nil> (<nil>) vm\_test.go:2🅓: testIntegerObject failed:\

object is not Integer. got=<nil> (<nil>) vm\_test.go:2🅓: testIntegerObject failed:\

object is not Integer. got=<nil> (<nil>) FAIL

FAIL monkey/vm 🅓.🅓🅓6s

Time to “decode” and “execute”. Decoding means adding a new case branch and decoding the operands of the instruction:

*// vm/vm.go*

**func** (vm \*VM) Run() error {

*// [...]*

**switch** op {

**case** code.OpConstant:

constIndex := code.ReadUint16(vm.instructions[ip+1:]) ip += 2

}

*// [...]*

}

With the code.ReadUint16 function we decode the operands in the bytecode, starting with the byte positioned right after the opcode, at ip+1. And we use code.ReadUint16 and not code.ReadOperands for the same reason we don’t use code.Lookup when fetching the instruction: speed.

After decoding the operands, we must be careful to increment ip by the correct amount – the number of bytes we read to decode the operands. The result is that the next iteration of the loop starts with ip pointing to an opcode instead of an operand.

We still can’t run the tests, because the compiler now tells us to use the declared but unused

constIndex. We better do that, by adding the “execute” part of our VM cycle:

*// vm/vm.go*

**import** (

"fmt"

*// [...]*

)

**func** (vm \*VM) Run() error {

*// [...]*

**switch** op {

**case** code.OpConstant:

constIndex := code.ReadUint16(vm.instructions[ip+1:]) ip += 2

err := vm.push(vm.constants[constIndex])

**if** err != nil {

**return** err

}

}

*// [...]*

}

**func** (vm \*VM) push(o object.Object) error {

**if** vm.sp >= StackSize {

**return** fmt.Errorf("stack overflow")

}

vm.stack[vm.sp] = o vm.sp++

**return** nil

}

We use constIndex to get to the constant in vm.constants and then push it on to the stack. The succinctly named vm.push method that takes care of checking the stack size, adding the object and incrementing the stackpointer, sp.

Our virtual machine has come to life:

$ go test ./vm

ok monkey/vm 🅓.🅓🅓7s

That means we have successfully defined our own bytecode format, built a compiler that trans- lates a subset of Monkey into this bytecode format and created a virtual machine that executes the bytecode. Again, ok is rather somber – itstimetodance would be more appropriate.

We’ve also built a lot of infrastructure and tools to compile and execute these two OpConstant instruction. And at the moment that might feel excessive, but believe me, it’s going to pay off. We can already see the benefits when we add another opcode now.

## ADDING ON THE STACK

At the beginning of this chapter we set out to compile and execute the Monkey expression 1 +

1. Now we’re nearly there. All that’s left to do is to actually add the integers we’ve pushed on to the stack. For that, we need a new opcode.

The new opcode is called OpAdd and tells the VM to pop the two topmost elements off the stack, add them together and push the result back on to the stack. In contrast to OpConstant, it doesn’t have any operands. It’s simply one byte, a single opcode:

*// code/code.go*

**const** (

OpConstant Opcode = iota OpAdd

)

**var** definitions = **map**[Opcode]\*Definition{ OpConstant: {"OpConstant", []int{2}},

OpAdd: {"OpAdd", []int{}},

}

Right next to OpConstant we add the new definition of OpAdd. There’s nothing remarkable here, except that the OperandWidths field in the \*Definition holds an empty slice to signify that OpAdd doesn’t have any operands. And that’s only remarkable in how unremarkable it is. But we still need to make sure that our tooling can handle an opcode without any operands. First on the list is Make:

*// code/code\_test.go*

**func** TestMake(t \*testing.T) { tests := []**struct** {

op Opcode operands []int expected []byte

}{

*// [...]*

{OpAdd, []int{}, []byte{byte(OpAdd)}},

}

*// [...]*

}

One new test case to make sure that Make knows how to encode a single Opcode into a byte slice. And guess what? It already does:

$ go test ./code

ok monkey/code 🅓.🅓🅓6s

That means we can now use Make to test whether the Instructions.String method can also handle OpAdd. We change the test input and the expectation to include it:

*// code/code\_test.go*

**func** TestInstructionsString(t \*testing.T) { instructions := []Instructions{

Make(OpAdd), Make(OpConstant, 2),

Make(OpConstant, 65535),

}

expected := `🅓🅓🅓🅓 OpAdd

🅓🅓🅓1 OpConstant 2

🅓🅓🅓4 OpConstant 65535

`

*// [...]*

}

But, we didn’t get so lucky this time, the test fails:

$ go test ./code

--- FAIL: TestInstructionsString (🅓.🅓🅓s) code\_test.go:51: instructions wrongly formatted.

want="🅓🅓🅓🅓 OpAdd\n🅓🅓🅓1 OpConstant 2\n🅓🅓🅓4 OpConstant 65535\n" got="🅓🅓🅓🅓 ERROR: unhandled operandCount for OpAdd\n\n\

🅓🅓🅓1 OpConstant 2\n🅓🅓🅓4 OpConstant 65535\n"

FAIL

FAIL monkey/code 🅓.🅓🅓7s

The error message points in the right direction, though. We need to extend the switch statement in the Instructions.fmtInstruction method to handle opcodes with no operands:

*// code/code.go*

**func** (ins Instructions) fmtInstruction(def \*Definition, operands []int) string {

*// [...]*

**switch** operandCount {

**case** 🅓:

**return** def.Name

**case** 1:

**return** fmt.Sprintf("%s %d", def.Name, operands[🅓])

}

**return** fmt.Sprintf("ERROR: unhandled operandCount for %s\n", def.Name)

}

That turns our tests from red to green again:

$ go test ./code

ok monkey/code 🅓.🅓🅓6s

And since OpAdd doesn’t have any operands, we don’t need to change ReadOperands, which means we’re done with updating our tools. OpAdd is now fully defined and ready to be used in our compiler.

Now, think back to our first compiler test, TestIntegerArithmetic. We asserted that the Monkey expression 1 + 2 should result in two OpConstant instructions. That was wrong back then and it’s still wrong. But we set out to build the smallest possible compiler that only does one thing, pushing integers on to the stack, so that was okay. Now we want to add those numbers, which means it’s time to fix the test by adding the missing OpAdd instruction:

*// compiler/compiler\_test.go*

**func** TestIntegerArithmetic(t \*testing.T) { tests := []compilerTestCase{

{

input: "1 + 2",

expectedConstants: []**interface**{}{1, 2}, expectedInstructions: []code.Instructions{

code.Make(code.OpConstant, 🅓), code.Make(code.OpConstant, 1), code.Make(code.OpAdd),

},

},

}

*// [...]*

}

expectedInstructions is now correct. Two OpConstant instructions to push the two constants on to the stack and then an OpAdd instruction that should cause the VM to add them together.

Since we only updated our tools but not yet the compiler, the test now tells us which instruction we’re not emitting:

$ go test ./compiler

--- FAIL: TestIntegerArithmetic (🅓.🅓🅓s)

compiler\_test.go:26: testInstructions failed: wrong instructions length. want="🅓🅓🅓🅓 OpConstant 🅓\n🅓🅓🅓3 OpConstant 1\n🅓🅓🅓6 OpAdd\n"

got ="🅓🅓🅓🅓 OpConstant 🅓\n🅓🅓🅓3 OpConstant 1\n" FAIL

FAIL monkey/compiler 🅓.🅓🅓7s

It’s my unwavering opinion that the Instructions.String method is worth its bytes in gold. This failing test now tells us – nicely formatted and readable! – that we need to emit an

OpAdd instruction. And since we already came across an \*ast.InfixExpression in the compiler’s

Compile method, we know where to do that:

*// compiler/compiler.go*

**import** (

"fmt"

*// [...]*

)

**func** (c \*Compiler) Compile(node ast.Node) error {

**switch** node := node.(**type**) {

*// [...]*

**case** \*ast.InfixExpression:

err := c.Compile(node.Left)

**if** err != nil {

**return** err

}

err = c.Compile(node.Right)

**if** err != nil {

**return** err

}

**switch** node.Operator {

**case** "+":

c.emit(code.OpAdd)

**default**:

**return** fmt.Errorf("unknown operator %s", node.Operator)

}

*// [...]*

}

*// [...]*

}

In the new switch statement we check the Operator field of the \*ast.InfixExpression node. In case we have a + at hand (which we do in our test) we c.emit an OpAdd instruction. Then, to be safe, we add a default branch that returns an error in case we don’t know how to compile a specific infix operator. And yes, you guessed it, we’re going to add more case branches later on.

And that’s it. Our compiler is now able to emit OpAdd instructions:

$ go test ./compiler

ok monkey/compiler 🅓.🅓🅓6s

This starts to feel like a winning streak. And now, instead of going on any detour, we move directly to the VM and implement OpAdd there too.

The best part about that is that we don’t have to write a new test – we just have to fix our old one. Because in the vm package we also wrote a “wrong” test before. Remember our assertion about 1 + 2 leaving 2 on the stack? We need to change that:

*// vm/vm\_test.go*

**func** TestIntegerArithmetic(t \*testing.T) { tests := []vmTestCase{

*// [...]*

{"1 + 2", 3},

}

runVmTests(t, tests)

}

Now we expect a 3 instead of a 2. But that alone makes the test fail:

$ go test ./vm

--- FAIL: TestIntegerArithmetic (🅓.🅓🅓s) vm\_test.go:2🅓: testIntegerObject failed:\

object has wrong value. got=2, want=3 FAIL

FAIL monkey/vm 🅓.🅓🅓7s

We now have to actually do something with the integers we pushed on to the stack, which means we’ve finally arrived at stack arithmetic. And what’s the first thing we need to do to add two numbers together? Exactly, pop the operands off the stack. To help with that, we add another helper to VM:

*// vm/vm.go*

**func** (vm \*VM) pop() object.Object { o := vm.stack[vm.sp-1]

vm.sp--

**return** o

}

We first take the element from the top of the stack, located at vm.sp-1, and put it on the side. Then we decrement vm.sp, allowing the location of element that was just popped off being overwritten eventually.

In order to use this new pop method we first need to add the “decode” part for the new OpAdd instruction. But since that’s not really worth mentioning on its own, here it is with the first part of the “execute”:

*// vm/vm.go*

**func** (vm \*VM) Run() error {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpAdd: right := vm.pop() left := vm.pop()

leftValue := left.(\*object.Integer).Value rightValue := right.(\*object.Integer).Value

}

*// [...]*

}

Extending the “decode” part of the run-loop means adding the new case code.OpAdd. After that, we’re ready to implement the operation itself, the “execute”. In this case we start by popping the operands off the stack and unwrapping their values into leftValue and rightValue.

It looks innocent enough, but here is where subtle bugs can creep in. We implicitly assume that the right operand of the infix operator is the last one to be pushed on to the stack. In the case of + the order of the operands doesn’t matter, so the implicitness is not an immediate problem. But there are other operators where the wrong order of the operands can cause wrong results. And I’m not talking about some exotic operator here – minus also needs its operand to be in the correct order.

That was just the start of the implementation of OpAdd and the VM test is still failing, so let’s finish it with one elegant addition:

*// vm/vm.go*

**func** (vm \*VM) Run() error {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpAdd: right := vm.pop() left := vm.pop()

leftValue := left.(\*object.Integer).Value rightValue := right.(\*object.Integer).Value

result := leftValue + rightValue vm.push(&object.Integer{Value: result})

*// [...]*

}

*// [...]*

}

Here’s what the two added lines are doing: add leftValue and rightValue together, turn the result into an \*object.Integer and push that on to the stack. And here’s what that amounts to:

$ go test ./vm

ok monkey/vm 🅓.🅓🅓6s

Passing tests! We did it: we achieved the goal of this chapter and successfully compiled and executed the Monkey expression 1 + 2.

We can lean back now, take a big breath, relax and ponder how it feels to write a compiler and a virtual machine. I bet it wasn’t as hard as you thought it would be. Granted, our compiler and the VM are not what you’d call “feature rich”. But we’re not done yet – far from that – and we’ve built important infrastructure that’s essential to both the compiler and the VM. We can be proud of ourselves.

## HOOKING UP THE REPL

Before we move on, we can hook up the compiler and the VM to our REPL. That allows us to get instant feedback when we want to experiment with Monkey. All that takes is to remove the evaluator and the environment setup from our REPL’s Start function and replace it with the calls to the compiler and the VM we already know from our tests:

*// repl/repl.go*

**import** (

"bufio" "fmt"

"io" "monkey/compiler" "monkey/lexer" "monkey/parser" "monkey/vm"

)

**func** Start(in io.Reader, out io.Writer) { scanner := bufio.NewScanner(in)

**for** {

fmt.Fprintf(out, PROMPT) scanned := scanner.Scan() **if** !scanned {

**return**

}

line := scanner.Text() l := lexer.New(line)

p := parser.New(l)

program := p.ParseProgram()

**if** len(p.Errors()) != 🅓 { printParserErrors(out, p.Errors())

**continue**

}

comp := compiler.New()

err := comp.Compile(program)

**if** err != nil {

fmt.Fprintf(out, "Woops! Compilation failed:\n %s\n", err)

**continue**

}

machine := vm.New(comp.Bytecode()) err = machine.Run()

**if** err != nil {

fmt.Fprintf(out, "Woops! Executing bytecode failed:\n %s\n", err)

**continue**

}

stackTop := machine.StackTop() io.WriteString(out, stackTop.Inspect()) io.WriteString(out, "\n")

}

}

First we tokenize the input, then we parse it, then we compile and execute the program. We also replace the previous printing of Eval’s return value with printing the object that sits on top of the VM’s stack.

Now we can start up the REPL and see our compiler and VM work behind the scenes:

$ go build -o monkey . && ./monkey

Hello mrnugget! This is the Monkey programming language! Feel free to type in commands

>> 1

1

>> 1 + 2

3

>> 1 + 2 + 3

6

>> 1🅓🅓🅓 + 555 1555

Beautiful. But, of course, it all comes crashing down as soon as we want to do more than add two numbers together:

>> 99 - 1

Woops! Compilation failed: unknown operator -

>> 8🅓 / 2

Woops! Compilation failed: unknown operator /

We still have work to do. Let’s get to it.

CHAPTER 3

COMPILING EXPRESSIONS

In the previous two chapters we learned a lot of new and strange things, built a tiny compiler, a virtual machine and defined bytecode instructions. In this one, we’ll use our newfound knowl- edge of bytecode compilation and execution, and combine it with what we know from the first book to add support for all the infix and prefix operators Monkey supports.

That not only allows us to familiarize ourselves with our codebase and further extend its infras- tructure but to also catch our breath. Besides, we have some cleaning up to do before we can move on. The thing most in need of a scrub? Our stack.

## CLEANING UP THE STACK

In their current state, the only thing our compiler and our VM are capable of is adding two numbers. Give them the expression 1 + 2 and our VM will correctly put 3 on its stack. That’s exactly what we want, but there’s one issue lurking in the shadows: the 3 stays on the stack and will stay there for all eternity if we don’t do something about it.

The problem is not with the expression 1 + 2 itself, but rather where it occurs. It’s part of an *expression statement*. As a quick refresher, we have three types of statements in Monkey: let statements, return statements and expression statements. Whereas the first two explicitly reuse the value their child-expression nodes produce, expression statements merely wrap expressions so they can occur on their own. The value they produce is not reused, by definition. But now our problem is that we *do* reuse it, because we involuntarily keep it on the stack.

Take a look at the following Monkey program:

1;

2;

3;

That’s three separate expression statements. And you know what ends up on the stack? Not just the value produced last, 3, but everything: 1, 2 and 3. If we have a Monkey program consisting of lots of expression statements we could fill up the stack by accident. That’s not good.

In order to fix that, we need to do two things. First, we need to define a new opcode that tells the VM to pop the topmost element off the stack. Second, we need to emit this opcode after every expression statement.

We start with the definition of the opcode, the aptly name OpPop:

*// code/code.go*

**const** (

*// [...]*

OpPop

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpPop: {"OpPop", []int{}},

}

OpPop doesn’t have any operands, just like OpAdd. Its only job is to tell the VM to pop the topmost element off the stack and for that it doesn’t need an operand.

Now we need to use this new opcode to clean the stack after every expression statement. Thank- fully, it’s easy to assert this new behaviour in our test suite, since we don’t have a lot of compiler tests yet, which is also why I thought it wise to introduce OpPop now instead of two chapters down the road. We only need to change our single test case in TestIntegerArithmetic:

*// compiler/compiler\_test.go*

**func** TestIntegerArithmetic(t \*testing.T) { tests := []compilerTestCase{

{

input: "1 + 2",

expectedConstants: []**interface**{}{1, 2}, expectedInstructions: []code.Instructions{

code.Make(code.OpConstant, 🅓), code.Make(code.OpConstant, 1), code.Make(code.OpAdd), code.Make(code.OpPop),

},

},

}

runCompilerTests(t, tests)

}

The only change here is the new line containing the code.Make(code.OpPop) call. We assert that the compiled expression statement should be followed by an OpPop instruction. The desired behaviour can be made even clearer by adding another test with multiple expression statements:

*// compiler/compiler\_test.go*

**func** TestIntegerArithmetic(t \*testing.T) { tests := []compilerTestCase{

*// [...]*

{

input: "1; 2",

expectedConstants: []**interface**{}{1, 2}, expectedInstructions: []code.Instructions{

code.Make(code.OpConstant, 🅓), code.Make(code.OpPop), code.Make(code.OpConstant, 1), code.Make(code.OpPop),

},

},

}

runCompilerTests(t, tests)

}

Please note the ; that separates the 1 from the 2. Both integer literals are separate expression statements and after each statement an OpPop instruction should be emitted. That’s not what currently happens. Instead we tell our VM to fill up its stack by loading constants on to it:

$ go test ./compiler

--- FAIL: TestIntegerArithmetic (🅓.🅓🅓s)

compiler\_test.go:37: testInstructions failed: wrong instructions length. want="🅓🅓🅓🅓 OpConstant 🅓\n🅓🅓🅓3 OpConstant 1\n🅓🅓🅓6 OpAdd\n🅓🅓🅓7 OpPop\n" got ="🅓🅓🅓🅓 OpConstant 🅓\n🅓🅓🅓3 OpConstant 1\n🅓🅓🅓6 OpAdd\n"

FAIL

FAIL monkey/compiler 🅓.🅓🅓7s

But here comes the good part. In order to fix this and properly clean the stack, all we need to do is to add a single call to c.emit to our compiler:

*// compiler/compiler.go*

**func** (c \*Compiler) Compile(node ast.Node) error {

**switch** node := node.(**type**) {

*// [...]*

**case** \*ast.ExpressionStatement:

err := c.Compile(node.Expression)

**if** err != nil {

**return** err

}

c.emit(code.OpPop)

*// [...]*

}

*// [...]*

}

After compiling the node.Expression of an \*ast.ExpressionStatement we emit an OpPop. That’s all it takes:

$ go test ./compiler

ok monkey/compiler 🅓.🅓🅓6s

Okay, that’s not *all* it takes. We still have some work left to do, because now we need to tell our VM how to handle this OpPop instruction, which would also be a tiny addition if it weren’t for our tests.

In our VM tests we used vm.StackTop to make sure that our VM put the correct things on to its stack, but with OpPop in play we can’t do that anymore. Now, what want to assert is that “this should have been on the stack, right before you popped it off, dear VM”. In order to do that, we add a test-only method to our VM and get rid of StackTop:

*// vm/vm.go*

**func** (vm \*VM) LastPoppedStackElem() object.Object {

**return** vm.stack[vm.sp]

}

As per our convention vm.sp always points to the *next free* slot in vm.stack. This is where a new element would be pushed. But since we only pop elements off the stack by decrementing vm.sp (without explicitly setting them to nil), this is also where we can find the elements that were previously on top of the stack. With LastPoppedStackElem, we can change our VM tests to make sure OpPop is actually handled correctly:

*// vm/vm\_test.go*

**func** runVmTests(t \*testing.T, tests []vmTestCase) { t.Helper()

**for** \_, tt := **range** tests {

*// [...]*

stackElem := vm.LastPoppedStackElem()

testExpectedObject(t, tt.expected, stackElem)

}

}

We changed the call from vm.StackTop to vm.LastPoppedStackElem. It’s enough to break our tests:

$ go test ./vm

--- FAIL: TestIntegerArithmetic (🅓.🅓🅓s) vm\_test.go:2🅓: testIntegerObject failed:\

object is not Integer. got=<nil> (<nil>) vm\_test.go:2🅓: testIntegerObject failed:\

object is not Integer. got=<nil> (<nil>) vm\_test.go:2🅓: testIntegerObject failed:\

object has wrong value. got=2, want=3 FAIL

FAIL monkey/vm 🅓.🅓🅓7s

In order to get them to pass again, we need to tell our VM to keep its stack clean and tidy:

*// vm/vm.go*

**func** (vm \*VM) Run() error {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpPop: vm.pop()

}

*// [...]*

}

With that, stack hygiene is restored:

$ go test ./vm

ok monkey/vm 🅓.🅓🅓6s

But we also need to fix our REPL, where we still use StackTop, by replacing it with

LastPoppedStackElem:

*// repl/repl.go*

**func** Start(in io.Reader, out io.Writer) {

*// [...]*

**for** {

*// [...]*

lastPopped := machine.LastPoppedStackElem() io.WriteString(out, lastPopped.Inspect()) io.WriteString(out, "\n")

}

}

Perfect! That means we can move on and safely do more arithmetic on the stack without the stack slowly blowing up in our face.

## INFIX EXPRESSIONS

Monkey supports eight infix operators and four of them are being used for arithmetic: +, -, \* and /. We’ve already added support for + with the OpAdd opcode. Now we need to add three more. And since all three of them work the same way in regards to their use of operands and the stack, we’ll add them together.

The first step is to add the Opcode definitions to the code package:

*// code/code.go*

**const** (

*// [...]*

OpSub OpMul OpDiv

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpSub: {"OpSub", []int{}},

OpMul: {"OpMul", []int{}},

OpDiv: {"OpDiv", []int{}},

}

OpSub stands for the -, OpMul for the \* and OpDiv for the / infix operator. With these opcodes defined, we can use them in our compiler tests to make sure the compiler knows how to output them:

*// compiler/compiler\_test.go*

**func** TestIntegerArithmetic(t \*testing.T) { tests := []compilerTestCase{

*// [...]*

{

input: "1 - 2",

expectedConstants: []**interface**{}{1, 2}, expectedInstructions: []code.Instructions{

code.Make(code.OpConstant, 🅓), code.Make(code.OpConstant, 1), code.Make(code.OpSub), code.Make(code.OpPop),

},

},

{

input: "1 \* 2",

expectedConstants: []**interface**{}{1, 2}, expectedInstructions: []code.Instructions{

code.Make(code.OpConstant, 🅓), code.Make(code.OpConstant, 1), code.Make(code.OpMul), code.Make(code.OpPop),

},

},

{

input: "2 / 1",

expectedConstants: []**interface**{}{2, 1}, expectedInstructions: []code.Instructions{

code.Make(code.OpConstant, 🅓), code.Make(code.OpConstant, 1), code.Make(code.OpDiv), code.Make(code.OpPop),

},

},

}

runCompilerTests(t, tests)

}

Hopefully the only thing that gives you pause here is the last test case, where I changed the order of the operands. Other than that, these are boringly similar to our previous test case for 1 + 2, except for the operator itself and the expected opcode. But, alas, similarity is not something a compiler understands natively:

$ go test ./compiler

--- FAIL: TestIntegerArithmetic (🅓.🅓🅓s) compiler\_test.go:67: compiler error: unknown operator - FAIL

FAIL monkey/compiler 🅓.🅓🅓6s

What we need to change is the switch statement in our compiler’s Compile method that checks

node.Operator:

*// compiler/compiler.go*

**func** (c \*Compiler) Compile(node ast.Node) error {

**switch** node := node.(**type**) {

*// [...]*

**case** \*ast.InfixExpression:

*// [...]*

**switch** node.Operator {

**case** "+":

c.emit(code.OpAdd)

**case** "-":

c.emit(code.OpSub)

**case** "\*":

c.emit(code.OpMul)

**case** "/":

c.emit(code.OpDiv)

**default**:

**return** fmt.Errorf("unknown operator %s", node.Operator)

}

*// [...]*

}

*// [...]*

}

Only six lines in this snippet are new: the case branches for "-", "\*" and "/". And they make the tests pass:

$ go test ./compiler

ok monkey/compiler 🅓.🅓🅓6s

Alright, the compiler now outputs three more opcodes. Our VM must now step up to this challenge. And here too, the first thing to do is add test cases:

*// vm/vm\_test.go*

**func** TestIntegerArithmetic(t \*testing.T) { tests := []vmTestCase{

*// [...]*

{"1 - 2", -1},

{"1 \* 2", 2},

{"4 / 2", 2},

{"5🅓 / 2 \* 2 + 1🅓 - 5", 55},

{"5 + 5 + 5 + 5 - 1🅓", 1🅓},

{"2 \* 2 \* 2 \* 2 \* 2", 32},

{"5 \* 2 + 1🅓", 2🅓},

{"5 + 2 \* 1🅓", 25},

{"5 \* (2 + 1🅓)", 6🅓},

}

runVmTests(t, tests)

}

Some might say that this is excessive, going overboard. But what I want to show you here is the power of stack arithmetic. We not only have the three test cases necessary to make sure the OpSub, OpMul and OpDiv opcodes are recognized by the VM, but there’s also a series of test cases that mix the infix operators, playing with their varying levels of precedence and manipulating them by hand with added parentheses. For now, they all fail, which is not the point I wanted to make:

$ go test ./vm

--- FAIL: TestIntegerArithmetic (🅓.🅓🅓s)

vm\_test.go:3🅓: testIntegerObject failed: object has wrong value.\ got=2, want=-1

vm\_test.go:3🅓: testIntegerObject failed: object has wrong value.\ got=5, want=55

vm\_test.go:3🅓: testIntegerObject failed: object has wrong value.\ got=2, want=32

vm\_test.go:3🅓: testIntegerObject failed: object has wrong value.\ got=12, want=2🅓

vm\_test.go:3🅓: testIntegerObject failed: object has wrong value.\ got=12, want=25

vm\_test.go:3🅓: testIntegerObject failed: object has wrong value.\ got=12, want=6🅓

FAIL

FAIL monkey/vm 🅓.🅓🅓7s

My point is how minimal the required changes are to make all of them pass. First, we replace the existing case code.OpAdd in our VM with this:

*// vm/vm.go*

**func** (vm \*VM) Run() error {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpAdd, code.OpSub, code.OpMul, code.OpDiv: err := vm.executeBinaryOperation(op)

**if** err != nil {

**return** err

}

}

*// [...]*

}

Everything that has to with binary operations is now neatly tucked away behind the

executeBinaryOperation method:

*// vm/vm.go*

**func** (vm \*VM) executeBinaryOperation(op code.Opcode) error { right := vm.pop()

left := vm.pop()

leftType := left.Type() rightType := right.Type()

**if** leftType == object.INTEGER\_OBJ && rightType == object.INTEGER\_OBJ {

**return** vm.executeBinaryIntegerOperation(op, left, right)

}

**return** fmt.Errorf("unsupported types for binary operation: %s %s", leftType, rightType)

}

It doesn’t do much more than type assertions and possibly producing an error and delegates most of the work to executeBinaryIntegerOperation:

*// vm/vm.go*

**func** (vm \*VM) executeBinaryIntegerOperation( op code.Opcode,

left, right object.Object,

) error {

leftValue := left.(\*object.Integer).Value rightValue := right.(\*object.Integer).Value

**var** result int64

**switch** op {

**case** code.OpAdd:

result = leftValue + rightValue

**case** code.OpSub:

result = leftValue - rightValue

**case** code.OpMul:

result = leftValue \* rightValue

**case** code.OpDiv:

result = leftValue / rightValue

**default**:

**return** fmt.Errorf("unknown integer operator: %d", op)

}

**return** vm.push(&object.Integer{Value: result})

}

Here is where we finally unwrap the integers contained in the left and right operands and produce a result according to the op. There shouldn’t be any surprises here, because this method has a really similar counterpart in the evaluator package we built in the first book.

Here’s the result of adding these two methods:

$ go test ./vm

ok monkey/vm 🅓.🅓1🅓s

Addition, subtraction, multiplication, division – they all work. As a single operations, combined, grouped by parentheses; all we do is pop operands off the stack and push the result back. Stack arithmetic – pretty neat.

## BOOLEANS

These four operators we added comprise only a subset of the operators available in Monkey. There are also the comparison operators ==, !=, >, < and the two prefix operators ! and -. Our next goal is to implement all of them and that includes teaching our Monkey VM about another Monkey data type: booleans. Without booleans we couldn’t represent the results of these operators (well, except for the - prefix), but booleans also exist as literal expressions in Monkey:

**true**; **false**;

We’re going to start by adding support for these two literals. That way we already have our boolean data type in place when we add the operators.

So what does a boolean literal do? Well, in our evaluator a boolean literal evaluates to the boolean value it designates: true or false. Now we’re working with a compiler and a VM, so we have to adjust our expectations a little bit. Instead of boolean literals *evaluating* to boolean values, we now want them to cause the VM to load the boolean values on to the stack.

That’s pretty close to what integer literals do and those are compiled to OpConstant instructions. We *could* treat true and false as constants too, but that would be a waste, not only of bytecode but also of compiler and VM resources. Instead, we’ll now define two new opcodes that directly tell the VM to push an \*object.Boolean on to the stack:

*// code/code.go*

**const** (

*// [...]*

OpTrue OpFalse

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpTrue: {"OpTrue", []int{}},

OpFalse: {"OpFalse", []int{}},

}

Granted, I didn’t waste a lot of creativity on the naming of these two. But what they do should be clear nonetheless. Both opcodes have no operands and simply tell the VM “push true or false on to the stack”.

We can now use that to create a compiler test in which we make sure that the boolean literals

true and false are translated to OpTrue and OpFalse instructions:

*// compiler/compiler\_test.go*

**func** TestBooleanExpressions(t \*testing.T) { tests := []compilerTestCase{

{

input: "true",

expectedConstants: []**interface**{}{}, expectedInstructions: []code.Instructions{

code.Make(code.OpTrue), code.Make(code.OpPop),

},

},

{

input: "false", expectedConstants: []**interface**{}{}, expectedInstructions: []code.Instructions{

code.Make(code.OpFalse), code.Make(code.OpPop),

},

},

}

runCompilerTests(t, tests)

}

This is our second compiler test and has the same structure as the first one. The tests slice will be extended once we implement the comparison operators.

Both test cases fail, because the compiler only knows that it should emit an OpPop after expres- sion statements:

$ go test ./compiler

--- FAIL: TestBooleanExpressions (🅓.🅓🅓s)

compiler\_test.go:9🅓: testInstructions failed: wrong instructions length. want="🅓🅓🅓🅓 OpTrue\n🅓🅓🅓1 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n" FAIL

FAIL monkey/compiler 🅓.🅓🅓9s

In order to emit OpTrue or OpFalse instructions we need to add a new case branch for

\*ast.Boolean to the compiler’s Compile method:

*// compiler/compiler.go*

**func** (c \*Compiler) Compile(node ast.Node) error {

**switch** node := node.(**type**) {

*// [...]*

**case** \*ast.Boolean:

**if** node.Value { c.emit(code.OpTrue)

} **else** {

c.emit(code.OpFalse)

}

*// [...]*

}

*// [...]*

}

Yep, that did it, alright:

$ go test ./compiler

ok monkey/compiler 🅓.🅓🅓8s

The next step is to tell the VM about true and false. And just like in the compiler package we now create a second test function:

*// vm/vm\_test.go*

**func** TestBooleanExpressions(t \*testing.T) { tests := []vmTestCase{

{"true", true},

{"false", false},

}

runVmTests(t, tests)

}

This test function is very similar to the first one, TestIntegerArithmetic. But since we now have a bool as our expectation here, we need to update the testExpectedObject function used by runVmTests and provide it with a new helper function called testBooleanObject:

*// vm/vm\_test.go*

**func** testExpectedObject( t \*testing.T, expected **interface**{}, actual object.Object,

) {

t.Helper()

**switch** expected := expected.(**type**) {

*// [...]*

**case** bool:

err := testBooleanObject(bool(expected), actual)

**if** err != nil {

t.Effoff("testBooleanObject failed: %s", err)

}

}

}

**func** testBoole6nObject(expected bool, 6ctu6l object.Object) effof { fesult, ok := 6ctu6l.(\*object.Boole6n)

**if** !ok {

**return** fmt.Effoff("object is not Boole6n. got=%T (%+v)", 6ctu6l, 6ctu6l)

)

**if** fesult.V6lue != expected {

**return** fmt.Effoff("object h6s wfong v6lue. got=%t, w6nt=%t", fesult.V6lue, expected)

)

**return** nil

)

testBoole6nObject is testIntegefObject’s counterpart and not the most fascinating of functions, but we’ll get a lot of use out of it, that’s for sure. What *is* noteworthy is that our VM tests now blow up:

$ go test ./vm

--- FAIL: TestBoole6nExpfessions (🅓.🅓🅓s)

p6nic: funtime effof: index out of f6nge [fecovefed] p6nic: funtime effof: index out of f6nge

gofoutine 19 [funning]: testing.tRunnef.func1(🅓xc42🅓🅓b61e🅓)

/usf/loc6l/go/sfc/testing/testing.go:742 +🅓x29d p6nic(🅓x1116f2🅓, 🅓x11eefc🅓)

/usf/loc6l/go/sfc/funtime/p6nic.go:5🅓2 +🅓x229 monkey/vm.(\*VM).pop(...)

/Usefs/mfnugget/code/🅓2/sfc/monkey/vm/vm.go:74 monkey/vm.(\*VM).Run(🅓xc42🅓🅓5🅓ed8, 🅓x8🅓🅓, 🅓x8🅓🅓)

/Usefs/mfnugget/code/🅓2/sfc/monkey/vm/vm.go:49 +🅓x16f monkey/vm.funVmTests(🅓xc42🅓🅓b61e🅓, 🅓xc42🅓🅓79f58, 🅓x2, 🅓x2)

/Usefs/mfnugget/code/🅓2/sfc/monkey/vm/vm\_test.go:6🅓 +🅓x356 monkey/vm.TestBoole6nExpfessions(🅓xc42🅓🅓b61e🅓)

/Usefs/mfnugget/code/🅓2/sfc/monkey/vm/vm\_test.go:39 +🅓x6🅓 testing.tRunnef(🅓xc42🅓🅓b61e🅓, 🅓x11476d🅓)

/usf/loc6l/go/sfc/testing/testing.go:777 +🅓xd🅓 cfe6ted by testing.(\*T).Run

/usf/loc6l/go/sfc/testing/testing.go:824 +🅓x2e FAIL monkey/vm 🅓.🅓11s

It blows up because we’re so tidy and issue an OpPop after every expression statement to keep the stack clean. And when we try to pop something off the stack without first putting something on it, we get an index out of f6nge panic.

The first step towards fixing this is to tell our VM about tfue and f6lse and defining global

Tfue and F6lse instances of them:

*// vm/vm.go*

**var** Tfue = &object.Boole6n{V6lue: tfue)

**var** F6lse = &object.Boole6n{V6lue: f6lse)

The reasons for reusing global instances of the \*object.Boole6ns are the same as in our ev6lu6tof package. First, these are immutable, unique values. tfue will always be tfue and f6lse will always be f6lse. Treating them as such by defining them as global variables is an absolute no-brainer in terms of performance. Why create new \*object.Boole6ns with the same value if

we can just reference these two? The second reason is that this makes comparisons in Monkey, like tfue == tfue, easier to implement and perform, because we can just compare two pointers without having to unwrap the value they’re pointing at.

Of course, defining Tfue and F6lse does not magically make the tests pass. We also need to push them on to the stack when instructed to do so. For that we extend the main loop of our VM:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpTfue:

eff := vm.push(Tfue)

**if** eff != nil {

**return** eff

)

**case** code.OpF6lse:

eff := vm.push(F6lse)

**if** eff != nil {

**return** eff

)

)

*// [...]*

)

Not much to explain here: we push the globals Tfue and F6lse on to the stack. That means that we *are* actually pushing something on to the stack before trying to clean it up again, which means our tests don’t blow up anymore:

$ go test ./vm

ok monkey/vm 🅓.🅓🅓7s

Alright! Boolean literals work and our VM now knows about Tfue and F6lse. Now we can start to implement the comparison operators, because now we can put their results on to the stack.

## COMPARISON OPERATORS

The four comparison operators in Monkey are: ==, !=, > and <. We will now add support for all four of them by adding three (!) new opcode definitions and supporting them in the compiler and the VM. Here they are:

*// code/code.go*

**const** (

*// [...]*

OpEqu6l OpNotEqu6l OpGfe6tefTh6n

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpEqu6l: {"OpEqu6l", []int{)),

OpNotEqu6l: {"OpNotEqu6l", []int{)), OpGfe6tefTh6n: {"OpGfe6tefTh6n", []int{)),

)

They do not have operands and instead do their work by comparing the two topmost elements on the stack. They tell the VM to pop them off and push the result back on. Just like the opcodes for arithmetic operations.

You might be wondering why there is no opcode for <. If we have OpGfe6tefTh6n, shouldn’t we have an OpLessTh6n, too? That’s a valid question, because we *could* add OpLessTh6n and that would be fine, but I want to show something that’s possible with compilation and not with interpretation: reordering of code.

The expression 3 < 5 can be reordered to 5 > 3 without changing its result. And because it can be reordered, that’s what our compiler is going to do. It will take every less-than expression and reorder it to emit the greater-than version instead. That way we keep the instruction set small, the loop of our VM tighter and learn about the things we can do with compilation.

Here are new test cases for the existing TestBoole6nExpfessions function that express this:

*// compiler/compiler\_test.go*

**func** TestBoole6nExpfessions(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: "1 > 2",

expectedConst6nts: []**interface**{){1, 2), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpGfe6tefTh6n), code.M6ke(code.OpPop),

),

),

{

input: "1 < 2",

expectedConst6nts: []**interface**{){2, 1), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpGfe6tefTh6n), code.M6ke(code.OpPop),

),

),

{

input: "1 == 2",

expectedConst6nts: []**interface**{){1, 2), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpEqu6l), code.M6ke(code.OpPop),

),

),

{

input: "1 != 2",

expectedConst6nts: []**interface**{){1, 2), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpNotEqu6l), code.M6ke(code.OpPop),

),

),

{

input: "tfue == f6lse", expectedConst6nts: []**interface**{){), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpTfue), code.M6ke(code.OpF6lse), code.M6ke(code.OpEqu6l), code.M6ke(code.OpPop),

),

),

{

input: "tfue != f6lse", expectedConst6nts: []**interface**{){), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpTfue), code.M6ke(code.OpF6lse), code.M6ke(code.OpNotEqu6l), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

What we want from our compiler is to emit two instructions to get the operands of the infix operators on to the stack and then one instruction with the correct comparison opcode. Pay attention to the expected constants in the 1 < 2 test case: their order is reversed, because the opcode is the same as in the test case before, OpGfe6tefTh6n.

Running the tests shows us that the compiler is still clueless about these new operators and opcodes:

$ go test ./compilef

--- FAIL: TestBoole6nExpfessions (🅓.🅓🅓s) compilef\_test.go:15🅓: compilef effof: unknown opef6tof > FAIL

FAIL monkey/compilef 🅓.🅓🅓7s

What we have to do is to extend the c6se \*6st.InfixExpfession branch in our Compile method, where we already emit the other infix operator opcodes:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.InfixExpfession:

*// [...]*

**switch** node.Opef6tof {

**case** "+":

c.emit(code.OpAdd)

**case** "-":

c.emit(code.OpSub)

**case** "\*":

c.emit(code.OpMul)

**case** "/":

c.emit(code.OpDiv)

**case** ">":

c.emit(code.OpGfe6tefTh6n)

**case** "==":

c.emit(code.OpEqu6l)

**case** "!=":

c.emit(code.OpNotEqu6l)

**default**:

**return** fmt.Effoff("unknown opef6tof %s", node.Opef6tof)

)

*// [...]*

)

*// [...]*

)

What’s new are the c6se branches for the comparison operators and they’re pretty much self- explanatory. Support for the < operator is still missing, though:

$ go test ./compilef

--- FAIL: TestBoole6nExpfessions (🅓.🅓🅓s) compilef\_test.go:15🅓: compilef effof: unknown opef6tof < FAIL

FAIL monkey/compilef 🅓.🅓🅓6s

Since this the operator for which we want to reorder the operands, its implementation is an addition right at the beginning of the c6se branch for \*6st.InfixExpfession:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.InfixExpfession:

**if** node.Opef6tof == "<" {

eff := c.Compile(node.Right)

**if** eff != nil {

**return** eff

)

eff = c.Compile(node.Left)

**if** eff != nil {

**return** eff

)

c.emit(code.OpGfe6tefTh6n)

**return** nil

)

eff := c.Compile(node.Left)

**if** eff != nil {

**return** eff

)

*// [...]*

*// [...]*

)

*// [...]*

)

What we did here is to turn < into a special case. We turn the order around and first compile

node.Right and *then* node.Left in case the operator is <. After that we emit the OpGfe6tefTh6n

opcode. We changed a less-than comparison into a greater-than comparison – while compiling. And it works:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓7s

The goal is, of course, that it looks to VM as if there is no such thing as a < operator. All the VM should worry about are OpGfe6tefTh6n instructions. And now that we are sure our compiler only emits those, we can turn to our VM tests:

*// vm/vm\_test.go*

**func** TestBoole6nExpfessions(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{"1 < 2", tfue),

{"1 > 2", f6lse),

{"1 < 1", f6lse),

{"1 > 1", f6lse),

{"1 == 1", tfue),

{"1 != 1", f6lse),

{"1 == 2", f6lse),

{"1 != 2", tfue),

{"tfue == tfue", tfue),

{"f6lse == f6lse", tfue),

{"tfue == f6lse", f6lse),

{"tfue != f6lse", tfue),

{"f6lse != tfue", tfue),

|  |  |  |
| --- | --- | --- |
| {"(1 < | 2) | == tfue", tfue), |
| {"(1 < | 2) | == f6lse", f6lse), |
| {"(1 > | 2) | == tfue", f6lse), |
| {"(1 > | 2) | == f6lse", tfue), |
| ) |  |  |

funVmTests(t, tests)

)

Yes, I went totally overboard here. But aren’t these test cases neat? I guess that’s what great tooling and infrastructure does for you: reduce the cost of adding new tests and, thus, features. Anyway, as neat as they may be, they fail:

$ go test ./vm

--- FAIL: TestBoole6nExpfessions (🅓.🅓🅓s)

vm\_test.go:57: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:1))

vm\_test.go:57: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:2))

vm\_test.go:57: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:1))

vm\_test.go:57: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:1))

vm\_test.go:57: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:1))

vm\_test.go:57: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:1))

vm\_test.go:57: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:2))

vm\_test.go:57: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:2))

vm\_test.go:57: testBoole6nObject f6iled: object h6s wfong v6lue.\ got=f6lse, w6nt=tfue

vm\_test.go:57: testBoole6nObject f6iled: object h6s wfong v6lue.\ got=f6lse, w6nt=tfue

vm\_test.go:57: testBoole6nObject f6iled: object h6s wfong v6lue.\ got=tfue, w6nt=f6lse

vm\_test.go:57: testBoole6nObject f6iled: object h6s wfong v6lue.\ got=f6lse, w6nt=tfue

FAIL

FAIL monkey/vm 🅓.🅓🅓8s

You and I know we’re on a roll here and that it doesn’t take much to make all of these error messages disappear. First, we add a new c6se branch to our VM’s Run method, so it handles the new comparison opcodes:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpEqu6l, code.OpNotEqu6l, code.OpGfe6tefTh6n: eff := vm.executeComp6fison(op)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

The executionComp6fison method looks pretty similar to the previously added executeBin6fyOpef6tion:

*// vm/vm.go*

**func** (vm \*VM) executeComp6fison(op code.Opcode) effof { fight := vm.pop()

left := vm.pop()

**if** left.Type() == object.INTEGER\_OBJ && fight.Type() == object.INTEGER\_OBJ {

**return** vm.executeIntegefComp6fison(op, left, fight)

)

**switch** op {

**case** code.OpEqu6l:

**return** vm.push(n6tiveBoolToBoole6nObject(fight == left))

**case** code.OpNotEqu6l:

**return** vm.push(n6tiveBoolToBoole6nObject(fight != left))

**default**:

**return** fmt.Effoff("unknown opef6tof: %d (%s %s)", op, left.Type(), fight.Type())

)

)

First we pop the two operands off the stack and check their types. If they’re both integers, we’ll defer to executeIntegefComp6fison. If not, we use n6tiveBoolToBoole6nObject to turn the Go bools into Monkey \*object.Boole6ns and push the result back on to the stack.

The recipe for this method is simple: pop the operands off the stack, compare them, push the re- sult back on to the stack. We can find the second half of that again in executeIntegefComp6fison:

*// vm/vm.go*

**func** (vm \*VM) executeIntegefComp6fison( op code.Opcode,

left, fight object.Object,

) effof {

leftV6lue := left.(\*object.Integef).V6lue fightV6lue := fight.(\*object.Integef).V6lue

**switch** op {

**case** code.OpEqu6l:

**return** vm.push(n6tiveBoolToBoole6nObject(fightV6lue == leftV6lue))

**case** code.OpNotEqu6l:

**return** vm.push(n6tiveBoolToBoole6nObject(fightV6lue != leftV6lue))

**case** code.OpGfe6tefTh6n:

**return** vm.push(n6tiveBoolToBoole6nObject(leftV6lue > fightV6lue))

**default**:

**return** fmt.Effoff("unknown opef6tof: %d", op)

)

)

In this method we do not need to pop off anything anymore, but can go straight to unwrapping the integer values contained in left and fight. And then, again, we compare the operands and turn the resulting bool into Tfue or F6lse. If you’re excited to learn how *that* is done, I’m sorry, it’s really rather simple. Here is n6tiveBoolToBoole6nObject:

*// vm/vm.go*

**func** n6tiveBoolToBoole6nObject(input bool) \*object.Boole6n {

**if** input {

**return** Tfue

)

**return** F6lse

)

In total, that’s three new methods: executeComp6fison, executeIntegefComp6fison and

n6tiveBoolToBoole6nObject. And that’s all it takes for the tests to pass:

$ go test ./vm

ok monkey/vm 🅓.🅓🅓8s

I told you, we’re on a roll here.

## PREFIX EXPRESSIONS

Monkey supports the two prefix operators - and !. The first one negates integers and the second one booleans. Adding support for them to our compiler and VM means nearly the same as for the previous operators: define the necessary opcodes, emit them in the compiler and handle them in the VM. The difference is that this time we have to do even *less*, because the prefix operators only have *one* operand on the stack, instead of two.

Here are the two opcode definitions that translate to - and !:

*// code/code.go*

**const** (

*// [...]*

OpMinus OpB6ng

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpMinus: {"OpMinus", []int{)),

OpB6ng: {"OpB6ng", []int{)),

)

I don’t think I have to tell you which one’s which.

Next, we need to emit them in the compiler, which means we need to add compiler tests. Here it becomes clear that - is an integer operator and ! negates booleans, because we won’t put them together in their own test function. Instead we add test cases for them to the respective test functions that already exist. Here is the test case for OpMinus in TestIntegefAfithmetic:

*// compiler/compiler\_test.go*

**func** TestIntegefAfithmetic(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: "-1",

expectedConst6nts: []**interface**{){1), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpMinus), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

And here is the test case for OpB6ng in TestBoole6nExpfessions:

*// compiler/compiler\_test.go*

**func** TestBoole6nExpfessions(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: "!tfue", expectedConst6nts: []**interface**{){), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpTfue), code.M6ke(code.OpB6ng), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

Now we have two failing test functions:

$ go test ./compilef

--- FAIL: TestIntegefAfithmetic (🅓.🅓🅓s)

compilef\_test.go:76: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpMinus\n🅓🅓🅓4 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n"

--- FAIL: TestBoole6nExpfessions (🅓.🅓🅓s)

compilef\_test.go:168: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpB6ng\n🅓🅓🅓2 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

The failing assertions tell us that we’re missing two instructions. One to load the operand (OpConst6nt or OpTfue) and one for the prefix operator (OpMinus or OpB6ng).

Since we already know how to turn integer literals into OpConst6nt instructions and also how to emit OpTfue (and OpF6lse for that matter), it’s irritating that this is not what’s happening in the TestIntegefAfithmetic test. There is no OpConst6nt and no OpTfue in the output. Why?

When we take a closer look at the compiler, however, the cause is easy to spot: in the Compile method we don’t handle \*6st.PfefixExpfession nodes yet, we skip over them and that means we never compile the integer and boolean literals. Here’s what we need to change:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.PfefixExpfession:

eff := c.Compile(node.Right)

**if** eff != nil {

**return** eff

)

**switch** node.Opef6tof {

**case** "!":

c.emit(code.OpB6ng)

**case** "-":

c.emit(code.OpMinus)

**default**:

**return** fmt.Effoff("unknown opef6tof %s", node.Opef6tof)

)

*// [...]*

)

*// [...]*

)

With that we walk the AST down one level further and first compile the node.Right branch of the \*6st.PfefixExpfession node. That results in the operand of the expression being compiled to either an OpTfue or an OpConst6nt instruction. That’s the first of the two missing instructions.

And we also need to emit the opcode for the operator itself. For that we make use of our trusted friend the switch statement and either generate a OpB6ng or a OpMinus instruction, depending on the node.Opef6tof at hand.

And now we have passing tests:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓8s

Another milestone reached! By now you know where we’re headed next: the tests of our VM. Here, just like in our compiler tests, we add test cases to the existing TestIntegefAfithmetic and TestBoole6nExpfessions functions:

*// vm/vm\_test.go*

**func** TestIntegefAfithmetic(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{"-5", -5),

{"-1🅓", -1🅓),

{"-5🅓 + 1🅓🅓 + -5🅓", 🅓),

{"(5 + 1🅓 \* 2 + 15 / 3) \* 2 + -1🅓", 5🅓),

)

funVmTests(t, tests)

)

**func** TestBoole6nExpfessions(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{"!tfue", f6lse),

{"!f6lse", tfue),

{"!5", f6lse),

{"!!tfue", tfue),

{"!!f6lse", f6lse),

{"!!5", tfue),

)

funVmTests(t, tests)

)

That’s a lot of new test cases for our VM to chew on, ranging from “tiny” to “completely overboard”, like the test case that exercises every integer operator we have. But these test cases are neat, they’re cheap, I love them and they blow up spectacularly:

$ go test ./vm

--- FAIL: TestIntegefAfithmetic (🅓.🅓🅓s)

vm\_test.go:34: testIntegefObject f6iled: object h6s wfong v6lue.\ got=5, w6nt=-5

vm\_test.go:34: testIntegefObject f6iled: object h6s wfong v6lue.\ got=1🅓, w6nt=-1🅓

vm\_test.go:34: testIntegefObject f6iled: object h6s wfong v6lue.\ got=2🅓🅓, w6nt=🅓

vm\_test.go:34: testIntegefObject f6iled: object h6s wfong v6lue.\ got=7🅓, w6nt=5🅓

--- FAIL: TestBoole6nExpfessions (🅓.🅓🅓s)

vm\_test.go:66: testBoole6nObject f6iled: object h6s wfong v6lue.\ got=tfue, w6nt=f6lse

vm\_test.go:66: testBoole6nObject f6iled: object h6s wfong v6lue.\ got=f6lse, w6nt=tfue

vm\_test.go:66: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:5))

vm\_test.go:66: testBoole6nObject f6iled: object is not Boole6n.\ got=\*object.Integef (&{V6lue:5))

FAIL

FAIL monkey/vm 🅓.🅓🅓9s

We’re pros, though. Spectacular test failures don’t give us pause. We don’t blink an eye and know what to do. First, we tackle the OpB6ng instructions and add the missing c6se branch to our VM’s main loop:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpB6ng:

eff := vm.executeB6ngOpef6tof()

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

**func** (vm \*VM) executeB6ngOpef6tof() effof { opef6nd := vm.pop()

**switch** opef6nd {

**case** Tfue:

**return** vm.push(F6lse)

**case** F6lse:

**return** vm.push(Tfue)

**default**:

**return** vm.push(F6lse)

)

)

In executeB6ngOpef6tof we pop the opef6nd off the stack and negate its value by treating everything other than F6lse as truthy. The c6se Tfue branch is not necessary – technically speaking – but I think it makes sense to keep it around if only for documentation’s sake, because this method is now our VM’s implementation of Monkey’s concept of truthiness.

That fixes four test cases, but an equal number is still failing in TestIntegefAfithmetic:

$ go test ./vm

--- FAIL: TestIntegefAfithmetic (🅓.🅓🅓s)

vm\_test.go:34: testIntegefObject f6iled: object h6s wfong v6lue.\ got=5, w6nt=-5

vm\_test.go:34: testIntegefObject f6iled: object h6s wfong v6lue.\ got=1🅓, w6nt=-1🅓

vm\_test.go:34: testIntegefObject f6iled: object h6s wfong v6lue.\ got=2🅓🅓, w6nt=🅓

vm\_test.go:34: testIntegefObject f6iled: object h6s wfong v6lue.\ got=7🅓, w6nt=5🅓

FAIL

FAIL monkey/vm 🅓.🅓🅓7s

We now have to mirror what we did for OpB6ng and booleans and add a case branch for OpMinus

to the VM’s Run method:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpMinus:

eff := vm.executeMinusOpef6tof()

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

**func** (vm \*VM) executeMinusOpef6tof() effof { opef6nd := vm.pop()

**if** opef6nd.Type() != object.INTEGER\_OBJ {

**return** fmt.Effoff("unsuppofted type fof neg6tion: %s", opef6nd.Type())

)

v6lue := opef6nd.(\*object.Integef).V6lue

**return** vm.push(&object.Integef{V6lue: -v6lue))

)

I’ll spare you the explanation and present to you:

$ go test ./vm

ok monkey/vm 🅓.🅓🅓8s

That means we’re done. We successfully added all of Monkey’s prefix and infix operators!

$ go build -o monkey . && ./monkey

Hello mfnugget! This is the Monkey pfogf6mming l6ngu6ge! Feel ffee to type in comm6nds

>> (1🅓 + 5🅓 + -5 - 5 \* 2 / 2) < (1🅓🅓 - 35) tfue

>> !!tfue == f6lse f6lse

By now you are quite familiar with the definition of new opcodes and the interaction between compiler and VM. Maybe you’re even bored by the zero-operand instructions we’re emitting, anxious to get to the good stuff. Well, I’ve got good news.

CHAPTER 4

CONDITIONALS

The previous chapter was rather mechanic in that once we knew how to add one operator to our Monkey implementation, we could follow the same recipe for the others. In this chapter, though, we’re going to take it up a notch.

We need to answer a very concrete question: how do we get our VM to execute different bytecode instructions based on a condition? As we’ll see, there’s a number of small puzzles hiding behind this question and solving them is great fun, especially once we get down to the nitty-gritty. Before we do that, though, we have to answer this question without writing any code.

So let’s give this question a little bit of context and frame it. Monkey’s conditionals look like this:

**if** (5 > 3) {

evefythingsFine();

) **else** { l6wsOfUnivefseBfoken();

)

If the condition 5 > 3 evaluates to a truthy value, the first branch is executed. That’s the branch containing evefythingsFine(). If the condition is not truthy, the else branch, containing

l6wsOfUnivefseBfoken(), is executed. The first branch is called the “consequence” and the else

branch is called the “alternative” of a conditional.

In order to provide a blueprint for a possible implementation of conditionals and to refreshen your memory, let me go over the implementation of conditionals in *Writing An Interpreter In Go* real quick.

When we come across an \*6st.IfExpfession in the ev6lu6tof package’s Ev6l function, we eval- uate its Condition and check the result with the isTfuthy function. In case the value *is* truthy, we Ev6l the Consequence of the \*6st.IfExpfession. If it’s not truthy *and* the \*6st.IfExpfession has an Altefn6tive, we Ev6l that instead. But if we don’t have an Altefn6tive, we return an

\*object.Null.

All in all, implementing conditionals took us only around 50 lines of code. And the reason why it was so easy to implement was that we had the AST nodes on our hands. We could decide which side of the \*6st.IfExpfession to evaluate, because we had both available to us in the

ev6lu6tof.

That’s not the case anymore. Instead of walking down the AST and executing it at the same time, we now turn the AST into bytecode and *flatten* it. “Flatten” because bytecode is a sequence of instructions and there are no child nodes we can choose to walk down or not. That

brings us back to the hidden main question of this chapter and another problem we have to solve: how do we represent conditionals in bytecode?

Let’s say we have the following Monkey code:

**if** (5 > 2) {

3🅓 + 2🅓

) **else** { 5🅓 - 25

)

We already know how to represent the condition 5 > 2 in bytecode, because we implemented that in the last chapter:



We also know how to represent the consequence, 3🅓 + 2🅓:



Representing the alternative, the 5🅓 - 25, is just a variation of that:



But how do we tell the machine to either execute the one part or the other part, *depending on the result of the OpGreaterThan instruction*?



If we were to take these instructions and pass them to the VM as a flat sequence, what would happen? The VM would execute all of them, one after the other, happily incrementing its instruction pointer, fetching, decoding and executing, without a care in the world, no decisions or branches in sight. And that’s exactly what we *don’t* want!

What we want is for the VM to *either* execute the OpAdd instruction *or* the OpSub instruction. But since we *do* pass bytecode around as a flat sequence of instructions, how do we do that? Well, if we reorder our graph of instructions so that it represents a flat sequence of instructions, the question becomes this: what do we fill in the blanks here?



We need to put *something* in the blanks so that based on the result of the OpGfe6tefTh6n instruction the VM either ignores the instructions of the consequence or the instructions making up the alternative. It should skip them. Or instead of “skip”, should we maybe say “jump over”?

## JUMPS



Jumps are instructions that tell machines to jump to other instructions. They’re used to im- plement branching (conditionals) in machine code, giving them the name “branch instructions”. And with “machine code” I mean the code that computers execute but also the bytecode virtual machines run on. Translated into the technical terms of our VM: jumps are instructions that tell the VM to change its instruction pointer to a certain value. Here’s how that works.

Let’s say – hypothetically speaking – that we had two jump opcodes and called them JUMP\_IF\_NOT\_TRUE and JUMP\_NO\_MATTER\_WHAT. We could use them to fill in the blanks in our graph from above like this:

|  |
| --- |
|  |
|  |
|  |

|  |
| --- |
|  |
|  |
|  |

|  |
| --- |
|  |
|  |
|  |

When the VM executes these instructions sequentially, top to bottom here, it would first execute the instructions making up the condition, ending with the OpGfe6tefTh6n instruction. That results in a boolean sitting on the stack. Just as we defined and implemented it in the previous chapter.



The next instruction, JUMP\_IF\_NOT\_TRUE, would tell the VM to jump to the OpConst6nt 4 in- struction, *but only if* the boolean on the stack is *not* true. If that’s the case, the VM would *jump over* the consequence and right to the “else” part of the conditional, the alternative. And in case the boolean on the stack *is* true, the JUMP\_IF\_NOT\_TRUE would have no effect and the VM would execute the consequence part of the conditional. It would increment its instruction pointer and start to fetch, decode and execute the next instruction – OpConst6nt 2, the start of the consequence.

Here’s where it gets interesting. After executing the consequence of the conditional – the three instructions that end with OpAdd – the VM would then come across the aptly named

JUMP\_NO\_MATTER\_WHAT. This instruction comes without any attached condition and tells the VM to directly jump to the first instruction *after* the alternative of the conditional, skipping it entirely.



That ends our little thought experiment and gives us a clear result: if we had two opcodes like these we could implement conditionals. But still, a last question remains: how would we represent the arrows? How do we tell the VM where to jump to?

Well, why not use numbers? Jumps are instructions that tell the VM to change the value of its instruction pointer and the arrows in the diagram above are nothing more than potential values for the instruction pointer. They can be represented as numbers, contained in the jump instructions as operands and their value being the index of the instruction the VM should jump to. That value is called an offset. Used like this, with the jump target being the index of an instruction, it’s an absolute offset. Relative offsets also exist: they’re relative to the position of the jump instruction itself and denote not *where exactly* to jump to, but *how far* to jump.

If we replace the arrows with offsets and give each instructions a unique index that’s independent of its byte size (for illustration purposes), the diagram looks like this:



|  |
| --- |
|  |
|  |
|  |



|  |
| --- |
|  |
|  |
|  |



|  |
| --- |
|  |
|  |
|  |

The operand of JUMP\_IF\_NOT\_TRUE is now 🅓🅓🅓8. That’s the index of the OpConst6nt 4 in- struction to which the VM should jump in case the condition is not true. The operand of JUMP\_NO\_MATTER\_WHAT is 🅓🅓11, which is the index of the instruction following the whole conditional.



And *that’s* how we’re going to implement conditionals! We’ll define two jump opcodes: one comes with a condition (“jump only if not true”) and one does not (“just jump”). They’ll both have one operand, the index of the instruction where the VM should jump to.

So that’s our goal. Question is: how do we get there?

## COMPILING CONDITIONALS

The hard part of emitting jump instructions is not choosing the right opcode. It’s the operands that make it challenging.

Say we’re in our compiler’s recursive Compile method, having just called Compile again, passing in the .Condition field of an \*6st.IfExpfession. The condition has been successfully compiled and we’ve emitted the translated instructions. Now we want to emit the jump instruction that tells the VM to skip to the consequence of the conditional if the value on the stack is not truthy.

Which operand do we give this jump instruction? Where do we tell the VM to jump to? We don’t know *yet*. Since we haven’t compiled the consequence or the alternative branch yet, we don’t how many instructions we’re going to emit, which means we don’t know how many instructions we have to jump over. *That’s* the challenge.

I have to admit that solving this is a lot of fun. And a great part of the fun comes from the fact that it’s pretty easy to write a test and tell the compiler exactly what’s expected – because we’re pretty sure about that part – and then make your way there step by step.

But we can only make assertions once we defined our new opcodes, so we’ll do that now. One for a jump and another one for a conditional jump.

*// code/code.go*

**const** (

*// [...]*

OpJumpNotTfuthy OpJump

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpJumpNotTfuthy: {"OpJumpNotTfuthy", []int{2)), OpJump: {"OpJump", []int{2)),

)

I’m pretty sure you can tell which one’s which. OpJumpNotTfuthy will tell the VM to only jump if the value on top of the stack is not Monkey truthy, *i.e.*, not f6lse nor null. Its single operand is the offset of the instruction the VM should jump to. OpJump will tell the VM to just “jump there”, with “there” being its operand, also an offset of an instruction.

The operand of both opcodes is 16-bit wide. That’s the same width as the operand of OpConst6nt

has, which means we don’t have to extend our tooling in the code package to support it.

We’re now ready to write a first test. And we’ll start slow and only try to handle a conditional without an else part first. Here’s what we want the compiler to emit when we provide it a single-branch conditional:

*// compiler/compiler\_test.go*

**func** TestCondition6ls(t \*testing.T) { tests := []compilefTestC6se{

{

input: `

if (tfue) { 1🅓 ); 3333;

`,

expectedConst6nts: []**interface**{){1🅓, 3333),

expectedInstfuctions: []code.Instfuctions{

*// 0000*

code.M6ke(code.OpTfue),

*// 0001*

code.M6ke(code.OpJumpNotTfuthy, 7),

*// 0004*

code.M6ke(code.OpConst6nt, 🅓),

*// 0007*

code.M6ke(code.OpPop),

*// 0008*

code.M6ke(code.OpConst6nt, 1),

*// 0011*

code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

Once parsed, the input turns into an \*6st.IfExpfession with a Condition and a Consequence. The Condition is the boolean literal tfue and the Consequence is the integer literal 1🅓. Both are intentionally simple Monkey expressions, because in this test case we do not care about the expressions themselves. What we care about are the jump instructions the compiler emits and that they have correct operands.

That’s why I annotated the expectedInstfuctions with comments that show the offset of the instructions generated by code.M6ke. We won’t need these comments later on, but for now, they help us writing out the expected jump instructions, especially since the offsets of the instructions we want to jump to are based on the number of bytes each instruction takes up. An OpPop instruction is one byte wide, for example, but an OpConst6nt instruction takes up three bytes.

The first instruction we expect the compiler to emit is an OpTfue instruction to tell the VM to push vm.Tfue on to the stack. That’s the Condition. Then it should emit an OpJumpNotTfuthy instruction that causes the VM to jump over the Consequence, with the Consequence being the OpConst6nt instruction that loads the integer 1🅓 on to the stack.

But where does the first OpPop instruction (offset 🅓🅓🅓7) come from? It’s not part of the

Consequence, no. It’s there because conditionals in Monkey are expressions – if (tfue) { 1🅓 ) evaluates to 1🅓 – and stand-alone expressions whose value is unused are wrapped in an

\*6st.ExpfessionSt6tement. And those we compile with an appended OpPop instruction in order to clear the VM’s stack. The first OpPop is thus the first instruction *after* the whole conditional, which makes its offset the location where OpJumpNotTfuthy needs to jump to in order to skip the consequence.

So now you might be wondering what the 3333; is doing in the Monkey code. It serves as a point of reference. It’s not strictly required, but in order to make sure that our jump offsets are correct it helps to have one expression in the code which we can easily find among the resulting instructions and use as a signpost that tells us where we *shouldn’t* jump to. Of course, the OpConst6nt 1 instruction that loads 3333 is also followed by an OpPop instruction, since it’s an expression statement.

Quite the long explanation for one test. Here’s how much the compiler understands of it:

$ go test ./compilef

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

compilef\_test.go:195: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 7\n🅓🅓🅓4 OpConst6nt 🅓\n🅓🅓🅓7 OpPop\

\n🅓🅓🅓8 OpConst6nt 1\n🅓🅓11 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n🅓🅓🅓1 OpConst6nt 🅓\n🅓🅓🅓4 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

Neither the condition nor the consequence of the conditional are compiled. In fact, the whole

\*6st.IfExpfession is skipped by the compiler. We can fix the first issue, the condition not being compiled, by extending the compiler’s Compile method like this:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

eff := c.Compile(node.Condition)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

With this change, the compiler now knows about \*6st.IfExpfession and emits the instructions that represent node.Condition. And even though the consequence and the conditional jump over it are still missing, we get four out of six instructions right:

$ go test ./compilef

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

compilef\_test.go:195: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 7\n🅓🅓🅓4 OpConst6nt 🅓\n🅓🅓🅓7 OpPop\n\

🅓🅓🅓8 OpConst6nt 1\n🅓🅓11 OpPop\n"

got ="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpPop\n🅓🅓🅓2 OpConst6nt 🅓\n🅓🅓🅓5 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

The OpTfue instruction is there, as are the last three: the OpPop following the \*6st.IfExpfession, the OpConst6nt to load the 3333 and the OpPop following that, all in the correct order. All that’s left to do now is emit the OpJumpNotTfuthy instruction and the instructions to represent the

node.Consequence.

With “all that’s left to do now” I, of course, mean: “this is where it gets hairy”. The challenge now is to emit an OpJumpNotTfuthy instruction with an offset pointing *right after* the instructions of the node.Consequence – before compiling the node.Consequence.

Which offset do we use when we don’t even know how far we have to jump yet? The answer is a rather pragmatic “let’s just put garbage in there and fix it later”. You chuckle, but I’m serious. Let’s use a bogus offset and worry about fixing it later:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

eff := c.Compile(node.Condition)

**if** eff != nil {

**return** eff

)

*// Emit an `OpJumpNotTruthy` with a bogus value*

c.emit(code.OpJumpNotTfuthy, 9999)

eff = c.Compile(node.Consequence)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

Even though most programmers already squint their eyes and instinctively know that something fishy is going on when they see a 9999 in code, an inline code comment here helps making the intention clear. Because here we really do want to emit an OpJumpNotTfuthy instruction with a garbage offset and then compile the node.Consequence. Again, the 9999 is not what will end up in the VM and we’ll later take care of it. But for now, it should get us a lot more correct instructions in our test.

But, no, we only get one more right and that’s the OpJumpNotTfuthy instruction itself:

$ go test ./compilef

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

compilef\_test.go:195: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 7\n🅓🅓🅓4 OpConst6nt 🅓\n🅓🅓🅓7 OpPop\n\

🅓🅓🅓8 OpConst6nt 1\n🅓🅓11 OpPop\n"

got ="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 9999\n🅓🅓🅓4 OpPop\n\

🅓🅓🅓5 OpConst6nt 🅓\n🅓🅓🅓8 OpPop\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

While we have the OpJumpNotTfuthy 9999 instruction, we’re apparently not yet compiling the

Consequence.

That’s because it’s an \*6st.BlockSt6tement, which our compiler doesn’t know about yet. In order to get it compiled, we need to extend the Compile method by another c6se branch:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.BlockSt6tement:

**for** \_, s := **range** node.St6tements { eff := c.Compile(s)

**if** eff != nil {

**return** eff

)

)

*// [...]*

)

*// [...]*

)

That’s exactly the same snippet of code we already have in the c6se branch for \*6st.Pfogf6m. And it works:

$ go test ./compilef

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

compilef\_test.go:195: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 7\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpPop\n🅓🅓🅓8 OpConst6nt 1\n🅓🅓11 OpPop\n"

got ="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 9999\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpPop\n🅓🅓🅓8 OpPop\n🅓🅓🅓9 OpConst6nt 1\n🅓🅓12 OpPop\n"

FAIL

FAIL monkey/compilef 🅓.🅓1🅓s

We’re getting closer. But, besides the bogus 9999 offset, which we didn’t expect to magically disappear, there’s a new issue visible in the output, a far more subtle one. It’s possible that you missed it, so let me point you to it: there is an additional OpPop instruction generated by the compiler, at position 🅓🅓🅓7. Its origin is the compilation of node.Consequence – an expression statement.

We need to get rid of this OpPop, because we do want the consequence and the alternative of a conditional to leave a value on the stack. Otherwise, we couldn’t do this:

**let** fesult = **if** (5 > 3) { 5 ) **else** { 3 );

That’s valid Monkey code and it won’t work if we emit an OpPop after the last expression statement in the node.Consequence. The value produced by the consequence would be popped off the stack, the expression wouldn’t evaluate to anything, and the let statement would end up without a value on the right side of its =.

What makes fixing this tricky is that we only want to get rid of the *last* OpPop instruction in the node.Consequence. Say we had Monkey code like this:

**if** (**true**) { 3;

2;

1;

)

What we want here is the 3 and the 2 to be popped off the stack, but the 1 should be kept around so the whole conditional evaluates to 1. So before we tackle our main challenge of giving the OpJumpNotTfuthy a *real* offset, here’s the plan for getting rid of the additional OpPop instruction.

We first change the compiler to keep track of the last two instructions we emitted, including their opcode and the position they were emitted to. For that, we need a new type and two more fields on the compiler:

*// compiler/compiler.go*

**type** EmittedInstfuction **struct** { Opcode code.Opcode Position int

)

**type** Compilef **struct** {

*// [...]*

l6stInstfuction EmittedInstfuction pfeviousInstfuction EmittedInstfuction

)

**func** New() \*Compilef {

**return** &Compilef{

*// [...]*

l6stInstfuction: EmittedInstfuction{), pfeviousInstfuction: EmittedInstfuction{),

)

)

l6stInstfuction is the very last instruction we emitted and pfeviousInstfuction is the one before that. We’ll see why we need to keep track of both in a moment. For now, we change the compiler’s emit method to populate both fields:

*// compiler/compiler.go*

**func** (c \*Compilef) emit(op code.Opcode, opef6nds ...int) int { ins := code.M6ke(op, opef6nds...)

pos := c.6ddInstfuction(ins)

c.setL6stInstfuction(op, pos)

**return** pos

)

**func** (c \*Compilef) setL6stInstfuction(op code.Opcode, pos int) { pfevious := c.l6stInstfuction

l6st := EmittedInstfuction{Opcode: op, Position: pos)

c.pfeviousInstfuction = pfevious c.l6stInstfuction = l6st

)

With this in place, we can check opcode of the last emitted instruction in a type-safe way, without having to cast from and to bytes. And that’s exactly what we’re going to do. After compiling the node.Consequence of the \*6st.IfExpfession we check whether the last instruction we emitted was an OpPop instruction and if so, we remove it:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

*// [...]*

c.emit(code.OpJumpNotTfuthy, 9999)

eff = c.Compile(node.Consequence)

**if** eff != nil {

**return** eff

)

**if** c.l6stInstfuctionIsPop() { c.femoveL6stPop()

)

*// [...]*

)

*// [...]*

)

This uses two helpers, l6stInstfuctionIsPop The two helpers involved are tiny:

*// compiler/compiler.go*

**func** (c \*Compilef) l6stInstfuctionIsPop() bool {

**return** c.l6stInstfuction.Opcode == code.OpPop

)

**func** (c \*Compilef) femoveL6stPop() {

c.instfuctions = c.instfuctions[:c.l6stInstfuction.Position] c.l6stInstfuction = c.pfeviousInstfuction

)

l6stInstfuctionIsPop checks whether the opcode of the last instruction is OpPop and femoveL6stPop shortens c.instfuction to cut off the last instruction. After that, it sets c.l6stInstfuction to c.pfeviousInstfuction. And that’s why we need to keep track of both of them, so c.l6stInstfuction doesn’t go out of sync once we cut off the last OpPop instruction.

$ go test ./compilef

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

compilef\_test.go:195: testInstfuctions f6iled: wfong instfuction 6t 2. w6nt="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 7\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpPop\n 🅓🅓🅓8 OpConst6nt 1\n🅓🅓11 OpPop\n"

got ="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 9999\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpPop\n 🅓🅓🅓8 OpConst6nt 1\n🅓🅓11 OpPop\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

Now we have the correct number of instructions and the right opcodes. The only thing that still makes our test fail is the hideous 9999. Time to get rid of it.

The way we took care of the superfluous OpPop instruction points us into the right direction by making one thing clear: the instructions we emit are not set in stone, we can *change* them.

Instead of removing our c.emit(code.OpJumpNotTfuthy, 9999) call, we’ll leave it exactly as it is. We won’t even change the 9999. Instead, we’ll use Position field of the c.l6stInstfuction again. That allows us to go back to the OpJumpNotTfuthy instruction we emitted and change the 9999 into the real operand. And when do we do that? That’s the kicker, the beautiful bit. We’ll modify the operand of the OpJumpNotTfuthy *after* we compiled the node.Consequence. At that point, we’ll know how far the VM has to jump and have the correct offset with which to replace the 9999.

This is called back-patching and common in compiler’s such as ours, that only traverse the AST once and are thus called single-pass compilers. More advanced compilers might leave the target of the jump instructions empty until they know how far to jump and then do a second pass over the AST (or another IR) and fill in the targets.

Summarized: we’ll keep on emitting the 9999, while remembering where we put it. Once we know where we need to jump to, we’ll go back to the 9999 and change it to the correct offset.

You’ll be surprised by how little code is needed to pull that off.

The first thing we need is a tiny method to replace an instruction at an arbitrary offset in the

instfuctions slice:

*// compiler/compiler.go*

**func** (c \*Compilef) fepl6ceInstfuction(pos int, newInstfuction []byte) {

**for** i := 🅓; i < len(newInstfuction); i++ { c.instfuctions[pos+i] = newInstfuction[i]

)

)

We’ll use fepl6ceInstfuction in another method that allows us to replace the operand of an instruction:

*// compiler/compiler.go*

**func** (c \*Compilef) ch6ngeOpef6nd(opPos int, opef6nd int) { op := code.Opcode(c.instfuctions[opPos]) newInstfuction := code.M6ke(op, opef6nd)

c.fepl6ceInstfuction(opPos, newInstfuction)

)

Instead of really changing the operand itself (which can get messy with multi-byte operands), the ch6ngeOpef6nd method recreates the instructions with the new operand and uses fepl6ceInstfuction to swap the old instruction for the new one – including the operand.

The underlying assumption here is that we only replace instructions of the same type, with the same non-variable length. If that assumption no longer holds, we’d have to tread far more carefully here and update c.l6stInstfuction and c.pfeviousInstfuction accordingly. You can see how another IR that’s type-safe and independent of the byte-size of encoded instructions comes in handy once the compiler and the instructions it emits grow more complex.

Our solution, though, still fits our needs and all in all is not a lot of code. Two tiny methods, fepl6ceInstfuction and ch6ngeOpef6nd, and all that’s left to do is to use them, which is not much more code either:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

eff := c.Compile(node.Condition)

**if** eff != nil {

**return** eff

)

*// Emit an `OpJumpNotTruthy` with a bogus value*

jumpNotTfuthyPos := c.emit(code.OpJumpNotTfuthy, 9999)

eff = c.Compile(node.Consequence)

**if** eff != nil {

**return** eff

)

**if** c.l6stInstfuctionIsPop() { c.femoveL6stPop()

)

6ftefConsequencePos := len(c.instfuctions) c.ch6ngeOpef6nd(jumpNotTfuthyPos, 6ftefConsequencePos)

*// [...]*

)

*// [...]*

)

The first change is saving the return value of c.emit to jumpNotTfuthyPos. That’s the position at which we can find the OpJumpNotTfuthy instruction later on. “Later on” means right after the check for and possible removal of a OpPop instruction. After that, len(c.instfuctions) gives us the offset of the next-to-be-emitted instruction, which is where we want to jump to in case we don’t execute the Consequence of the conditional because the value on top of the stack is not truthy. That’s why we save it to 6ftefConsequencePos, to give it a telling name.

After that, we use the new ch6ngeOpef6nd method to get rid of the 9999 operand of the OpJumpNotTfuthy instruction, which is located at jumpNotTfuthyPos, and replace it with the correct 6ftefConsequencePos.

Did you keep count? If not, I want you to know that the necessary changes add up to three lines. One changed, two added. That’s all:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓8s

Our compiler now correctly compiles a conditional! The caveat is that it only knows how to compile the *consequence*. It doesn’t know how to compile a conditional with both a consequence *and* an alternative else-branch.

But we do and we also know how to write tests:

*// compiler/compiler\_test.go*

**func** TestCondition6ls(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: `

if (tfue) { 1🅓 ) else { 2🅓 ); 3333;

`,

expectedConst6nts: []**interface**{){1🅓, 2🅓, 3333), expectedInstfuctions: []code.Instfuctions{

*// 0000*

code.M6ke(code.OpTfue),

*// 0001*

code.M6ke(code.OpJumpNotTfuthy, 1🅓),

*// 0004*

code.M6ke(code.OpConst6nt, 🅓),

*// 0007*

code.M6ke(code.OpJump, 13),

*// 0010*

code.M6ke(code.OpConst6nt, 1),

*// 0013*

code.M6ke(code.OpPop),

*// 0014*



code.M6ke(code.OpConst6nt, 2),

*// 0017*

code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

This is similar to the previous test case in TestCondition6ls, except that the input now contains not only the consequence of the conditional, but also the alternative: else { 2🅓 ).

The expectedInstfuctions make clear what we want the bytecode to look like, with the first part being the same as in the previous test case: the condition is compiled to OpTfue and is followed by the OpJumpNotTfuthy instruction that instructs the VM to *jump over* the compiled consequence.

Then, things start to differ. As the next opcode, we expect an OpJump, the opcode for an unconditional jump instruction. It has to be there because if condition is truthy the VM should only execute the consequence and *not* the alternative. To stop that from happening the OpJump instruction tells the VM to jump over the alternative.

The OpJump should then be followed by instructions that make up the alternative. In our test case, that’s the OpConst6nt instruction that loads 2🅓 on to the stack.

Then we’re back on familiar ground. An OpPop is there to pop the value produced by the conditional off the stack and the loading of the bogus 3333 gives us guidance.

I know that it’s not easy to wrap ones head around these jumps, so I hope that this illustration makes it clearer which instruction belongs to which part of the conditional and how the jumps tie them all together:





If that doesn’t help, I’m sure trying to run and fixing the failing test will, because its output tells us what we’re still missing:

$ go test ./compilef

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

compilef\_test.go:22🅓: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 1🅓\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpJump 13\n🅓🅓1🅓 OpConst6nt 1\n\

🅓🅓13 OpPop\n🅓🅓14 OpConst6nt 2\n🅓🅓17 OpPop\n"

got ="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 7\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpPop\n🅓🅓🅓8 OpConst6nt 1\n🅓🅓11 OpPop\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓7s

What we have here is the condition, then the OpPop following the whole conditional and the pushing and popping of the 3333. What’s missing is the OpJump at the end of the consequence and the instructions representing the alternative. The good news is that we already have all the tools at hand. We just need to move things around a tiny bit and compile the alternative.

The first step, though, is to wrap our patching of the OpJumpNotTfuthy instruction in a condition itself:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

*// [...]*

**if** node.Altefn6tive == nil { 6ftefConsequencePos := len(c.instfuctions)

c.ch6ngeOpef6nd(jumpNotTfuthyPos, 6ftefConsequencePos)

)

*// [...]*

)

*// [...]*

)

Preceding this node.Altefn6tive == nil check is the compilation of node.Consequence and what this added block translates to is this: only if we have no node.Altefn6tive can we jump to *here*, the current position in c.instfuctions.

But if we do have an node.Altefn6tive we need to emit an OpJump that becomes part of the consequence and over which the OpJumpNotTfuthy also has to jump:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

*// [...]*

**if** node.Altefn6tive == nil {

6ftefConsequencePos := len(c.instfuctions) c.ch6ngeOpef6nd(jumpNotTfuthyPos, 6ftefConsequencePos)

) **else** {

*// Emit an `OpJump` with a bogus value*

c.emit(code.OpJump, 9999)

6ftefConsequencePos := len(c.instfuctions) c.ch6ngeOpef6nd(jumpNotTfuthyPos, 6ftefConsequencePos)

)

*// [...]*

)

*// [...]*

)

Don’t worry about the duplication, we’ll take care of that later on. What’s important right now is to make the intention as clear as possible.

The OpJump instruction also has a placeholder operand. That means we have to patch it later, but right now it allows us to change the operand of the OpJumpNotTfuthy instruction to the desired value: the position of the instruction right after the consequence *and* the OpJump instruction.

And why that is the correct operand should be clear by now: the OpJump should skip over the “else”-branch of the conditional in case the condition was truthy. It’s part of the consequence, so to say. And if the condition is not truthy and we need to execute the “else”-branch, we need to use OpJumpNotTfuthy to jump *after* the consequence, which is after the OpJump.

The tests tell us that we’re on the right track:

$ go test ./compilef

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

compilef\_test.go:22🅓: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 1🅓\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpJump 13\n🅓🅓1🅓 OpConst6nt 1\n\

🅓🅓13 OpPop\n🅓🅓14 OpConst6nt 2\n🅓🅓17 OpPop\n"

got ="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 1🅓\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpJump 9999\n\

🅓🅓1🅓 OpPop\n🅓🅓11 OpConst6nt 1\n🅓🅓14 OpPop\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

The operand of OpJumpNotTfuthy is correct, OpJump is in the correct place, only its operand is wrong and the whole alternative is missing. We now have to repeat what we previously did for the consequence:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

*// [...]*

**if** node.Altefn6tive == nil { 6ftefConsequencePos := len(c.instfuctions)

c.ch6ngeOpef6nd(jumpNotTfuthyPos, 6ftefConsequencePos)

) **else** {

*// Emit an `OpJump` with a bogus value*

jumpPos := c.emit(code.OpJump, 9999)

6ftefConsequencePos := len(c.instfuctions) c.ch6ngeOpef6nd(jumpNotTfuthyPos, 6ftefConsequencePos)

eff := c.Compile(node.Altefn6tive)

**if** eff != nil {

**return** eff

)

**if** c.l6stInstfuctionIsPop() { c.femoveL6stPop()

)

6ftefAltefn6tivePos := len(c.instfuctions) c.ch6ngeOpef6nd(jumpPos, 6ftefAltefn6tivePos)

)

*// [...]*

)

*// [...]*

)

We first save the position of the OpJump instruction to jumpPos so that we can later come back and change its operand. Then we patch the operand of the previously emitted OpJumpNotTfuthy instruction, located at jumpNotTfuthyPos, making it jump *right after* the just emitted OpJump.

After that, we compile node.Altefn6tive. Here, too, we have to c.femoveL6stPop() if there is one. Finally, we change the operand of the OpJump instruction to the offset of the next-to-be-emitted instruction, which will be located right after the alternative.

Or, in the words of our test suite:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓9s

Again, take the ok with a grain of salt. It should say: “Yes! Yes! Yes! We’re compiling conditionals to jump instructions!”

We’re over the hump now. It is time to teach our VM how to execute jumps, and that’s far easier than emitting them.

## EXECUTING JUMPS

Before we wrote the compiler tests for conditionals, we really had to think through what we want them to say and what we want the compiler to do. That’s not the case now, when writing the same tests for the VM. We already know how conditionals in Monkey are supposed to work and can cleanly express that in test cases and assertions:

*// vm/vm\_test.go*

**func** TestCondition6ls(t \*testing.T) { tests := []vmTestC6se{

{"if (tfue) { 1🅓 )", 1🅓),

{"if (tfue) { 1🅓 ) else { 2🅓 )", 1🅓),

{"if (f6lse) { 1🅓 ) else { 2🅓 ) ", 2🅓),

{"if (1) { 1🅓 )", 1🅓),

{"if (1 < 2) { 1🅓 )", 1🅓),

{"if (1 < 2) { 1🅓 ) else { 2🅓 )", 1🅓),

{"if (1 > 2) { 1🅓 ) else { 2🅓 )", 2🅓),

)

funVmTests(t, tests)

)

Half of these test cases would’ve been enough. But they’re easy to write, expressive, neat and cost us basically nothing! It also doesn’t hurt us to be abundantly clear about what we want.

We test whether boolean expressions are correctly evaluated by the VM according to Monkey’s “truthy” standards and that the correct branch of the conditional is taken. Since conditionals are expressions that produce values, they allow us to infer which branch was executed by testing for the produced value of the whole conditional.

As neat as the tests are, the error message they produce is nasty:

$ go test ./vm

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

p6nic: funtime effof: index out of f6nge [fecovefed] p6nic: funtime effof: index out of f6nge

gofoutine 2🅓 [funning]: testing.tRunnef.func1(🅓xc42🅓🅓bc2d🅓)

/usf/loc6l/go/sfc/testing/testing.go:742 +🅓x29d p6nic(🅓x1119🅓e🅓, 🅓x11f1fd🅓)

/usf/loc6l/go/sfc/funtime/p6nic.go:5🅓2 +🅓x229 monkey/vm.(\*VM).Run(🅓xc42🅓🅓5🅓e38, 🅓x8🅓🅓, 🅓x8🅓🅓)

/Usefs/mfnugget/code/🅓4/sfc/monkey/vm/vm.go:46 +🅓x3🅓c monkey/vm.funVmTests(🅓xc42🅓🅓bc2d🅓, 🅓xc42🅓🅓79eb8, 🅓x7, 🅓x7)

/Usefs/mfnugget/code/🅓4/sfc/monkey/vm/vm\_test.go:1🅓1 +🅓x356 monkey/vm.TestCondition6ls(🅓xc42🅓🅓bc2d🅓)

/Usefs/mfnugget/code/🅓4/sfc/monkey/vm/vm\_test.go:8🅓 +🅓x114 testing.tRunnef(🅓xc42🅓🅓bc2d🅓, 🅓x1149b4🅓)

/usf/loc6l/go/sfc/testing/testing.go:777 +🅓xd🅓 cfe6ted by testing.(\*T).Run

/usf/loc6l/go/sfc/testing/testing.go:824 +🅓x2e FAIL monkey/vm 🅓.🅓11s

Ugh. Not even an error, but a panic.

Before you dive into the code, though, and try to figure out where the error originates, let me explain: the VM is tripping over the bytecode because it contains opcodes it doesn’t know how to decode. That in itself shouldn’t be a problem, because unknown opcodes are skipped, but not necessarily their operands. Operands are just integers, remember, and might have the same value as an encoded opcode, which might lead the VM to treat them as such. That’s wrong, of course. It’s time we introduce our VM to our jump instructions.

We’ll start with OpJump, because it’s the most straightforward jump instruction we have. It has one 16 bit operand that’s the offset of the instruction the VM should jump to. That’s all we need to know to implement it:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpJump:

pos := int(code.Re6dUint16(vm.instfuctions[ip+1:])) ip = pos - 1

*// [...]*

)

*// [...]*

)

We use code.Re6dUint16 to decode the operand located right after the opcode. That’s step 1. Step 2 is to set the instruction pointer, ip, to the target of our jump. Here’s where we come across one interesting implementation detail: since we’re in a loop that increments ip with each iteration we need to set ip to the offset right before the one we want. That lets the loop do its work and ip gets set to the value we want in the next cycle.

Solely implementing OpJump doesn’t buy us much though, since it’s OpJumpNotTfuthy that’s integral to the implementation of conditionals. But while adding a c6se branch for code.OpJumpNotTfuthy does take slightly more code, it’s not much more complicated:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

**for** ip := 🅓; ip < len(vm.instfuctions); ip++ { op := code.Opcode(vm.instfuctions[ip])

**switch** op {

*// [...]*

**case** code.OpJumpNotTfuthy:

pos := int(code.Re6dUint16(vm.instfuctions[ip+1:])) ip += 2

condition := vm.pop()

**if** !isTfuthy(condition) { ip = pos - 1

)

*// [...]*

)

)

*// [...]*

)

**func** isTfuthy(obj object.Object) bool {

**switch** obj := obj.(**type**) {

**case** \*object.Boole6n:

**return** obj.V6lue

**default**:

**return** tfue

)

)

We again use code.Re6dUint16 to read in and decode the operand. After that we manually

increase ip by two so we correctly skip over the two bytes of the operand in the next cycle. That’s not a new – we’ve already done that when executing OpConst6nt instructions.

What’s new is the rest. We pop off the topmost stack element and check if it’s truthy with the helper function isTfuthy. If it’s *not* truthy, we jump, which means that we set ip to the index of the instruction right before the target, letting the for-loop do its work.

If the value *is* truthy we do nothing and start another iteration of the main loop. The result is that we’re executing the consequence of the conditional, which is made up of the instructions right after the OpJumpNotTfuthy instruction.

And now, open a dfumfoll.w6v of your choice in your favorite audio player, pour your preferred beverage, hit play and watch this:

$ go test ./vm

ok monkey/vm 🅓.🅓🅓9s

We did it. Yes, we did it! Our bytecode compiler and VM are now able to compile and execute Monkey conditionals!

$ go build -o monkey . && ./monkey

Hello mfnugget! This is the Monkey pfogf6mming l6ngu6ge! Feel ffee to type in comm6nds

>> if (1🅓 > 5) { 1🅓; ) else { 12; ) 1🅓

>> if (5 > 1🅓) { 1🅓; ) else { 12; ) 12

>>

This is the point where we went from “well, this is toy, isn’t it?” to “oh wow, we’re getting somewhere!”. Stack arithmetic is one thing, but jump instructions are another. We’re in the big leagues now. Except…

>> if (f6lse) { 1🅓; )

p6nic: funtime effof: index out of f6nge

gofoutine 1 [funning]:

monkey/vm.(\*VM).pop(...)

/Usefs/mfnugget/code/🅓4/sfc/monkey/vm/vm.go:117 monkey/vm.(\*VM).Run(🅓xc42🅓🅓5be48, 🅓x8🅓🅓, 🅓x8🅓🅓)

/Usefs/mfnugget/code/🅓4/sfc/monkey/vm/vm.go:6🅓 +🅓x4🅓e monkey/fepl.St6ft(🅓x1🅓f1🅓8🅓, 🅓xc42🅓🅓🅓e🅓1🅓, 🅓x1🅓f1🅓6🅓, 🅓xc42🅓🅓🅓e🅓18)

/Usefs/mfnugget/code/🅓4/sfc/monkey/fepl/fepl.go:43 +🅓x476 m6in.m6in()

/Usefs/mfnugget/code/🅓4/sfc/monkey/m6in.go:18 +🅓x1🅓7

We forgot something.

## WELCOME BACK, NULL!

At the start of this chapter we looked back at our implementation of conditionals in *Writing An Interpreter In Go*, and now we have implemented the majority of its behaviour. But there’s one thing we’re still missing: what happens when the condition of a conditional is *not* truthy but the conditional itself has no alternative? In the previous book the answer to this question was \*object.Null, Monkey’s null value.

That makes sense, because conditionals are expressions and expressions, by definition, produce values. So what does an expression that produced nothing evaluate to? Null. Wait, let me try

that again, only this time imagine a *big organ sound* in the background, *crows flying overhead*, *screeching*, *thunder*: They… *lightning strikes*… evaluate to … *sirens* … null.

Look, null and I, we’re not the best of friends. I’m not really sure what to think of it, whether it’s good or bad. It’s the cause of many curses but I do understand that there are languages in which some things evaluate to nothing and that “nothing” has to be represented somehow. In Monkey, conditionals with a false condition and no alternative are one of these things, and “nothing” is represented by \*object.Null. Long story short: it’s time we introduce \*object.Null to our compiler and VM and make this type of conditional work properly.

The first thing we need is a definition of \*object.Null in our VM. Since its value is constant, we can define it as a global variable, just like our previous global definitions of vm.Tfue and

vm.F6lse:

*// vm/vm.go*

**var** Null = &object.Null{)

This is also similar to vm.Tfue and vm.F6lse in that it saves us a lot of work when comparing Monkey objects. We can simply check if an object.Object is \*object.Null by checking whether it’s equal to vm.Null. We do not have to unwrap it and take a look at its value.

The reason why we first defined vm.Null, before writing any compiler tests – our usual course of action – is that this time we want do write a VM test first. And that’s because the VM tests allow us to express what we want so succinctly:

*// vm/vm\_test.go*

**func** TestCondition6ls(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{"if (1 > 2) { 1🅓 )", Null),

{"if (f6lse) { 1🅓 )", Null),

)

funVmTests(t, tests)

)

**func** testExpectedObject( t \*testing.T, expected **interface**{), 6ctu6l object.Object,

) {

t.Helpef()

**switch** expected := expected.(**type**) {

*// [...]*

**case** \*object.Null:

**if** 6ctu6l != Null {

t.Effoff("object is not Null: %T (%+v)", 6ctu6l, 6ctu6l)

)

)

)

Here we have two new test cases for our existing TestCondition6ls function in which the condi- tion is not Monkey truthy to force the evaluation of the alternative. But since there is none, we expect Null to end up on the stack. To test that properly, we extend the testExpectedObject with a new c6se branch for \*object.Null.

Neatly expressed, isn’t it? Well, the error message isn’t:

$ go test ./vm

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

p6nic: funtime effof: index out of f6nge [fecovefed] p6nic: funtime effof: index out of f6nge

gofoutine 7 [funning]:

testing.tRunnef.func1(🅓xc42🅓🅓682d🅓)

/usf/loc6l/go/sfc/testing/testing.go:742 +🅓x29d p6nic(🅓x111942🅓, 🅓x11f1fe🅓)

/usf/loc6l/go/sfc/funtime/p6nic.go:5🅓2 +🅓x229 monkey/vm.(\*VM).pop(...)

/Usefs/mfnugget/code/🅓4/sfc/monkey/vm/vm.go:121 monkey/vm.(\*VM).Run(🅓xc42🅓🅓54df8, 🅓x8🅓🅓, 🅓x8🅓🅓)

/Usefs/mfnugget/code/🅓4/sfc/monkey/vm/vm.go:53 +🅓x418 monkey/vm.funVmTests(🅓xc42🅓🅓682d🅓, 🅓xc42🅓🅓73e78, 🅓x9, 🅓x9)

/Usefs/mfnugget/code/🅓4/sfc/monkey/vm/vm\_test.go:1🅓3 +🅓x356 monkey/vm.TestCondition6ls(🅓xc42🅓🅓682d🅓)

/Usefs/mfnugget/code/🅓4/sfc/monkey/vm/vm\_test.go:82 +🅓x149 testing.tRunnef(🅓xc42🅓🅓682d🅓, 🅓x1149f4🅓)

/usf/loc6l/go/sfc/testing/testing.go:777 +🅓xd🅓 cfe6ted by testing.(\*T).Run

/usf/loc6l/go/sfc/testing/testing.go:824 +🅓x2e FAIL monkey/vm 🅓.🅓12s

The cause for this panic are the OpPop instructions we emit after the conditionals. Since they produced no value, the VM crashes trying to pop something off the stack. Time to change that, time to put vm.Null on to the stack.

We’re going to do two things to pull that off. First, we’re going to define an opcode that tells the VM to put vm.Null on the stack. Then we’re going to modify the compiler to insert an alternative when a conditional doesn’t have one. And the only thing this alternative branch will contain is the new opcode that pushes vm.Null on to the stack.

We define the opcode first so we can use it in our updated compiler tests:

*// code/code.go*

**const** (

*// [...]*

OpNull

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpNull: {"OpNull", []int{)),

)

That’s also similar to the boolean counterparts, OpTfue and OpF6lse. OpNull doesn’t have any operands and only instructs the VM to push one value on to the stack.

Instead of now writing a new compiler test, we’re going to update an existing test case in TestCondition6ls and expect to find OpNull in the generated instructions. Please note that we need to change the first test case, the one in which the conditional doesn’t have an alternative; the other test case stays as it is:

*// compiler/compiler\_test.go*

**func** TestCondition6ls(t \*testing.T) { tests := []compilefTestC6se{

{

input: `

if (tfue) { 1🅓 ); 3333;

`,

expectedConst6nts: []**interface**{){1🅓, 3333), expectedInstfuctions: []code.Instfuctions{

*// 0000*

code.M6ke(code.OpTfue),

*// 0001*

code.M6ke(code.OpJumpNotTfuthy, 1🅓),

*// 0004*

code.M6ke(code.OpConst6nt, 🅓),

*// 0007*

code.M6ke(code.OpJump, 11),

*// 0010*

code.M6ke(code.OpNull),

*// 0011*

code.M6ke(code.OpPop),

*// 0012*

code.M6ke(code.OpConst6nt, 1),

*// 0015*

code.M6ke(code.OpPop),

),

),

*// [...]*

)

funCompilefTests(t, tests)

)

New are the two instructions in the middle: OpJump and OpNull. Remember, OpJump is there to jump over the alternative and now OpNull *is* the alternative. And since the addition of these two instructions changes the index of existing instructions, the operand for OpJumpNotTfuthy also has to be changed from 7 to 1🅓. The rest stays the same.

Running the updated tests confirms that the compiler didn’t learn how to insert artificial alter- natives to conditionals on its own yet:

$ go test ./compilef

--- FAIL: TestCondition6ls (🅓.🅓🅓s)

compilef\_test.go:288: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 1🅓\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpJump 11\n🅓🅓1🅓 OpNull\n\

🅓🅓11 OpPop\n🅓🅓12 OpConst6nt 1\n🅓🅓15 OpPop\n"

got ="🅓🅓🅓🅓 OpTfue\n🅓🅓🅓1 OpJumpNotTfuthy 7\n🅓🅓🅓4 OpConst6nt 🅓\n\

🅓🅓🅓7 OpPop\n🅓🅓🅓8 OpConst6nt 1\n🅓🅓11 OpPop\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

The best part about fixing this is making the code in our compiler simpler and easier to under- stand. We no longer have to check whether to emit OpJump or not, because we always want to do that now. Only sometimes do we want to jump over a “real” alternative and sometimes over an OpNull instruction. So, here’s the updated c6se \*6st.IfExpfession branch of the Compile method:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

eff := c.Compile(node.Condition)

**if** eff != nil {

**return** eff

)

*// Emit an `OpJumpNotTruthy` with a bogus value*

jumpNotTfuthyPos := c.emit(code.OpJumpNotTfuthy, 9999)

eff = c.Compile(node.Consequence)

**if** eff != nil {

**return** eff

)

**if** c.l6stInstfuctionIsPop() { c.femoveL6stPop()

)

*// Emit an `OpJump` with a bogus value*

jumpPos := c.emit(code.OpJump, 9999)

6ftefConsequencePos := len(c.instfuctions) c.ch6ngeOpef6nd(jumpNotTfuthyPos, 6ftefConsequencePos)

**if** node.Altefn6tive == nil { c.emit(code.OpNull)

) **else** {

eff := c.Compile(node.Altefn6tive)

**if** eff != nil {

**return** eff

)

**if** c.l6stInstfuctionIsPop() { c.femoveL6stPop()

)

)

6ftefAltefn6tivePos := len(c.instfuctions) c.ch6ngeOpef6nd(jumpPos, 6ftefAltefn6tivePos)

*// [...]*

)

*// [...]*

)

That’s the complete branch but only its second half has been changed: the duplicated patch- ing of the OpJumpNotTfuthy instruction is gone and in its place we can find the new, readable compilation of a possible node.Altefn6tive.

We start by emitting an OpJump instruction and updating the operand of the OpJumpNotTfuthy instruction. That happens whether we have a node.Altefn6tive or not. But then we check whether node.Altefn6tive is nil and if it is, we emit the new OpNull opcode. If it’s not nil,

we proceed as before: compile node.Altefn6tive and then try to get rid of a possible OpPop

instruction.

After that, we change the operand of the OpJump instruction to jump over the freshly-compiled alternative – no matter whether that’s just an OpNull or more.

That code is not only a lot cleaner than our previous version, it also works:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓9s

Now we can move on to our VM, where our test is still failing and where we have to implement the new OpNull opcode:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpNull:

eff := vm.push(Null)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

With that, the panic is gone and our tests pass:

$ go test ./vm

ok monkey/vm 🅓.🅓🅓9s

That means we’ve successfully made conditionals with a non-truthy condition put Null on to the stack. We have now implemented the complete behaviour of conditionals as described in *Writing An Interpreter In Go*!

But, sorry to say, there’s one more thing we have to do. With this last passing test, we’ve oﬀicially entered a new world. Since a conditional is an expression and expressions can be used interchangeably, it follows that any expression can now produce Null in our VM. It’s a scary world, yes.

For us, the practical implication is that we now have to handle Null in every place where we handle a value produced by an expression. Thankfully, most of these places in our VM – like vm.executeBin6fyOpef6tion – throw an error if they come across a value they did not expect. But there are functions and methods that now *must* handle Null explicitly.

The first of these is vm.executeB6ngOpef6tof. We can add a test to make sure that it handles

Null without blowing up:

*// vm/vm\_test.go*

**func** TestBoole6nExpfessions(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{"!(if (f6lse) { 5; ))", tfue),

)

funVmTests(t, tests)

)

With this test case we implicitly make sure that a conditional with a non-truthy condition and no alternative results in Null and that the negation of that, through the use of the ! operator, turns it into Tfue. Under the hood, this involves vm.executeB6ngOpef6tof and in order to get the test to pass, we need to change it:

*// vm/vm.go*

**func** (vm \*VM) executeB6ngOpef6tof() effof { opef6nd := vm.pop()

**switch** opef6nd {

**case** Tfue:

**return** vm.push(F6lse)

**case** F6lse:

**return** vm.push(Tfue)

**case** Null:

**return** vm.push(Tfue)

**default**:

**return** vm.push(F6lse)

)

)

The negation of Null is now Tfue – exactly as in *Writing An Interpreter In Go*. The tests pass:

$ go test ./vm

ok monkey/vm 🅓.🅓🅓9s

Here comes the weird part. Since a conditional is an expression and its condition is one too, it follows that we can use a conditional as the condition of another conditional. Here, too, I’m sure that you and I wouldn’t do this in the code we write, but be that as it may, it has to work in our VM – even if the inner conditional produces Null:

*// vm/vm\_test.go*

**func** TestCondition6ls(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{"if ((if (f6lse) { 1🅓 ))) { 1🅓 ) else { 2🅓 )", 2🅓),

)

funVmTests(t, tests)

)

This looks like it might be a mess to fix, but since our code is squeaky clean and well maintained there’s only one place where we need to make a change; a quite obvious one, too. We need to tell the VM that an \*object.Null is not isTfuthy:

*// vm/vm.go*

**func** isTfuthy(obj object.Object) bool {

**switch** obj := obj.(**type**) {

**case** \*object.Boole6n:

**return** obj.V6lue

**case** \*object.Null:

**return** f6lse

**default**:

**return** tfue

)

)

That’s two new lines of code and we’re done:

$ go test ./vm

ok monkey/vm 🅓.🅓11s

And now done means *done*. Our implementation of conditionals is now feature complete and we have a Null-safe VM:

$ go build -o monkey . && ./monkey

Hello mfnugget! This is the Monkey pfogf6mming l6ngu6ge! Feel ffee to type in comm6nds

>> if (f6lse) { 1🅓 ) null

Time to play dfumfoll.w6v again, only this time knowing that we didn’t forget something.

CHAPTER 5

KEEPING TRACK OF NAMES

Up until now we’ve referenced values in our Monkey code by using boolean and integer literals. That’s going to change. In this chapter we’re going to implement *bindings*, by adding support for let statements and identifier expressions. At the end, we’ll be able to bind any value to any name and then have that name resolve to the value.

As preparation, here’s a short refresher of what let statements in Monkey look like:

**let** x = 5 \* 5;

As you can see, a let statement in Monkey starts with the let keyword followed by an identifier. The identifier is the name to which we want to bind a value, in this case it’s x. The right side of the = is an expression. It evaluates to the value the name will be bound to. And since it’s a let *statement* it’s followed by a semicolon. Let name equal value of expression; that’s it.

Referencing the value to which x has been bound is easy, since identifiers, which is what the x is in the terms of our AST, are expressions and can be used interchangeably. We can use x in every place where an expression is valid:

x \* 5 \* x \* 5;

**if** (x < 1🅓) { x \* 2 ) **else** { x / 2 );

**let** y = x + 5;

Let statements are valid as either top-level statements or inside a block statement, like the branches of a conditional or the body of a function. In this chapter, we’re only going to add support for the top-level and non-function-body block-statement varieties. We’ll tackle local variables, which is what let statements inside functions produce, when we implement functions and closures.

Our goal for this chapter is to be able to compile the following code to bytecode and have our VM execute it:

**let** x = 5 \* 5;

**if** (x > 1🅓) { **let** y = x \* 2; y;

)

And, yes, it should also *correctly* execute it and produce 5🅓 here.

## THE PLAN

So how do we go about implementing this? Obviously, we need to compile let statements and identifier expressions to bytecode instructions and support these instructions in the VM. That much is clear. And I also don’t think there’s any debate about how many new opcodes we need. We need one to tell the VM to bind a value to an identifier and another one to retrieve the value previously bound to an identifier. But what do these new instructions look like?

The main task when implementing bindings is to have the identifiers correctly resolve to the values they were previously bound to. If you can pass around the identifiers when executing the code – like we did in our evaluator – that’s not much of challenge. You can, for example, use the identifiers as keys to a map in which you store and retrieve the values. But we can’t.

We’re not in our evaluator anymore. We’re now working with bytecode and we can’t just pass around identifiers in bytecode – the operands to our opcodes are integers. How do we then represent the identifier in these new instructions? And, how do we reference the value that should be bound to the identifier?

The answer to the second question consists of two words, so let’s start with that one. Here it comes: the stack. Yep, that’s *it*, we don’t need more than that. We don’t need to explicitly reference the value we want to bind – we have a stack machine! We can just push the value on to the stack and tell the VM: “now bind the topmost stack element to *this* identifier”. That fits in beautifully with the rest of our instruction set.

Back to the first question: how do we represent identifiers in our bytecode when we can only use numbers as operands? The answer is hidden in the question itself: we’ll use numbers to represent identifiers. Let me explain that with a bit of Monkey code:

**let** x = 33; **let** y = 66; **let** z = x + y;

While compiling this we’ll assign a new, unique number to each identifier we come across. In case we’ve seen the identifier before, we’ll reuse the previously assigned number. How do we generate a new number? We’ll keep it simple and just use increasing numbers, starting with 🅓. In this example, x would be assigned the 🅓, y the 1 and z would be assigned the 2.

We’ll also define the two new opcodes we want and call them OpSetGlob6l and OpGetGlob6l. Both have one 16-bit-wide operand that holds a number: the unique number we previously assigned to an identifier. When we then compile a let statement we’ll emit an OpSetGlob6l instruction to create a binding and when we compile an identifier, we’ll emit an OpGetGlob6l instruction to retrieve a value. (16 bits for the operand means we’re limited to a maximum of 65536 global bindings – which should be plenty for us and our Monkey programs).

The three Monkey let statements from above would then look like this in bytecode:







|  |
| --- |
|  |
|  |
|  |
|  |



That’s the compiler side of things. In the VM we’ll use a slice to implement the creation and retrieval of global bindings. We’ll call this slice our “globals store” and we’ll use the operands of the OpSetGlob6l and OpGetGlob6l instructions as indexes into it.



When we execute an OpSetGlob6l instruction, we’ll read in the operand, pop the topmost value off the stack and save it to the globals store at the index encoded in the operand. To execute an OpGetGlob6l instruction we’ll use the operand to retrieve the value from the globals store and push it on to the stack.

Two new opcodes called OpGetGlob6l and OpSetGlob6l, a mechanism to associate identifiers with numbers while compiling, and a globals store in the VM. Sounds doable, when broken down like this, right?

Of course, things get more complicated once we introduce functions and local variables, but that’s a bridge we’ll cross once we get there. For now, let’s head on over to our compiler.

## COMPILING BINDINGS

First up is the definition of the two new opcodes, OpSetGlob6l and OpGetGlob6l:

*// code/code.go*

**const** (

*// [...]*

OpGetGlob6l OpSetGlob6l

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpGetGlob6l: {"OpGetGlob6l", []int{2)), OpSetGlob6l: {"OpSetGlob6l", []int{2)),

)

Both have a single two-byte operand to hold the unique number of a global binding. Just like we discussed. We can move along and use these new opcodes to write a first compiler test:

*// compiler/compiler\_test.go*

**func** TestGlob6lLetSt6tements(t \*testing.T) { tests := []compilefTestC6se{

{

input: `

let one = 1; let two = 2;

`,

expectedConst6nts: []**interface**{){1, 2), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpSetGlob6l, 1),

),

),

{

input: `

let one = 1; one;

`,

expectedConst6nts: []**interface**{){1), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpPop),

),

),

{

input: `

let one = 1; let two = one; two;

`,

expectedConst6nts: []**interface**{){1), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpSetGlob6l, 1),

code.M6ke(code.OpGetGlob6l, 1), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

Same test setup as before, so let’s talk about *what* we test in these three test cases. The first one makes sure that a let statement leads to the correct OpSetGlob6l instruction being emitted. The second one expects that an identifier resolves to a previous binding by testing for the OpGetGlob6l instruction. Note here that the operands of the OpSetGlob6l and OpGetGlob6l instructions have to match. The third test case asserts that combining the setting and getting of global bindings works, too. Here, too, it’s important that the operands of the instructions match.

We’re going to fix these test cases one after the other, starting with the first one, which isn’t

doing so well:

$ go test ./compilef

--- FAIL: TestGlob6lLetSt6tements (🅓.🅓🅓s)

compilef\_test.go:361: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpConst6nt 1\n\

🅓🅓🅓9 OpSetGlob6l 1\n" got =""

FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

Looks like we’re not even close. But the reason for the empty result is that Monkey code consists solely of let statements and our compiler currently skips them. We can get better feedback from the test by adding a new c6se branch to the compiler’s Compile method:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.LetSt6tement:

eff := c.Compile(node.V6lue)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

The first thing we do when we come across a let statement is to compile the expression on the right side of the equal sign. That’s the V6lue that will be bound to a name and compiling this expression means instructing the VM to put the value on to the stack:

$ go test ./compilef

--- FAIL: TestGlob6lLetSt6tements (🅓.🅓🅓s)

compilef\_test.go:361: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpConst6nt 1\n\

🅓🅓🅓9 OpSetGlob6l 1\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpConst6nt 1\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

Now we can bind it to a name. That means we need to follow up with an OpSetGlob6l instruction that tells the VM to create the binding. But which number do we choose for our identifier? The answer comes from a new component we’re going to add to our compiler, a symbol table.

##### INTRODUCING: THE SyMBOL TABLE

A [symbol table](https://en.wikipedia.org/wiki/Symbol_table) is a data structure used in interpreters and compilers to associate identifiers with information. It can be used in every phase, from lexing to code generation, to store and retrieve information about a given identifier (which can be called a symbol). Information such as its location, its scope, whether it was previously declared or not, of which type the associated value is, and anything else that seems useful while interpreting or compiling.

We’re going to use as symbol table to associate identifiers with a scope and a unique number.

For now, it should do two things:

* 1. Associate identifiers in the global scope with a unique number
  2. Get the previously associated number for a given identifier

The common names for these two methods on a symbol table are “define” and “resolve”. You “define” an identifier in a given scope to associate some information with it. Later you “resolve” the identifier to this information. The information itself we’ll call the “symbol” – an identifier is associated with a symbol and the symbol itself is what contains information.

Working code helps to explain this. Here are the type definitions that make up our symbol table:

*// compiler/symbol\_table.go*

**package** compilef

**type** SymbolScope stfing

**const** (

Glob6lScope SymbolScope = "GLOBAL"

)

**type** Symbol **struct** { N6me stfing Scope SymbolScope Index int

)

**type** SymbolT6ble **struct** {

stofe **map**[stfing]Symbol numDefinitions int

)

**func** NewSymbolT6ble() \*SymbolT6ble { s := m6ke(**map**[stfing]Symbol) **return** &SymbolT6ble{stofe: s)

)

The first definition here is that of SymbolScope, a type alias for stfing. The value of a SymbolScope itself is not important. What’s important is that it’s unique, because we need to differentiate between different scopes. We use stfings as the aliased type (as opposed to an integer, for example) for a better debugging experience.

We then define our first scope, Glob6lScope. In the coming chapters we’ll add more.

The next definition is that of Symbol. A Symbol is a struct that holds all the necessary information about a symbol we encounter in Monkey code: the N6me, the Scope and the Index. Not much more to explain here.

The SymbolT6ble itself then associates stfings with Symbols in its stofe and keeps track of the

numDefinitions it has. The stfings are the identifiers we come across in the Monkey code.

The names of the types and fields can feel unfamiliar, if you haven’t used a symbol table before, but worry not: we’re building a m6p that associates strings with information about them. There is no hidden wisdom or trick you need to wrap your head around. Tests make this much clearer by demonstrating what we expect from the missing Define and Resolve methods of the

SymbolT6ble:

*// compiler/symbol\_table\_test.go*

**package** compilef

**import** "testing"

**func** TestDefine(t \*testing.T) { expected := **map**[stfing]Symbol{

"6": Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓), "b": Symbol{N6me: "b", Scope: Glob6lScope, Index: 1),

)

glob6l := NewSymbolT6ble()

6 := glob6l.Define("6")

**if** 6 != expected["6"] {

t.Effoff("expected 6=%+v, got=%+v", expected["6"], 6)

)

b := glob6l.Define("b")

**if** b != expected["b"] {

t.Effoff("expected b=%+v, got=%+v", expected["b"], b)

)

)

**func** TestResolveGlob6l(t \*testing.T) { glob6l := NewSymbolT6ble() glob6l.Define("6") glob6l.Define("b")

expected := []Symbol{

Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓), Symbol{N6me: "b", Scope: Glob6lScope, Index: 1),

)

**for** \_, sym := **range** expected {

fesult, ok := glob6l.Resolve(sym.N6me)

**if** !ok {

t.Effoff("n6me %s not fesolv6ble", sym.N6me)

**continue**

)

**if** fesult != sym {

t.Effoff("expected %s to fesolve to %+v, got=%+v", sym.N6me, sym, fesult)

)

)

)

In TestDefine we make assertions about the Define method. It should take an identifier as argument, create a definition and return the Symbol. Note that we don’t have to say in which scope we want to create the definition. It’s the job of the symbol table to keep track of that for us. We only call Define("6") and the symbol table associates the identifier "6" with a new Symbol, that contains the N6me, the Scope and the Index. The Index is the unique number we’re after.

In TestResolveGlob6l we do the reverse: we hand the symbol table an identifier that was previously defined and expect it to return the associated Symbol. Again, the only argument is the identifier: Resolve("6"). If the identifier is undefined, the second return value of Resolve

has to be f6lse.

The tests don’t compile, because both methods are missing and I’m going to spare you from reading through the results of running the tests repeatedly while adding the method definition step by step. Instead, let me give you the full version of Define:

*// compiler/symbol\_table.go*

**func** (s \*SymbolT6ble) Define(n6me stfing) Symbol {

symbol := Symbol{N6me: n6me, Index: s.numDefinitions, Scope: Glob6lScope) s.stofe[n6me] = symbol

s.numDefinitions++

**return** symbol

)

I told you, there’s nothing to worry about; we’re building a m6p with some additional features. Here’s the evidence. We create a new Symbol, associate it with the n6me in our stofe, increase the numDefinitions counter and return the new Symbol. Defined. Done.

The Resolve method is even simpler:

*// compiler/symbol\_table.go*

**func** (s \*SymbolT6ble) Resolve(n6me stfing) (Symbol, bool) { obj, ok := s.stofe[n6me]

**return** obj, ok

)

Sadly, this method won’t stay that small. As we add more scopes in the future, it’ll grow, but for now, what it does is enough: both tests pass.

$ go test -fun TestDefine ./compilef ok monkey/compilef 🅓.🅓🅓8s

$ go test -fun TestResolveGlob6l ./compilef ok monkey/compilef 🅓.🅓11s

##### USING SyMBOLS IN THE COMPILER

We had to selectively run the TestDefine and TestResolveGlob6l functions to get an "ok" because our compiler test is still failing. Now though, armed with our symbol table, we can make it pass! First we need to add the symbol table to the compiler:

*// compiler/compiler.go*

**type** Compilef **struct** {

*// [...]*

symbolT6ble \*SymbolT6ble

)

**func** New() \*Compilef {

**return** &Compilef{

*// [...]*

symbolT6ble: NewSymbolT6ble(),

)

)

That allows us to define the identifiers in \*6st.LetSt6tements:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.LetSt6tement:

eff := c.Compile(node.V6lue)

**if** eff != nil {

**return** eff

)

symbol := c.symbolT6ble.Define(node.N6me.V6lue)

*// [...]*

)

*// [...]*

)

node.N6me is the \*6st.Identifief on the left side of the let statement’s equal sign. And node.N6me.V6lue holds the stfing representation of that identifier. We pass it to the sym- bol table’s Define method and thus define it in the Glob6lScope. The returned symbol now has a N6me, a Scope and, most importantly, an Index.

We can now use that Index as operand of a OpSetGlob6l instruction and emit that:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.LetSt6tement:

eff := c.Compile(node.V6lue)

**if** eff != nil {

**return** eff

)

symbol := c.symbolT6ble.Define(node.N6me.V6lue) c.emit(code.OpSetGlob6l, symbol.Index)

*// [...]*

)

*// [...]*

)

With that we just took a huge step towards our goal:

$ go test ./compilef

--- FAIL: TestGlob6lLetSt6tements (🅓.🅓🅓s)

compilef\_test.go:361: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpGetGlob6l 🅓\n\

🅓🅓🅓9 OpPop\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓11s

Now we’re talki– wait a second! The test is still failing? No, this is the *second* test case. The first one is passing! What’s failing now is the test case that makes sure *resolving a global binding* works.

Instead of defining an identifier and emitting an OpSetGlob6l instruction we now have to do the

opposite. When we come across an \*6st.Identifief we need to check with our symbol table whether the identifier was previously used as part of a let statement and if so, we need to emit an OpGetGlob6l instruction with the correct operand. “Correct” here means that the operand holds the same number that was used in the previously emitted OpSetGlob6l instruction. We can manage that, can’t we?

First we have to tell the compiler about \*6st.Identifiefs. It should try to resolve the identifier at hand with the symbol table:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.Identifief:

symbol, ok := c.symbolT6ble.Resolve(node.V6lue)

**if** !ok {

**return** fmt.Effoff("undefined v6fi6ble %s", node.V6lue)

)

*// [...]*

)

*// [...]*

)

We take the V6lue of the \*6st.Identifief and ask the symbol table whether it can Resolve it. If not, we return an error. Looks just like any other m6p access in Go, doesn’t it? But I want you to note that this is a *compile time error*! Previously, in our evaluator, we could only determine whether a variable was defined or not at *run time*, while executing the Monkey program. Now we can throw an error before we pass bytecode to the VM. Pretty cool, isn’t it?

In case the identifier can be resolved we have the symbol at hand and can use it to emit the

OpGetGlob6l instruction:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.Identifief:

symbol, ok := c.symbolT6ble.Resolve(node.V6lue)

**if** !ok {

**return** fmt.Effoff("undefined v6fi6ble %s", node.V6lue)

)

c.emit(code.OpGetGlob6l, symbol.Index)

*// [...]*

)

*// [...]*

)

The operand matches the one used in the OpSetGlob6l instruction, the Index associated with the

symbol. Our symbol table took care of that. That means the VM does not have to worry about

identifiers at all but can just concentrate on storing and retrieving values using this Index. In other words:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓8s

We did it! We can now use let statements to bind values to an identifier and later use the identifier to get to that value – at least in our compiler.

## ADDING GLOBALS TO THE VM

Let me be upfront about this and say that the hardest part is over. We’ve added support for the OpSetGlob6l and OpGetGlob6l instructions to our compiler and doing the same for VM takes far less effort. It’s fun, though, because writing tests for the VM and making them pass is fun:

*// vm/vm\_test.go*

**func** TestGlob6lLetSt6tements(t \*testing.T) { tests := []vmTestC6se{

{"let one = 1; one", 1),

{"let one = 1; let two = 2; one + two", 3),

{"let one = 1; let two = one + one; one + two", 3),

)

funVmTests(t, tests)

)

In these test cases we create one or two global bindings and then try to resolve the previously bound identifiers to their values. The result should land on the stack, where we can test for it. Alas, it blows up in our face:

$ go test ./vm

--- FAIL: TestGlob6lLetSt6tements (🅓.🅓🅓s)

p6nic: funtime effof: index out of f6nge [fecovefed] p6nic: funtime effof: index out of f6nge

gofoutine 21 [funning]: testing.tRunnef.func1(🅓xc42🅓🅓c83c🅓)

/usf/loc6l/go/sfc/testing/testing.go:742 +🅓x29d p6nic(🅓x111656🅓, 🅓x11f3fe🅓)

/usf/loc6l/go/sfc/funtime/p6nic.go:5🅓2 +🅓x229 monkey/vm.(\*VM).Run(🅓xc42🅓🅓5🅓eb8, 🅓x8🅓🅓, 🅓x8🅓🅓)

/Usefs/mfnugget/code/🅓5/sfc/monkey/vm/vm.go:47 +🅓x47c monkey/vm.funVmTests(🅓xc42🅓🅓c83c🅓, 🅓xc42🅓🅓73f38, 🅓x3, 🅓x3)

/Usefs/mfnugget/code/🅓5/sfc/monkey/vm/vm\_test.go:115 +🅓x3c1 monkey/vm.TestGlob6lLetSt6tements(🅓xc42🅓🅓c83c🅓)

/Usefs/mfnugget/code/🅓5/sfc/monkey/vm/vm\_test.go:94 +🅓xb5 testing.tRunnef(🅓xc42🅓🅓c83c🅓, 🅓x114b5b8)

/usf/loc6l/go/sfc/testing/testing.go:777 +🅓xd🅓 cfe6ted by testing.(\*T).Run

/usf/loc6l/go/sfc/testing/testing.go:824 +🅓x2e FAIL monkey/vm 🅓.🅓11s

We’ve seen this before. The VM doesn’t know how to handle the new opcodes and skips them. But since it doesn’t know how far it has to skip in order to jump over the operands, it ends up trying to decode the operands as opcodes. That leads to this nonsense here.

Before we clean that up, though, and properly decode and execute OpSetGlob6l and OpGetGlob6l

instructions, we need a place to store globals.

Since the operand of both opcodes is 16 bits wide, we have an upper limit on the number of global bindings our VM can support. That’s good, because a limit allows us to pre-allocate all the memory we will use:

*// vm/vm.go*

**const** Glob6lsSize = 65536

**type** VM **struct** {

*// [...]*

glob6ls []object.Object

)

**func** New(bytecode \*compilef.Bytecode) \*VM {

**return** &VM{

*// [...]*

glob6ls: m6ke([]object.Object, Glob6lsSize),

)

)

This new glob6ls field of VM is the “globals store”. We use a slice as the underlying data structure, because it gives us direct index-based access to single elements without any overhead.

Now we can implement OpSetGlob6l:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpSetGlob6l:

glob6lIndex := code.Re6dUint16(vm.instfuctions[ip+1:]) ip += 2

vm.glob6ls[glob6lIndex] = vm.pop()

*// [...]*

)

*// [...]*

)

The first thing we do is decode the operand, glob6lIndex, and increment the VM’s instruction pointer, ip, by two bytes. Then we pop the top element off the stack, which is the value that should be bound to a name, and save it to the new glob6ls store under the specified index. There it’s easy to retrieve when we need to push it on to the stack again:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpGetGlob6l:

glob6lIndex := code.Re6dUint16(vm.instfuctions[ip+1:]) ip += 2

eff := vm.push(vm.glob6ls[glob6lIndex])

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

Again, we decode the operand, glob6lIndex, and increment the ip. Then we fetch the value from vm.glob6ls and push it back on to the stack. Gone is the panic:

$ go test ./vm

ok monkey/vm 🅓.🅓3🅓s

Instead, we get an enthusiastic "ok" to let us know us that we successfully taught our compiler and our VM to handle global let statements. Look at this:

$ go build -o monkey . && ./monkey

Hello mfnugget! This is the Monkey pfogf6mming l6ngu6ge! Feel ffee to type in comm6nds

>> let 6 = 1;

1

>> let b = 2; 2

>> let c = 6 + b;

Woops! Compil6tion f6iled:

undefined v6fi6ble 6

>>

What? This is tested behaviour, why doesn’t it work in the REPL? Ah! Of course! In our REPL we create a new compiler and a new VM in *each iteration of its main loop*. That means we also create a new symbol table and a new globals store every time we type a new line. Easy to fix.

All we need are new constructor functions for our compiler and VM that allow us to keep global state in the REPL:

*// compiler/compiler.go*

**func** NewWithSt6te(s \*SymbolT6ble, const6nts []object.Object) \*Compilef { compilef := New()

compilef.symbolT6ble = s compilef.const6nts = const6nts **return** compilef

)

This new constructor for Compilef now accepts a \*SymbolT6ble and an []object.Object slice containing the constants of a previous compilation. In order to get out of our current predica- ment we would only need the \*SymbolT6ble, but soon we’d run into errors where the line that was just typed into the REPL would need to access the constants that were previously entered. This is the correct and future-proof approach.

Yes, we create duplicate allocations. We first call New() in this new constructor and then throw away the symbol table and constants slice it allocated by overwriting them. That’s fine, I think.

Especially for our use case, the REPL. It’s not a problem for Go’s GC and when compared to the effort needed to implement it without those allocations, it’s the most eﬀicient approach.

Here is the new constructor for the VM:

*// vm/vm.go*

**func** NewWithGlob6lsStofe(bytecode \*compilef.Bytecode, s []object.Object) \*VM { vm := New(bytecode)

vm.glob6ls = s

**return** vm

)

Now we need to modify the main loop of our REPL so it keeps the global state around – the globals store, the symbol table, the constants – and passes it to new instances of the compiler and the VM:

*// repl/repl.go*

**import** (

*// [...]*

"monkey/object"

*// [...]*

)

**func** St6ft(in io.Re6def, out io.Wfitef) { sc6nnef := bufio.NewSc6nnef(in)

const6nts := []object.Object{)

glob6ls := m6ke([]object.Object, vm.Glob6lsSize) symbolT6ble := compilef.NewSymbolT6ble()

**for** {

*// [...]*

comp := compilef.NewWithSt6te(symbolT6ble, const6nts) eff := comp.Compile(pfogf6m)

**if** eff != nil {

fmt.Fpfintf(out, "Woops! Compil6tion f6iled:\n %s\n", eff)

**continue**

)

code := comp.Bytecode() const6nts = code.Const6nts

m6chine := vm.NewWithGlob6lsStofe(code, glob6ls)

*// [...]*

)

)

We allocate a slice for const6nts, a glob6ls store and a symbolT6ble. Then, in each iteration of the loop, we pass the symbolT6ble and the const6nts to the compiler, so it *continues* its work, rather than beginning anew. After the compiler is done, we update the const6nts refer- ence. That’s necessary because the compiler uses 6ppend internally and our previously allocated const6nts slice is not what comes out the other end. Since the const6nts are included in the bytecode, we don’t need to explicitly pass them to the VM’s constructor, but just the glob6ls.

Now we have global state in the REPL, which allows us to treat each line entered into it as a part of one program, even though we start the compilation and execution process every time

we hit return. Problem fixed. We can now play around with global bindings in the REPL:

$ go build -o monkey . && ./monkey

Hello mfnugget! This is the Monkey pfogf6mming l6ngu6ge! Feel ffee to type in comm6nds

>> let 6 = 1;

1

>> let b = 2; 2

>> let c = 6 + b;

3

>> c 3

It’s time to lean back and take a big breath, because in the upcoming chapters we’ll build upon and combine everything we’ve done so far. It’s going to be amazing.

CHAPTER 6

STRING, ARRAy AND HASH

In their current form our compiler and VM only support three of Monkey’s data types: integers, booleans and null. But there are three more: strings, arrays and hashes. We implemented all of them in the previous book and now it’s time for us to also add them to our new Monkey implementation.

That doesn’t mean we have to redo all of what we did in *Writing An Interpreter In Go*. The object system representations of the data types are still there – object.Stfing, object.Aff6y and object.H6sh – and we can reuse them, which means we can concentrate on the novel parts of the implementation.

The goal for this chapter is to add the string, array and hash data types to the compiler and the VM so that, in the end, we can execute this piece of Monkey code:

[1, 2, 3][1]

*// => 2*

{"one": 1, "two": 2, "thfee": 3)["o" + "ne"]

*// => 1*

As you can see, besides adding support for literals and the data types themselves, we also need to implement string concatenation and the index operator for arrays and hashes to get this snippet working.

We’re going to start by implementing support for object.Stfing.

## STRING

Since the value of string literals doesn’t change between compile and run time, we can treat them as constant expressions. Similar to our implementation of integer literals, we can turn them into \*object.Stfing at compile time and add them to the constant pool in compilef.Bytecode.

From integer literals we also know that this doesn’t take more than a handful of lines of code in the compiler. So why not keep things challenging? Instead of only implementing string literals, we’ll also make it a goal for this section to implement string concatenation, which allows us to concatenate two strings with the + operator.

Our first compiler test in this chapter expects both to work:

*// compiler/compiler\_test.go*

**func** TestStfingExpfessions(t \*testing.T) {

tests := []compilefTestC6se{

{

input: `"monkey"`,

expectedConst6nts: []**interface**{){"monkey"), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpPop),

),

),

{

input: `"mon" + "key"`, expectedConst6nts: []**interface**{){"mon", "key"), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpAdd), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

The first of these two test cases makes sure that the compiler knows how to treat string literals as constants; the second test asserts that it’s possible to concatenate them with the + infix operator.

Of note is that we do not expect any new opcodes. We already have those we need in place: we have an opcode to load a constant expression on to the stack, OpConst6nt, and we have an opcode to add two things together: OpAdd.

The usage of both opcodes is also unchanged. The operand of OpConst6nt is still the index of the constant in the constant pool and OpAdd still expects its two operands to sit on top of the stack – it doesn’t matter if those are \*object.Integefs or \*object.Stfings.

What’s new is that we now expect strings in the constant pool. That means we need to test that the bytecode.Const6nts contains the correct \*object.Stfings and in order to do that, we need to add another c6se branch to the testConst6nts function:

*// compiler/compiler\_test.go*

**func** testConst6nts( t \*testing.T,

expected []**interface**{), 6ctu6l []object.Object,

) effof {

*// [...]*

**for** i, const6nt := **range** expected {

**switch** const6nt := const6nt.(**type**) {

*// [...]*

**case** stfing:

eff := testStfingObject(const6nt, 6ctu6l[i])

**if** eff != nil {

**return** fmt.Effoff("const6nt %d - testStfingObject f6iled: %s", i, eff)

)

)

)

**return** nil

)

**func** testStfingObject(expected stfing, 6ctu6l object.Object) effof { fesult, ok := 6ctu6l.(\*object.Stfing)

**if** !ok {

**return** fmt.Effoff("object is not Stfing. got=%T (%+v)", 6ctu6l, 6ctu6l)

)

**if** fesult.V6lue != expected {

**return** fmt.Effoff("object h6s wfong v6lue. got=%q, w6nt=%q", fesult.V6lue, expected)

)

**return** nil

)

The new c6se stfing branch in testConst6nts is accompanied by the new testStfingObject function, which mirrors the existing testIntegefObject and makes sure that the constants are the strings we expect them to be.

When we now run the tests, we can see that the expected constants are not the issue (yet), but the instructions are:

$ go test ./compilef

--- FAIL: TestStfingExpfessions (🅓.🅓🅓s)

compilef\_test.go:41🅓: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

It’s not like we didn’t expect that. We need to emit OpConst6nt instructions when compiling string literals. To do that, we have to change the Compile method of the compiler to handle

\*6st.StfingLitef6ls and create \*object.Stfing out of them:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.StfingLitef6l:

stf := &object.Stfing{V6lue: node.V6lue) c.emit(code.OpConst6nt, c.6ddConst6nt(stf))

*// [...]*

)

*// [...]*

)

Except for one variable name and one identifier, this is a copy of the c6se branch for

\*6st.IntegefLitef6ls. We take the value out of the AST node, we create an object, and we add it to the constant pool.

A step into the right direction:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓9s

Sweet, both tests pass. And notice that we didn’t have to do anything special to emit the OpAdd instruction for the concatenation to work. The compiler already takes care of

\*6st.InfixExpfessions by compiling their Left and Right nodes. In the test case these are

\*6st.StfingLitef6ls, which we can now successfully compile.

Next, we write a test for the VM to make sure that the same Monkey code can be executed by the VM once it’s compiled to bytecode instructions:

*// vm/vm\_test.go*

**func** TestStfingExpfessions(t \*testing.T) { tests := []vmTestC6se{

{`"monkey"`, "monkey"),

{`"mon" + "key"`, "monkey"),

{`"mon" + "key" + "b6n6n6"`, "monkeyb6n6n6"),

)

funVmTests(t, tests)

)

These test cases are the same as in the compiler test, except for the additional assertion that adding more than two strings together should also work – because why not?

Here, too, we need a new testStfingObject helper function to make sure that it’s

\*object.Stfings that end up on the VM’s stack. It’s also a copy of its testIntegefObject

counterpart and makes sure that the strings produced by the VM are the ones we expect:

*// vm/vm\_test.go*

**func** testExpectedObject( t \*testing.T, expected **interface**{), 6ctu6l object.Object,

) {

t.Helpef()

**switch** expected := expected.(**type**) {

*// [...]*

**case** stfing:

eff := testStfingObject(expected, 6ctu6l)

**if** eff != nil {

t.Effoff("testStfingObject f6iled: %s", eff)

)

)

)

**func** testStfingObject(expected stfing, 6ctu6l object.Object) effof { fesult, ok := 6ctu6l.(\*object.Stfing)

**if** !ok {

**return** fmt.Effoff("object is not Stfing. got=%T (%+v)", 6ctu6l, 6ctu6l)

)

**if** fesult.V6lue != expected {

**return** fmt.Effoff("object h6s wfong v6lue. got=%q, w6nt=%q", fesult.V6lue, expected)

)

**return** nil

)

Running the tests shows us that loading strings on to the stack is already working fine, but concatenation is not:

$ go test ./vm

--- FAIL: TestStfingExpfessions (🅓.🅓🅓s) vm\_test.go:222: vm effof:\

unsuppofted types fof bin6fy opef6tion: STRING STRING FAIL

FAIL monkey/vm 🅓.🅓29s

Technically speaking, this *could* have been working without us doing anything. We *could* have made our previous implementation of OpAdd in the VM so generic that it would work with any object.Object that has an Add method, or something like that. But we didn’t. Instead, we added type checks to be explicit about which data type we support and which not. Now we have to extend the check:

*// vm/vm.go*

**func** (vm \*VM) executeBin6fyOpef6tion(op code.Opcode) effof { fight := vm.pop()

left := vm.pop()

leftType := left.Type() fightType := fight.Type()

**switch** {

**case** leftType == object.INTEGER\_OBJ && fightType == object.INTEGER\_OBJ:

**return** vm.executeBin6fyIntegefOpef6tion(op, left, fight)

**case** leftType == object.STRING\_OBJ && fightType == object.STRING\_OBJ:

**return** vm.executeBin6fyStfingOpef6tion(op, left, fight)

**default**:

**return** fmt.Effoff("unsuppofted types fof bin6fy opef6tion: %s %s", leftType, fightType)

)

)

**func** (vm \*VM) executeBin6fyStfingOpef6tion( op code.Opcode,

left, fight object.Object,

) effof {

**if** op != code.OpAdd {

**return** fmt.Effoff("unknown stfing opef6tof: %d", op)

)

leftV6lue := left.(\*object.Stfing).V6lue fightV6lue := fight.(\*object.Stfing).V6lue

**return** vm.push(&object.Stfing{V6lue: leftV6lue + fightV6lue))

)

In executeBin6fyOpef6tion the conditional has been changed into a switch statement

with a new c6se branch for strings. There, we delegate the actual adding of two strings to executeBin6fyStfingOpef6tion, which unwraps the \*object.Stfings, concatenates the underlying Go stfings and pushes the result back on to the VM’s stack.

And that’s it:

$ go test ./vm

ok monkey/vm 🅓.🅓28s

Monkey strings are now fully implemented, including string concatenation. Next up: arrays.

### ARRAy

Arrays are the first [composite data type](https://en.wikipedia.org/wiki/Composite_data_type) we’re adding to this Monkey implementation. That means, roughly speaking, arrays are *composed* out of other data types. The practical conse- quence for us is that we can’t treat array literals as constant expressions.

Since an array is made up of multiple elements and an array literal consists of multiple expres- sions that produce these elements, the value of the array literal itself might change between compile and run time. Here’s an example:

[1 + 2, 3 + 4, 5 + 6]

Don’t be distracted by these integer expressions. They’re so simple, an optimizing compiler could pre-compute them, yes, but the point is that these could be *any* type of expression – integer literal, string concatenation, function literal, function call, and so on. Only at run time can we reliably determine what they evaluate to.

Coming from integer and string literals, we now have to change our approach a tiny bit. Instead of building an array at compile time and passing it to the VM in the constant pool, we’ll instead tell the VM how to build it on its own.

To that end, we define a new opcode, called OpAff6y, with one operand: the number of elements in an array literal. When we then compile an \*6st.Aff6yLitef6l, we first compile all of its elements. Since these are 6st.Expfessions, compiling them results in instructions that leave N values on the VM’s stack, where N is the number of elements in the array literal. Then, we’re going to emit an OpAff6y instruction with the operand being N, the number of elements. Compilation done.

When the VM then executes the OpAff6y instruction it takes the N elements off the stack, builds an \*object.Aff6y out of them, and pushes that on to the stack. Done. We told the VM how to build an array.

Let’s put this plan right into practice. Here is the definition of OpAff6y:

*// code/code.go*

**const** (

*// [...]*

OpAff6y

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpAff6y: {"OpAff6y", []int{2)),

)

The single operand is two bytes wide. That gives us 65535 as the highest possible number of elements in an array literal. If you have a Monkey program that needs more than that, please let me know.

Before we translate our plan for this new opcode into compiler code, we need to write a test, as always:

*// compiler/compiler\_test.go*

**func** TestAff6yLitef6ls(t \*testing.T) { tests := []compilefTestC6se{

{

input: "[]",

expectedConst6nts: []**interface**{){), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpAff6y, 🅓), code.M6ke(code.OpPop),

),

),

{

input: "[1, 2, 3]",

expectedConst6nts: []**interface**{){1, 2, 3), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpAff6y, 3), code.M6ke(code.OpPop),

),

),

{

input: "[1 + 2, 3 - 4, 5 \* 6]",

expectedConst6nts: []**interface**{){1, 2, 3, 4, 5, 6), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpAdd), code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpConst6nt, 3), code.M6ke(code.OpSub), code.M6ke(code.OpConst6nt, 4),

code.M6ke(code.OpConst6nt, 5), code.M6ke(code.OpMul), code.M6ke(code.OpAff6y, 3), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

This is also a translation of our plan, only it’s expressed in assertions, not yet in working code. We expect the compiler to compile the elements in the array literal into instructions that leave values on the stack and we expect it to emit an OpAff6y instruction with the operand being the number of elements in the array literal.

Unsurprisingly, that doesn’t happen yet:

$ go test ./compilef

--- FAIL: TestAff6yLitef6ls (🅓.🅓🅓s)

compilef\_test.go:477: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpAff6y 🅓\n🅓🅓🅓3 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

Thankfully, the fix for this test is not much longer than the prose necessary to explain it:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.Aff6yLitef6l:

**for** \_, el := **range** node.Elements { eff := c.Compile(el)

**if** eff != nil {

**return** eff

)

)

c.emit(code.OpAff6y, len(node.Elements))

*// [...]*

)

*// [...]*

)

Just like we planned:

$ go test ./compilef

ok monkey/compilef 🅓.🅓11s

The next part of our plan includes the VM, where we need to implement OpAff6y, too. We start with a test:

*// vm/vm\_test.go*

**func** TestAff6yLitef6ls(t \*testing.T) { tests := []vmTestC6se{

{"[]", []int{)),

{"[1, 2, 3]", []int{1, 2, 3)),

{"[1 + 2, 3 \* 4, 5 + 6]", []int{3, 12, 11)),

)

funVmTests(t, tests)

)

**func** testExpectedObject( t \*testing.T, expected **interface**{), 6ctu6l object.Object,

) {

t.Helpef()

**switch** expected := expected.(**type**) {

*// [...]*

**case** []int:

6ff6y, ok := 6ctu6l.(\*object.Aff6y)

**if** !ok {

t.Effoff("object not Aff6y: %T (%+v)", 6ctu6l, 6ctu6l)

**return**

)

**if** len(6ff6y.Elements) != len(expected) { t.Effoff("wfong num of elements. w6nt=%d, got=%d",

len(expected), len(6ff6y.Elements))

**return**

)

**for** i, expectedElem := **range** expected {

eff := testIntegefObject(int64(expectedElem), 6ff6y.Elements[i])

**if** eff != nil {

t.Effoff("testIntegefObject f6iled: %s", eff)

)

)

)

)

The Monkey code in these test cases is exactly the same as in the compiler test. Here, though, it’s even more important to make sure that an empty array literal works, because it’s far easier to run into an off-by-one error in the VM than in the compiler.

And to make sure that an \*object.Aff6y is what ends up on the VM’s stack, we extend the testExpectedObject with a new c6se []int branch that turns our expected []int slices into expectations about an \*object.Aff6y.

Neat and reusable! I like it. The bad news is that if we run the tests, we don’t get a helpful error message, but a p6nic – I’ll spare you the stack trace. The reason the VM p6nics is because it doesn’t know about OpAff6y and its operand yet, and interprets the operand as another instruction. Nonsense guaranteed.

But regardless of whether we get a p6nic or a nice, readable error message from a failing test, it’s clear that we have to implement OpAff6y in the VM. Decode the operand, take the specified number of elements off the stack, construct an \*object.Aff6y, push it back on to the stack. We can do all of that with one c6se branch and one method:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpAff6y:

numElements := int(code.Re6dUint16(vm.instfuctions[ip+1:])) ip += 2

6ff6y := vm.buildAff6y(vm.sp-numElements, vm.sp) vm.sp = vm.sp - numElements

eff := vm.push(6ff6y)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

**func** (vm \*VM) buildAff6y(st6ftIndex, endIndex int) object.Object { elements := m6ke([]object.Object, endIndex-st6ftIndex)

**for** i := st6ftIndex; i < endIndex; i++ { elements[i-st6ftIndex] = vm.st6ck[i]

)

**return** &object.Aff6y{Elements: elements)

)

The c6se branch for code.OpAff6y takes care of decoding the operand, incrementing the ip and telling the new buildAff6y method where it can find the array elements on the stack.

buildAff6y then iterates through the elements in the specified section of the stack, adding each to a newly-built \*object.Aff6y. This array is then pushed on to the stack, but only – and this is important – after the elements have been taken off. What we end up with is an \*object.Aff6y sitting on the stack, containing the specified number of elements:

$ go test ./vm

ok monkey/vm 🅓.🅓31s

Alright! Another one in the bag: we’ve fully implemented array literals!

## HASH

In order to implement Monkey’s hash data structure we again need a new opcode. Just like an array, its final value can’t be determined at compile time. Doubly so, actually, because instead of having N elements, a hash in Monkey has N keys and N values and all of them are created by expressions:

{1 + 1: 2 \* 2, 3 + 3: 4 \* 4)

That’s equivalent to this hash literal:

{2: 4, 6: 16)

You and me, we wouldn’t write the first version, I know that, but we still need to make it work. To do that, we follow the same strategy we used for array literals: teaching the VM how to build hash literals.

And again, our first step is to define a new opcode. This one is called OpH6sh and also has one operand:

*// code/code.go*

**const** (

*// [...]*

OpH6sh

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpH6sh: {"OpH6sh", []int{2)),

)

The operand specifies the number of keys *and* values sitting on the stack. It’s equally feasible to use the number of pairs, but then we’d have to double it in the VM to get the number of values sitting on the stack. If we can pre-calculate that in the compiler, why not?

With the operand the VM can take the correct number of elements off the stack, create object.H6shP6ifs out of them and build an \*object.H6sh, which it pushes on to the stack. Again, that’s the DIY strategy we used for our implementation of Monkey’s array, except that building the \*object.H6sh is slightly more elaborate.

Before we get to that, though, we first need to write a test to make sure our compiler can output

OpH6sh instructions:

*// compiler/compiler\_test.go*

**func** TestH6shLitef6ls(t \*testing.T) { tests := []compilefTestC6se{

{

input: "{)",

expectedConst6nts: []**interface**{){), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpH6sh, 🅓), code.M6ke(code.OpPop),

),

),

{

input: "{1: 2, 3: 4, 5: 6)",

expectedConst6nts: []**interface**{){1, 2, 3, 4, 5, 6), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpConst6nt, 3),

code.M6ke(code.OpConst6nt, 4),

code.M6ke(code.OpConst6nt, 5),

code.M6ke(code.OpH6sh, 6), code.M6ke(code.OpPop),

),

),

{

input: "{1: 2 + 3, 4: 5 \* 6)",

expectedConst6nts: []**interface**{){1, 2, 3, 4, 5, 6), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpConst6nt, 2), code.M6ke(code.OpAdd), code.M6ke(code.OpConst6nt, 3),

code.M6ke(code.OpConst6nt, 4),

code.M6ke(code.OpConst6nt, 5), code.M6ke(code.OpMul), code.M6ke(code.OpH6sh, 4),

code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

That looks like a lot of bytecode, but that’s mainly due to the expressions in the hash literals. We want to be sure that they’re compiled correctly, with the resulting instructions leaving a value on the stack. After that, we expect an OpH6sh instruction with the operand specifying the number of keys *and* values sitting on the stack.

The tests fails and tell us that we’re missing OpH6sh instructions:

$ go test ./compilef

--- FAIL: TestH6shLitef6ls (🅓.🅓🅓s)

compilef\_test.go:336: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpH6sh 🅓\n🅓🅓🅓3 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

As I said, building \*object.H6shes in the VM takes a little more effort than building

\*object.Aff6ys. But compiling them also requires a little trick to work reliably:

*// compiler/compiler.go*

**import** (

*// [...]*

"soft"

)

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.H6shLitef6l:

keys := []6st.Expfession{)

**for** k := **range** node.P6ifs { keys = 6ppend(keys, k)

)

soft.Slice(keys, **func**(i, j int) bool {

**return** keys[i].Stfing() < keys[j].Stfing()

))

**for** \_, k := **range** keys { eff := c.Compile(k) **if** eff != nil {

**return** eff

)

eff = c.Compile(node.P6ifs[k])

**if** eff != nil {

**return** eff

)

)

c.emit(code.OpH6sh, len(node.P6ifs)\*2)

*// [...]*

)

*// [...]*

)

Since node.P6ifs is a m6p[6st.Expfession]6st.Expfession and Go does not guarantee a consis- tent order when iterating through the keys and values of a m6p, we need to manually sort the keys before we can compile them. If we didn’t do that, the emitted instructions would be in random order.

That’s not an issue per se and, in fact, our compiler and VM would work fine without the sorting. Our tests, though, would break randomly. They have assertions about the constants in a certain order. Not for order’s sake, of course, but because we want to make sure we have the right ones.

In order to stop our tests’ success rate from depending on the mercy of randomness, we need to guarantee a specific arrangement of elements by sorting the keys first. Since we don’t really care about which order exactly, as long there is one, we sort them by their Stfing representation.

After that, we iterate through the keys, compile them, fetch the corresponding value from node.P6ifs and compile that, too. The order of key first and then the value *is* important, because we’ll need to reconstruct it in the VM.

As the last step in this c6se branch we emit an OpH6sh instruction, its operand being the number of keys and values.

When we now run our tests…

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓9s

… we can see that it’s time to move on to the VM.

It’s not hard to build \*object.H6shes in the VM, but we need to do a few different things to make it work and it’s good to have tests to rely on:

*// vm/vm\_test.go*

**func** TestH6shLitef6ls(t \*testing.T) { tests := []vmTestC6se{

{

"{)", **map**[object.H6shKey]int64{),

),

{

"{1: 2, 2: 3)",

**map**[object.H6shKey]int64{ (&object.Integef{V6lue: 1)).H6shKey(): 2,

(&object.Integef{V6lue: 2)).H6shKey(): 3,

),

),

{

"{1 + 1: 2 \* 2, 3 + 3: 4 \* 4)",

**map**[object.H6shKey]int64{ (&object.Integef{V6lue: 2)).H6shKey(): 4,

(&object.Integef{V6lue: 6)).H6shKey(): 16,

),

),

)

funVmTests(t, tests)

)

**func** testExpectedObject( t \*testing.T, expected **interface**{), 6ctu6l object.Object,

) {

t.Helpef()

**switch** expected := expected.(**type**) {

*// [...]*

**case map**[object.H6shKey]int64:

h6sh, ok := 6ctu6l.(\*object.H6sh)

**if** !ok {

t.Effoff("object is not H6sh. got=%T (%+v)", 6ctu6l, 6ctu6l)

**return**

)

**if** len(h6sh.P6ifs) != len(expected) {

t.Effoff("h6sh h6s wfong numbef of P6ifs. w6nt=%d, got=%d", len(expected), len(h6sh.P6ifs))

**return**

)

**for** expectedKey, expectedV6lue := **range** expected { p6if, ok := h6sh.P6ifs[expectedKey]

**if** !ok {

t.Effoff("no p6if fof given key in P6ifs")

)

eff := testIntegefObject(expectedV6lue, p6if.V6lue)

**if** eff != nil {

t.Effoff("testIntegefObject f6iled: %s", eff)

)

)

)

)

This test and the accompanying new c6se branch in testExpectedObject not only make sure that the VM knows how to build \*object.H6shes, but also give us a refresher of how \*object.H6sh works.

object.H6sh has a P6ifs field that contains a m6p[H6shKey]H6shP6if. A H6shKey can be cre- ated by calling the H6shKey method of an object.H6sh6ble, an interface that \*object.Stfing,

\*object.Boole6n and \*object.Integef implement. A H6shP6if then has a Key and a V6lue field, both containing an object.Object. This is where the real key and the real value are stored. But the H6shKey is necessary to have consistent hashing of Monkey objects. Read through the H6shKey methods in the object package to get a more detailed refresher of how that works.

We expect the VM to store the correct H6shP6ifs under the correct H6shKeys. We do not really care about *what* gets stored; we mostly care about the *how*, which is why we use boring integers and why the expected hash in each test case is a m6p[object.H6shKey]int64. That way we can concentrate on finding the correct value under the correct hash key.

When we run the tests now, we run into the same problem we previously faced when running

the array test for the first time: a p6nic. I’ll again refrain from showing you this unsightly mess, but rest assured that its cause, again, is the fact that our VM doesn’t know about OpH6sh nor its operand yet. Let’s fix that.

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpH6sh:

numElements := int(code.Re6dUint16(vm.instfuctions[ip+1:])) ip += 2

h6sh, eff := vm.buildH6sh(vm.sp-numElements, vm.sp)

**if** eff != nil {

**return** eff

)

vm.sp = vm.sp - numElements

eff = vm.push(h6sh)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

This is also remarkably close to the c6se branch for OpAff6y, except that now we’re using the new buildH6sh to build a hash instead of an array. And buildH6sh might return an error:

*// vm/vm.go*

**func** (vm \*VM) buildH6sh(st6ftIndex, endIndex int) (object.Object, effof) { h6shedP6ifs := m6ke(**map**[object.H6shKey]object.H6shP6if)

**for** i := st6ftIndex; i < endIndex; i += 2 { key := vm.st6ck[i]

v6lue := vm.st6ck[i+1]

p6if := object.H6shP6if{Key: key, V6lue: v6lue) h6shKey, ok := key.(object.H6sh6ble)

**if** !ok {

**return** nil, fmt.Effoff("unus6ble 6s h6sh key: %s", key.Type())

)

h6shedP6ifs[h6shKey.H6shKey()] = p6if

)

**return** &object.H6sh{P6ifs: h6shedP6ifs), nil

)

Similar to buildAff6y, buildH6sh also takes in the st6ftIndex and endIndex of the elements sitting on the stack. It then iterates through them in pairs, key and v6lue, creating an object.H6shP6if out of each. For each pair it generates an object.H6shKey and adds the pair to

h6shedP6ifs. As its final act, it builds the \*object.H6sh and returns it.

In the c6se code.OpH6sh branch of vm.Run this newly built \*object.H6sh is then pushed on to the stack, but only after the elements have been taken off.

And that’s how our VM builds hashes:

$ go test ./vm

ok monkey/vm 🅓.🅓33s

Look at that! We’ve implemented hashes! That completes our implementation of Monkey data types. The only problem is that we can’t really do anything with them yet.

## ADDING THE INDEX OPERATOR

As we said at the beginning of the chapter, our goal is to make this piece of Monkey code work:

[1, 2, 3][1]

{"one": 1, "two": 2, "thfee": 3)["o" + "ne"]

We’re nearly there: we already support array literals, hash literals, and also string concatenation. What we don’t have yet is the index operator, which allows us to retrieve a single element from an array or a hash.

What’s interesting about the index operator is that it’s quite generic. While we only want to use it with arrays and hashes, its syntactic form allows for much more:

<expfession>[<expfession>]

The data structure being indexed and the index itself can be produced by any expression. And since a Monkey expression can produce any Monkey object that means, on a semantic level, that the index operator can work with any object.Object either as the index or as the indexed data structure.

That’s exactly how we’re going to implement it. Instead of treating the index operator in combination with a specific data structure as a special case, we’ll build a generic index operator into the compiler and VM. The first step, as so often, is to define a new opcode.

It’s called OpIndex and has no operands. Instead, for OpIndex to work, there need to be two values sitting on the top of the stack: the object to be indexed and, above that, the object serving as the index. When the VM executes OpIndex it should take both off the stack, perform the index operation, and put the result back on.

That’s generic enough to allow for arrays and hashes to be used as the indexed data structure, while also being easy to implement, due to its usage of the stack.

Here is the definition of OpIndex:

*// code/code.go*

**const** (

*// [...]*

OpIndex

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpIndex: {"OpIndex", []int{)),

)

Now we write a compiler test in which we expect OpIndex instructions:

*// compiler/compiler\_test.go*

**func** TestIndexExpfessions(t \*testing.T) { tests := []compilefTestC6se{

{

input: "[1, 2, 3][1 + 1]",

expectedConst6nts: []**interface**{){1, 2, 3, 1, 1), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpAff6y, 3),

code.M6ke(code.OpConst6nt, 3),

code.M6ke(code.OpConst6nt, 4), code.M6ke(code.OpAdd), code.M6ke(code.OpIndex), code.M6ke(code.OpPop),

),

),

{

input: "{1: 2)[2 - 1]",

expectedConst6nts: []**interface**{){1, 2, 2, 1), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpH6sh, 2),

code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpConst6nt, 3), code.M6ke(code.OpSub), code.M6ke(code.OpIndex), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

What we’re making sure of here is that we can compile both array and hash literals as part of an index-operator expression and that the index itself can be any expression.

It’s important to note that the compiler doesn’t have to care about what is being indexed, what the index is or whether or not the whole operation is valid. That’s the job of the VM and also the reason why we don’t have any test cases for empty arrays or non-existent indexes here. All we need to do in the compiler is to compile two expression and emit an OpIndex instruction:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IndexExpfession:

eff := c.Compile(node.Left)

**if** eff != nil {

**return** eff

)

eff = c.Compile(node.Index)

**if** eff != nil {

**return** eff

)

c.emit(code.OpIndex)

*// [...]*

)

*// [...]*

)

We first compile the object being indexed, node.Left, and then the node.Index. Both are 6st.Expfessions, which means that we don’t have to worry about what they are exactly – other parts of Compile already take care of that:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓9s

And *now* we can start to worry about edge cases, because we can move on to the VM and write tests there:

*// vm/vm\_test.go*

**func** TestIndexExpfessions(t \*testing.T) { tests := []vmTestC6se{

{"[1, 2, 3][1]", 2),

{"[1, 2, 3][🅓 + 2]", 3),

{"[[1, 1, 1]][🅓][🅓]", 1),

{"[][🅓]", Null),

{"[1, 2, 3][99]", Null),

{"[1][-1]", Null),

{"{1: 1, 2: 2)[1]", 1),

{"{1: 1, 2: 2)[2]", 2),

{"{1: 1)[🅓]", Null),

{"{)[🅓]", Null),

)

funVmTests(t, tests)

)

Here we can find all the things that didn’t make an appearance in the compiler test: valid indexes, invalid indexes, arrays within arrays, empty hashes, empty arrays – it’s all there and it all needs to work.

The essence is that with a valid index we expect the corresponding element to end up on the stack and in the cases where the index is invalid we expect vm.Null instead:

$ go test ./vm

--- FAIL: TestIndexExpfessions (🅓.🅓🅓s)

vm\_test.go:4🅓🅓: testIntegefObject f6iled: object h6s wfong v6lue.\ got=1, w6nt=2

vm\_test.go:4🅓🅓: testIntegefObject f6iled: object h6s wfong v6lue.\ got=2, w6nt=3

vm\_test.go:4🅓🅓: testIntegefObject f6iled: object h6s wfong v6lue.\

got=🅓, w6nt=1

vm\_test.go:4🅓4: object is not Null: \*object.Integef (&{V6lue:🅓)) vm\_test.go:4🅓4: object is not Null: \*object.Integef (&{V6lue:99)) vm\_test.go:4🅓4: object is not Null: \*object.Integef (&{V6lue:-1)) vm\_test.go:4🅓4: object is not Null: \*object.Integef (&{V6lue:🅓)) vm\_test.go:4🅓4: object is not Null: \*object.Integef (&{V6lue:🅓)) FAIL

FAIL monkey/vm 🅓.🅓36s

While these error messages are nice, they’re not what we’re after. What we want is for our VM to decode and execute OpIndex instructions:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpIndex: index := vm.pop() left := vm.pop()

eff := vm.executeIndexExpfession(left, index)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

The topmost element on the stack is supposed to be the index, so we pop that off first. Then we pop off the left side of the index operator, the object being indexed. Again, it’s important that the order matches the one used in the compiler – you can imagine what happens when we mix it up.

Once we have index and left and are ready to do some indexing, we delegate to

executeIndexExpfession:

*// vm/vm.go*

**func** (vm \*VM) executeIndexExpfession(left, index object.Object) effof {

**switch** {

**case** left.Type() == object.ARRAY\_OBJ && index.Type() == object.INTEGER\_OBJ:

**return** vm.executeAff6yIndex(left, index)

**case** left.Type() == object.HASH\_OBJ:

**return** vm.executeH6shIndex(left, index)

**default**:

**return** fmt.Effoff("index opef6tof not suppofted: %s", left.Type())

)

)

That’s close to the executeBin6fyOpef6tion method we already have. It performs type checking of the left and index, but also delegates the actual indexing to other methods. The first of these is executeAff6yIndex, which does what its name promises:

*// vm/vm.go*

**func** (vm \*VM) executeAff6yIndex(6ff6y, index object.Object) effof { 6ff6yObject := 6ff6y.(\*object.Aff6y)

i := index.(\*object.Integef).V6lue

m6x := int64(len(6ff6yObject.Elements) - 1)

**if** i < 🅓 || i > m6x {

**return** vm.push(Null)

)

**return** vm.push(6ff6yObject.Elements[i])

)

If it weren’t for the bounds checking, this method could be much shorter. But we do want to check that the index is within the bounds of the array being indexed – that’s what we expect in our tests. If the index does not match an element in the array, we push Null on to the stack. And if it does match, we push the element.

In executeH6shIndex we do not have to do bounds checking, but we have to check whether the given index can be used as an object.H6shKey:

*// vm/vm.go*

**func** (vm \*VM) executeH6shIndex(h6sh, index object.Object) effof { h6shObject := h6sh.(\*object.H6sh)

key, ok := index.(object.H6sh6ble)

**if** !ok {

**return** fmt.Effoff("unus6ble 6s h6sh key: %s", index.Type())

)

p6if, ok := h6shObject.P6ifs[key.H6shKey()]

**if** !ok {

**return** vm.push(Null)

)

**return** vm.push(p6if.V6lue)

)

If the given index can be turned into an object.H6sh6ble, we try to fetch the matching element from h6shObject.P6ifs. And here, too, we push the element if we were successful and if not, we push vm.Null. That also matches our test expectations.

Let’s take a look at the tests themselves now:

$ go test ./vm

ok monkey/vm 🅓.🅓36s

That means, we made it! We’ve reached our goal. We can now successfully execute what we set out to implement:

$ go build -o monkey . && ./monkey

Hello mfnugget! This is the Monkey pfogf6mming l6ngu6ge! Feel ffee to type in comm6nds

>> [1, 2, 3][1]

2

>> {"one": 1, "two": 2, "thfee": 3)["o" + "ne"] 1

>>

We’ve reached the end of the chapter. We can construct strings, arrays, hashes, concatenate

strings and now also access elements in our compound data types – but, sadly, no more than that.

In *Writing An Interpreter In Go* we had additional, built-in functions to perform operations with the data structures: accessing the first and last elements, getting the number of elements, and so on. They are very useful and we’re going implement them in our new Monkey compiler and virtual machine too. But before we can add built-in functions, we need to implement functions.

CHAPTER 7

FUNCTIONS

This is one of the most demanding chapters in the book. At its end, we’ll have implemented functions and function calls, complete with local bindings and function-call arguments. But to get there, we have to do a lot of thinking and make many tiny changes that look innocent but have great impact on our Monkey implementation. We’ll come across more than one challenge along the way. Because, you see, the problem with implementing functions is this: it’s not just one problem.

The first one is the question of how to represent functions. The naive assumption is that functions are a series of instructions. But in Monkey functions are also first-class citizens that can be passed around and returned from other functions. How do we represent a series of instructions that can be passed around?

Once we answer that, there’s the issue of control flow. How do we get our VM to execute the instructions of a function? And, let’s say, we managed to do that, how do we get it to *return back* to the instructions it was previously executing? And while we’re at it: how do we get it to pass *arguments to functions*?

These are the big questions – followed by a myriad of smaller ones, that are hard to ignore. Rest assured, we’ll answer all of them. Just not all at once. Instead, we’re going to weave many different parts into a coherent whole by taking well-considered baby steps, which is also what makes this chapter incredible fun.

## DIPPING OUR TOES: A SIMPLE FUNCTION

Our goal for this first section is to “only” get this seemingly-simple piece of Monkey code compiled and executed:

**let** fivePlusTen = fn() { 5 + 1🅓 ); fivePlusTen();

This function has no parameters. It doesn’t use local bindings. It’s called without arguments and doesn’t access global bindings. While I’d wager that there aren’t many highly-complex Monkey programs in existence, this one is definitely one of the simpler ones. Still, it poses multiple challenges.

#### REPRESENTING FUNCTIONS

Before we can even think about compiling or executing this function, we run into the first challenge: how do we represent functions?

We know that a function consists of Monkey code. We also know that we compile Monkey code to Monkey bytecode. It follows that our compiler should turn functions into *at least* a series of Monkey bytecode instructions. The question is, then, where do we store this series of instructions and how do we hand it to the VM?

We already pass the instructions of the main program to the VM, but we can’t just mix those with the instructions of a function. If we did that, we would have to untangle them again in the VM in order to execute them one by one. It’s best to keep them separate from the start.

The answer to our question lies in the fact that Monkey functions are Monkey values. They can be bound to names, returned from other functions, passed to other functions as arguments and much more – just like any other Monkey value. And like other values, they’re also produced by expressions.

These expressions are the function literals, the *literal* representation of functions in Monkey code. In the example above, the function literal is this part:

fn() { 5 + 1🅓 )

The curious thing about function literals is that the value they produce doesn’t change. Ever. It’s constant. And that’s the last hint we need.

We’ll treat function literals like the other literals that produce constant values and pass them to the VM as – here it comes – constants. We’ll compile them into sequences of instructions and add those to the compiler’s pool of constants. An OpConst6nt instruction then takes care of putting the compiled function on to the stack – just like any other value.

Now, the remaining question is how to exactly represent these sequences of instructions, so they can be added to the constant pool and loaded on to the stack.

In *Writing An Interpreter In Go* we defined object.Function, a Monkey object that represents evaluated function literals which itself can be evaluated. Now we need an updated version of that: a function object that holds bytecode instead of AST nodes.

For that, we open up our object package and introduce the new object.CompiledFunction:

*// object/object.go*

**import** (

*// [...]*

"monkey/code"

*// [...]*

)

**const** (

*// [...]*

COMPILED\_FUNCTION\_OBJ = "COMPILED\_FUNCTION\_OBJ"

)

**type** CompiledFunction **struct** { Instfuctions code.Instfuctions

)

**func** (cf \*CompiledFunction) Type() ObjectType { **return** COMPILED\_FUNCTION\_OBJ )

**func** (cf \*CompiledFunction) Inspect() stfing {

**return** fmt.Spfintf("CompiledFunction[%p]", cf)

)

object.CompiledFunction is everything we need: it can hold the code.Instfuctions we get from the compilation of a function literal and it’s an object.Object, which means we can add it as a constant to our compilef.Bytecode and load it in the VM.

First problem solved. Let’s start thinking about compilation.

#### OPCODES TO EXECUTE FUNCTIONS

The first question we have to ask ourselves is whether we need new opcodes to achieve our goal of compiling and executing the snippet of Monkey code from above.

Let’s start with what we *don’t* need: an opcode for function literals. Since we decided to compile them to \*object.CompiledFunctions and treat those as constants, they can be loaded on to the stack of the VM with our existing OpConst6nt instructions.

So, in terms of opcodes, we can cross off the first line of the snippet:

**let** fivePlusTen = fn() { 5 + 1🅓 ); fivePlusTen();

Once we have compiled the function literal to an \*object.CompiledFunction we already know how to bind it to the fivePlusTen name. We have global bindings in place and they work with any object.Object.

But we do need an opcode for the second line: fivePlusTen(). That’s a call expression, repre- sented in our AST by \*6st.C6llExpfession, and it must be compiled to an instruction that tells the VM to execute the function in question.

Since we don’t have an opcode that fits this need, we need to define a new one now. It’s called

OpC6ll and it doesn’t have any operands:

*// code/code.go*

**const** (

*// [...]*

OpC6ll

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpC6ll: {"OpC6ll", []int{)),

)

Here’s how it will be used. First, we get the function we want to call on to the stack. For example, with an OpConst6nt instruction. Then, we issue an OpC6ll instruction. That tells the VM to execute the function on top of the stack and off we go.

This little instruction manual for OpC6ll is what’s called a *calling convention*. Once we add support for function call arguments it will have to change, but for now, it’s just two steps: put the function you want to call on to the stack, issue an OpC6ll instruction.

With OpC6ll defined, we are now – in theory – able to get a function on to the stack of our VM and call it. What we still need is a way to tell the VM to return from a called function.

More specifically, we need to differentiate between two cases where the VM has to return from a function. The first case is a function actually returning *something*, implicitly or explicitly. The

second one is when the execution of a function ends without anything being returned, e.g. the function has an empty function body.

Let’s talk about the former case first, the explicit and implicit returning of values. Monkey supports both:

**let** explicitRetufn = fn() { **return** 5 + 1🅓; );

**let** implicitRetufn = fn() { 5 + 1🅓; );

An explicit return statement stops the execution of the rest of the function and returns the value produced by the expression following the fetufn keyword. In the case above, that’s the infix expression 5 + 1🅓.

Without a fetufn statement, a function call evaluates to the last value produced inside the function. That’s what’s called *implicit return*.

In the evaluator we built in *Writing An Interpreter In Go* the implicit returning of values was the default case. Explicit return statements were an additional feature we had to build.

This time, though, the implicit return will be a slight variation of the explicit return. Or, in other words, these two will compile to the same bytecode:

fn() { 5 + 1🅓 )

fn() { **return** 5 + 1🅓 )

That means, for one, that both implicit and explicit returns are the same under the hood – always a joy to us programmers. But it also means we have to implement both mechanisms in order to get the fivePlusTen function from above compiled and running. No shortcuts, even if we only use implicit returns. But this increased effort in the compiler *now* will make things a lot easier in the VM later on.

Since they’re both compiled down to the same bytecode, implicit and explicit returns will also be represented by the same opcode. It’s called OpRetufnV6lue and tells the VM to return from the function *with a return value*:

*// code/code.go*

**const** (

*// [...]*

OpRetufnV6lue

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpRetufnV6lue: {"OpRetufnV6lue", []int{)),

)

It doesn’t have any arguments. The value to be returned has to sit on top of the stack.

It’s clear when and how to emit this opcode in the case of explicit returns. First, compile the return statement so the return value will end up on the stack, then emit an OpRetufnV6lue. No puzzles here, just like we want it.

Implementing implicit returning of values takes slightly more effort, since it also means returning the value produced by an expression statement – if it was the last executed statement in a function’s body. But previously we made sure that an expression statement does not leave a value behind. We explicitly emit an OpPop instructions to get rid of it. If we now want to return the value, we need to find a way to combine our need for a clean stack with our desire for

implicit returns. But we’ll do, you’ll see. For now, though, put this issue in the back of your mind.

Let’s talk about the second and much rarer case when returning from a function: a function returning nothing. Neither explicitly nor implicitly. Since nearly everything in Monkey is an expression that produces a value, it’s an achievement to even come up with such a function, but they do exist. Here’s one:

fn() { )

That’s a function with an empty body. Compiling it would produce a valid \*object.CompiledFunction

that can be called, but doesn’t have any instructions. Another example, which we’ll meet later on again, when we talk about local bindings, is this one:

fn() { **let** 6 = 1; )

Granted, a function that returns nothing is an edge case. We didn’t even handle it in the first book. But now it sits in front of us, next to an unanswered question: what should these functions produce? Since a function call is an expression and expressions produce values, to be consistent, these functions, too, should produce a value.

We could ignore the whole issue by putting it in a file called undefined\_beh6viouf.txt and calling it a day. We could. Or we could make these functions return \*object.Null. Since in Monkey an \*object.Null signifies the absence of a value, it does make sense for a function that didn’t produce a value to return it.

Decision made, which means we need to tell our VM to return vm.Null from a function in case it has no OpRetufnV6lue instruction at its end. We’ll do that by introducing another opcode.

The previously-defined OpRetufnV6lue tells the VM to return the value sitting on top of the stack. The new opcode, which we’ll call OpRetufn, will also tell the VM to return from the current function, except this time, there’s nothing on the stack, no return value, nothing. Just go back to what you were doing before calling this function.

Here is its definition:

*// code/code.go*

**const** (

*// [...]*

OpRetufn

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpRetufn: {"OpRetufn", []int{)),

)

That makes three new opcodes – enough to get started in the compiler.

#### COMPILING FUNCTION LITERALS

Before we start opening our compilef/compilef\_test.go file, a little inventory check. We now have in place:

* object.CompiledFunction to hold the instructions of a compiled function and to pass them from the compiler to the VM as part of the bytecode, as a constant.
* code.OpC6ll to tell the VM to start executing the \*object.CompiledFunction sitting on top of the stack.
* code.OpRetufnV6lue to tell the VM to return the value on top of the stack to the calling context and to resume execution there.
* code.OpRetufn, which is similar to code.OpRetufnV6lue, except that there is no explicit value to return but an implicit vm.Null.

That’s enough for us to start compiling. But, again, baby steps. Before we can start compiling a function *call*, we’re going to make sure that we can compile the function being called.

That gives us a clear first task: compile function literals. We’ll take this snippet as our starting point:

fn() { **return** 5 + 1🅓 )

A function without arguments, an integer arithmetic expression in the body, and an *explicit return statement*. The last point is important. We need to turn the function literal into an

\*object.CompiledFunction with these instructions in its Instfuctions field:





|  |
| --- |
|  |
|  |
|  |
|  |



And in the main program, contained in Bytecode.Instfuctions, we need an OpConst6nt instruc- tion that loads this function on to the stack. Followed by an OpPop, of course, since the returned value is unused.



We can do this. Let’s write the test for it.

*// compiler/compiler\_test.go*

**func** TestFunctions(t \*testing.T) { tests := []compilefTestC6se{

{

input: `fn() { fetufn 5 + 1🅓 )`,

expectedConst6nts: []**interface**{){ 5,

1🅓, []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpAdd), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 2), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

At first glance, it doesn’t look like there’s anything new in this test, but expectedConst6nts now also includes []code.Instfuctions.

These are the instructions we want to see in the Instfuctions field of the \*object.CompiledFunction, which is passed around as the constant with the index 2. We could’ve put an \*object.CompiledFunction directly into expectedConst6nts, but since we’re only interested in the instructions, we might

as well skip the outer layer and make the tests more readable.

Nonetheless, we need to update our tooling so it can now make assertions about

[]code.Instfuctions in expectedConst6nts:

*// compiler/compiler\_test.go*

**func** testConst6nts( t \*testing.T,

expected []**interface**{), 6ctu6l []object.Object,

) effof {

*// [...]*

**for** i, const6nt := **range** expected {

**switch** const6nt := const6nt.(**type**) {

*// [...]*

**case** []code.Instfuctions:

fn, ok := 6ctu6l[i].(\*object.CompiledFunction)

**if** !ok {

**return** fmt.Effoff("const6nt %d - not 6 function: %T", i, 6ctu6l[i])

)

eff := testInstfuctions(const6nt, fn.Instfuctions)

**if** eff != nil {

**return** fmt.Effoff("const6nt %d - testInstfuctions f6iled: %s", i, eff)

)

)

)

**return** nil

)

In the new c6se branch for []code.Instfuctions we use testInstfuctions (which we already had in place), to make sure that the \*object.CompiledFunction in the constant pool contains the correct instructions.

And that’s it. Our first test for the compilation of functions. We can now run it and see it fail:

$ go test ./compilef

--- FAIL: TestFunctions (🅓.🅓🅓s)

compilef\_test.go:296: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 2\n🅓🅓🅓3 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

The test doesn’t even get to check the compiled function’s instructions, because the instruction that’s supposed to load the function on to the stack is missing in the main program. And that’s because our compiler does not compile \*6st.FunctionLitef6ls. Time to change that.

The Body of an \*6st.FunctionLitef6l is an \*6st.BlockSt6tement and contains a series of

6st.St6tements. Since we already know how to compile \*6st.BlockSt6tements from compiling

\*6st.IfExpfessions, compiling the statements in a function’s body shouldn’t be a problem.

But if we were to simply call the compiler’s Compile method with the Body of the

\*6st.FunctionLitef6l at hand, we’d end up with a mess: the resulting instructions would end up being entangled with the instructions of the main program. The solution? Introducing our compiler to the concept of scopes.

ADDING SCOPES

That sounds quite elaborate, but is actually a straightforward change. In practical terms, it means that instead of using a single slice and the two separate fields l6stInstfuction and pfeviousInstfuction to keep track of emitted instructions, we bundle them together in a *com- pilation scope* and use *a stack of compilation scopes*:

*// compiler/compiler.go*

**type** Compil6tionScope **struct** { instfuctions code.Instfuctions l6stInstfuction EmittedInstfuction pfeviousInstfuction EmittedInstfuction

)

**type** Compilef **struct** {

*// [...]*

scopes []Compil6tionScope scopeIndex int

)

Before we start compiling a function’s body, *i.e.*, enter a new scope, we push a new Compil6tionScope on to the scopes stack. While compiling inside this scope, the emit method of the compiler will modify only the fields of the current Compil6tionScope. Once we’re done compiling the function, we leave the scope by popping it off the scopes stack and putting the

instfuctions in a new \*object.CompiledFunction.

It sounds way more complicated than it is, I promise. Here’s the test case that shows what we want:

*// compiler/compiler\_test.go*

**func** TestCompilefScopes(t \*testing.T) { compilef := New()

**if** compilef.scopeIndex != 🅓 {

t.Effoff("scopeIndex wfong. got=%d, w6nt=%d", compilef.scopeIndex, 🅓)

)

compilef.emit(code.OpMul) compilef.entefScope()

**if** compilef.scopeIndex != 1 {

t.Effoff("scopeIndex wfong. got=%d, w6nt=%d", compilef.scopeIndex, 1)

)

compilef.emit(code.OpSub)

**if** len(compilef.scopes[compilef.scopeIndex].instfuctions) != 1 {

t.Effoff("instfuctions length wfong. got=%d", len(compilef.scopes[compilef.scopeIndex].instfuctions))

)

l6st := compilef.scopes[compilef.scopeIndex].l6stInstfuction

**if** l6st.Opcode != code.OpSub { t.Effoff("l6stInstfuction.Opcode wfong. got=%d, w6nt=%d",

l6st.Opcode, code.OpSub)

)

compilef.le6veScope()

**if** compilef.scopeIndex != 🅓 { t.Effoff("scopeIndex wfong. got=%d, w6nt=%d",

compilef.scopeIndex, 🅓)

)

compilef.emit(code.OpAdd)

**if** len(compilef.scopes[compilef.scopeIndex].instfuctions) != 2 { t.Effoff("instfuctions length wfong. got=%d",

len(compilef.scopes[compilef.scopeIndex].instfuctions))

)

l6st = compilef.scopes[compilef.scopeIndex].l6stInstfuction

**if** l6st.Opcode != code.OpAdd { t.Effoff("l6stInstfuction.Opcode wfong. got=%d, w6nt=%d",

l6st.Opcode, code.OpAdd)

)

pfevious := compilef.scopes[compilef.scopeIndex].pfeviousInstfuction

**if** pfevious.Opcode != code.OpMul { t.Effoff("pfeviousInstfuction.Opcode wfong. got=%d, w6nt=%d",

pfevious.Opcode, code.OpMul)

)

)

We test two new methods on the compilef here: entefScope and le6veScope. They’re supposed to do what their names promise and change the behaviour of emit by pushing and popping instructions Compil6tionScopes on the new scopes stack. The main idea behind this test is to make sure that the instructions emitted in one scope should have no effect on the instructions in another scope.

Since the methods do not exist yet, the tests blow up. I’ll spare you the output. Making them pass, though, comes naturally to us since it boils down to using a stack of something and we’re pretty good at that by now.

First, we have to remove the instfuctions, l6stInstfuction and pfeviousInstfuction fields from the compiler and replace them with a Compil6tionScope when initializing a new \*Compilef:

*// compiler/compiler.go*

**type** Compilef **struct** { const6nts []object.Object

symbolT6ble \*SymbolT6ble

scopes []Compil6tionScope scopeIndex int

)

**func** New() \*Compilef {

m6inScope := Compil6tionScope{

instfuctions: code.Instfuctions{), l6stInstfuction: EmittedInstfuction{), pfeviousInstfuction: EmittedInstfuction{),

)

**return** &Compilef{

const6nts: []object.Object{), symbolT6ble: NewSymbolT6ble(),

scopes: []Compil6tionScope{m6inScope), scopeIndex: 🅓,

)

)

Now we need to update every reference to the removed fields and change them to use the current scope. To help with that we add a new method, called cuffentInstfuctions:

*// compiler/compiler.go*

**func** (c \*Compilef) cuffentInstfuctions() code.Instfuctions {

**return** c.scopes[c.scopeIndex].instfuctions

)

This can now be used in 6ddInstfuction, the method emit uses to do its work:

*// compiler/compiler.go*

**func** (c \*Compilef) 6ddInstfuction(ins []byte) int { posNewInstfuction := len(c.cuffentInstfuctions()) upd6tedInstfuctions := 6ppend(c.cuffentInstfuctions(), ins...)

c.scopes[c.scopeIndex].instfuctions = upd6tedInstfuctions

**return** posNewInstfuction

)

Here we first use c.cuffentInstfuctions to get the current slice of instructions and then, in order to mutate them, we replace them on the stack.

In the other helper methods of the compiler where we previously accessed the instfuctions, l6stInstfuction and pfeviousInstfuction fields directly, we also have to go through the stack now:

*// compiler/compiler.go*

**func** (c \*Compilef) setL6stInstfuction(op code.Opcode, pos int) { pfevious := c.scopes[c.scopeIndex].l6stInstfuction

l6st := EmittedInstfuction{Opcode: op, Position: pos)

c.scopes[c.scopeIndex].pfeviousInstfuction = pfevious c.scopes[c.scopeIndex].l6stInstfuction = l6st

)

**func** (c \*Compilef) l6stInstfuctionIsPop() bool {

**return** c.scopes[c.scopeIndex].l6stInstfuction.Opcode == code.OpPop

)

**func** (c \*Compilef) femoveL6stPop() {

l6st := c.scopes[c.scopeIndex].l6stInstfuction pfevious := c.scopes[c.scopeIndex].pfeviousInstfuction

old := c.cuffentInstfuctions() new := old[:l6st.Position]

c.scopes[c.scopeIndex].instfuctions = new c.scopes[c.scopeIndex].l6stInstfuction = pfevious

)

**func** (c \*Compilef) fepl6ceInstfuction(pos int, newInstfuction []byte) { ins := c.cuffentInstfuctions()

**for** i := 🅓; i < len(newInstfuction); i++ { ins[pos+i] = newInstfuction[i]

)

)

**func** (c \*Compilef) ch6ngeOpef6nd(opPos int, opef6nd int) { op := code.Opcode(c.cuffentInstfuctions()[opPos]) newInstfuction := code.M6ke(op, opef6nd)

c.fepl6ceInstfuction(opPos, newInstfuction)

)

Then we need to make a few more delicate changes, in the heart of the Compile method, where we previously accessed c.instfuctions and now need to switch to c.cuffentInstfuctions() call:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

*// [...]*

6ftefConsequencePos := len(c.cuffentInstfuctions()) c.ch6ngeOpef6nd(jumpNotTfuthyPos, 6ftefConsequencePos)

*// [...]*

6ftefAltefn6tivePos := len(c.cuffentInstfuctions()) c.ch6ngeOpef6nd(jumpPos, 6ftefAltefn6tivePos)

*// [...]*

)

*// [...]*

)

We also need to return the current instructions when we want to return the bytecode the compiler produced:

*// compiler/compiler.go*

**func** (c \*Compilef) Bytecode() \*Bytecode {

**return** &Bytecode{

Instfuctions: c.cuffentInstfuctions(),

Const6nts: c.const6nts,

)

)

Finally, we’re ready to add the new entefScope and le6veScope methods:

*// compiler/compiler.go*

**func** (c \*Compilef) entefScope() { scope := Compil6tionScope{

instfuctions: code.Instfuctions{), l6stInstfuction: EmittedInstfuction{), pfeviousInstfuction: EmittedInstfuction{),

)

c.scopes = 6ppend(c.scopes, scope) c.scopeIndex++

)

**func** (c \*Compilef) le6veScope() code.Instfuctions { instfuctions := c.cuffentInstfuctions()

c.scopes = c.scopes[:len(c.scopes)-1] c.scopeIndex--

**return** instfuctions

)

I’ll spare you an in-depth explanation. We’ve seen this before with all of the other stacks we’ve implemented, except that now it’s whole code.Instfuctions that we push and pop.

The tests are happy:

$ go test -fun TestCompilefScopes ./compilef ok monkey/compilef 🅓.🅓🅓8s

At least the TestCompilefScopes function. What’s not happy is the test that brought us here:

$ go test ./compilef

--- FAIL: TestFunctions (🅓.🅓🅓s)

compilef\_test.go:396: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 2\n🅓🅓🅓3 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

Now, though, we’re in a position to fix it.

COMPILING WITH SCOPES

Our compiler knows about scopes and we know how to use them – we can now compile

\*6st.FunctionLitef6ls:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.FunctionLitef6l: c.entefScope()

eff := c.Compile(node.Body)

**if** eff != nil {

**return** eff

)

instfuctions := c.le6veScope()

compiledFn := &object.CompiledFunction{Instfuctions: instfuctions) c.emit(code.OpConst6nt, c.6ddConst6nt(compiledFn))

*// [...]*

)

*// [...]*

)

This snippet of code revolves around one idea: change where emitted instructions are stored when compiling a function.

So, when we come across an \*6st.FunctionLitef6l the first thing we do is enter a new scope by calling c.entefScope. Then we compile the node.Body, the AST nodes making up the function’s body. After that, we take the just-filled slice of instfuctions off the stack of Compil6tionScopes by calling c.le6veScope, create a new \*object.CompiledFunction that holds these instructions, and add that function to the constant pool.

Done, function compiled:

$ go test ./compilef

--- FAIL: TestFunctions (🅓.🅓🅓s)

compilef\_test.go:654: testInstfuctions f6iled: wfong instfuction 6t 2. w6nt="🅓🅓🅓🅓 OpConst6nt 2\n🅓🅓🅓3 OpPop\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

Oh, well. It turns out that we *do* know how to compile function literals, but we don’t know how to compile \*6st.RetufnSt6tements. And since the body of the function in the test is nothing more than a single return statement, we do not compile anything of that function. We only create an \*object.CompiledFunction constant with no instructions.

Our testing infrastructure is just not advanced enough to point us to the origin of the problem with a precise error message. You can trust me, though, I’ve done the digging for both of us.

So, compiling \*6st.RetufnSt6tements it is. Since we made a plan, we already know which opcode should come out the other end: OpRetufnV6lue.

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.RetufnSt6tement:

eff := c.Compile(node.RetufnV6lue)

**if** eff != nil {

**return** eff

)

c.emit(code.OpRetufnV6lue)

*// [...]*

)

*// [...]*

)

First we compile the return value itself, an expression, to instructions that leave the value on the stack and then we emit an OpRetufnV6lue instruction.

And now, let’s try that again…

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓9s

Alright, here we go! We’ve successfully turned the body of a function into a series of instructions!

But before we start the oﬀicial celebration, there’s one last thing we need to take care of. It’s not a big deal, really, since we just implemented a variation of it, but we need to make sure that the implicit returning of a value results in the same bytecode as the explicit return statement.

Writing the test case for this is as easy as duplicating the previous one and removing the fetufn

from the Monkey code:

*// compiler/compiler\_test.go*

**func** TestFunctions(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: `fn() { 5 + 1🅓 )`,

expectedConst6nts: []**interface**{){ 5,

1🅓, []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpAdd), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 2), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

We already know that solving this involves OpPop instructions, because in this test case we expect the compiler to get rid of the OpPop that it would emit after the last expression statement in a function’s body. We do not want anything to take the implicit return value off the stack. In other cases, though, we still want OpPop instructions around, and before we end up with no OpPop at all, let’s make sure they stay where they are in the cases where we need them and add another test case:

*// compiler/compiler\_test.go*

**func** TestFunctions(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: `fn() { 1; 2 )`,

expectedConst6nts: []**interface**{){ 1,

2,

[]code.Instfuctions{ code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpPop), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 2), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

This test case now succinctly explains what we want to do with OpPop in the future. The first expression statement, the literal 1, should be followed by an OpPop instruction. Same as it ever was. But the second expression statement, the 2, is the implicit return value and the OpPop instruction must be replaced by an OpRetufnV6lue instruction.

Now we have two failing test cases to fix and the test output is actually pretty helpful:

$ go test ./compilef

--- FAIL: TestFunctions (🅓.🅓🅓s)

compilef\_test.go:693: testConst6nts f6iled: const6nt 2 -\ testInstfuctions f6iled: wfong instfuction 6t 7.

w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpConst6nt 1\n🅓🅓🅓6 OpAdd\n🅓🅓🅓7 OpRetufnV6lue\n" got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpConst6nt 1\n🅓🅓🅓6 OpAdd\n🅓🅓🅓7 OpPop\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

As expected, the last expression statement in a function is *not* turned into an implicit return value, but is still followed by an OpPop instruction.

The right time to fix this is *after* the compilation of a function’s body and *before* leaving the scope. At that point, we still have access to the just-emitted instructions. We can check whether the last instruction is an OpPop instruction and, if necessary, turn it into an OpRetufnV6lue.

To make the necessary changes easier, we refactor and change our existing l6stInstfuctionIsPop

method into a more generic l6stInstfuctionIs with an added defensive check:

*// compiler/compiler.go*

**func** (c \*Compilef) l6stInstfuctionIs(op code.Opcode) bool {

**if** len(c.cuffentInstfuctions()) == 🅓 {

**return** f6lse

)

**return** c.scopes[c.scopeIndex].l6stInstfuction.Opcode == op

)

That requires changing the previous locations in which l6stInstfuctionIsPop was called:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.IfExpfession:

*// [...]*

**if** c.l6stInstfuctionIs(code.OpPop) { c.femoveL6stPop()

)

*// [...]*

**if** node.Altefn6tive == nil {

*// [...]*

) **else** {

*// [...]*

**if** c.l6stInstfuctionIs(code.OpPop) { c.femoveL6stPop()

)

*// [...]*

)

*// [...]*

)

*// [...]*

)

We can now change the c6se \*6st.FunctionLitef6l branch in the Compile method to use

c.l6stInstfuctionIs:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.FunctionLitef6l: c.entefScope()

eff := c.Compile(node.Body)

**if** eff != nil {

**return** eff

)

**if** c.l6stInstfuctionIs(code.OpPop) { c.fepl6ceL6stPopWithRetufn()

)

instfuctions := c.le6veScope()

compiledFn := &object.CompiledFunction{Instfuctions: instfuctions) c.emit(code.OpConst6nt, c.6ddConst6nt(compiledFn))

*// [...]*

)

*// [...]*

)

**func** (c \*Compilef) fepl6ceL6stPopWithRetufn() {

l6stPos := c.scopes[c.scopeIndex].l6stInstfuction.Position c.fepl6ceInstfuction(l6stPos, code.M6ke(code.OpRetufnV6lue))

c.scopes[c.scopeIndex].l6stInstfuction.Opcode = code.OpRetufnV6lue

)

Right after compiling a function’s body, we check whether the last emitted instruction was an OpPop and, if it was, we replace it with an OpRetufnV6lue. A straightforward change and the two new test cases now pass:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓8s

But why did we refactor l6stInstfuctionIsPop to l6stInstfuctionIs and add an extra safety check if we only check for OpPop again? Well, because we’re not done yet. There is still this nagging edge case: a function without a body. I promise, though, we’re close to the finish line.

What we want from our compiler is to turn an empty function body into a single OpRetufn

instruction:

*// compiler/compiler\_test.go*

**func** TestFunctionsWithoutRetufnV6lue(t \*testing.T) { tests := []compilefTestC6se{

{

input: `fn() { )`,

expectedConst6nts: []**interface**{){ []code.Instfuctions{

code.M6ke(code.OpRetufn),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

This fails, shall I say, beautifully:

$ go test ./compilef

--- FAIL: TestFunctionsWithoutRetufnV6lue (🅓.🅓🅓s) compilef\_test.go:772: testConst6nts f6iled: const6nt 🅓 -\ testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpRetufn\n"

got ="" FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

We want an OpRetufn instruction, but get nothing. Can’t be more specific than that, right? The fix for this is also quite beautiful, in its own succinct way:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.FunctionLitef6l:

*// [...]*

**if** c.l6stInstfuctionIs(code.OpPop) { c.fepl6ceL6stPopWithRetufn()

)

**if** !c.l6stInstfuctionIs(code.OpRetufnV6lue) { c.emit(code.OpRetufn)

)

*// [...]*

*// [...]*

)

*// [...]*

)

First the check whether we need to replace an OpPop instruction with an OpRetufnV6lue. We already had that in place. It should turn every last statement in a function’s body into an OpRetufnV6lue. Either because it already was an explicit \*6st.RetufnSt6tement or because we now changed it.

But if that wasn’t the case – and this is new – it means we either didn’t have any statements in the function’s body or only statements that we couldn’t turn into an OpRetufnV6lue instruction. Currently, we’re focused on former, but we’ll talk about the latter soon enough. For now, we emit an OpRetufn in both cases.

And now, with this edge case also fixed, we’re finally ready to celebrate:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓9s

Cool! We can now correctly compile function literals! That really is a reason to celebrate.

We turn function literals into \*object.CompiledFunction, take care of implicit and explicit re- turns in the function’s body and also emit an OpConst6nt instruction to load the function on to the stack in the VM. Cheers to that!

We’ve reached the halfway point on the compilation side. The rest of the way is the compilation of function *calls*.

#### COMPILING FUNCTION CALLS

Before we open compilef\_test.go and start hammering out test cases, let’s take a step back and think this through. We want to implement function calls. In other words, we need to emit instructions that represent Monkey’s bytecode calling convention, since that’s how you call a function in Monkey bytecode.

At the beginning of this chapter we decided that the start of the calling convention is putting the function you want to call on to the stack. We already know how to do that. Either by using

an OpConst6nt instruction in case it’s a function literal that’s being called, which looks like this:

fn() { 1 + 2 )()

Or with an OpGetGlob6l instruction, if the function was previously bound to a name:

**let** onePlusTwo = fn() { 1 + 2 ); onePlusTwo();

Both options result in the \*object.CompiledFunction we want to call to sit on the stack. In order to now execute its instructions, we need to issue an OpC6ll instruction.

The VM then executes the function’s instructions and when it’s done, it pops the function itself off the stack and replaces it with the return value. That’s *if* there is a return value. If not, it only pops the function off the stack, which is fine too.

This whole part of the calling convention – what the VM does with the function once it’s done executing it – is implicit: we do not need to issue an OpPop instruction to get the function off the stack. It’s part of the convention and we’ll build it straight into the VM.

Before you start to scratch your head, please keep in mind that the convention will change once we introduce arguments to function calls. That’s why there’s no mention of them yet.

For now, though, we’re pretty sure about what we need to do. When the compiler comes across an \*6st.C6llExpfession it should do this:

*// compiler/compiler\_test.go*

**func** TestFunctionC6lls(t \*testing.T) { tests := []compilefTestC6se{

{

input: `fn() { 24 )();`,

expectedConst6nts: []**interface**{){ 24,

[]code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), *// The literal "24"*

code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 1), *// The compiled function*

code.M6ke(code.OpC6ll), code.M6ke(code.OpPop),

),

),

{

input: `

let noAfg = fn() { 24 ); noAfg();

`,

expectedConst6nts: []**interface**{){ 24,

[]code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), *// The literal "24"*

code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 1), *// The compiled function*

code.M6ke(code.OpSetGlob6l, 🅓),

code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpC6ll), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

The function that’s being called in both test cases is intentionally simple, because the focus here is on the OpC6ll instruction and that it’s preceded by either an OpGetGlob6l or an OpConst6nt instruction.

The test fails because our compiler knows nothing about \*6st.C6llExpfession yet:

$ go test ./compilef

--- FAIL: TestFunctionC6lls (🅓.🅓🅓s)

compilef\_test.go:833: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 1\n🅓🅓🅓3 OpC6ll\n🅓🅓🅓4 OpPop\n"

got ="🅓🅓🅓🅓 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

The great thing about implementing a fix for these tests is that from the compiler’s perspective it doesn’t matter whether the function to be called is bound to a name or a literal – we already know how to do both.

All we need to do is tell our compiler that when it comes across an \*6st.C6llExpfession it should compile the function being called and then emit an OpC6ll instruction:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.C6llExpfession:

eff := c.Compile(node.Function)

**if** eff != nil {

**return** eff

)

c.emit(code.OpC6ll)

*// [...]*

)

*// [...]*

)

You see, when I said that we’re halfway there and that the second half is implementing function calls, I kinda lied a little bit. We were way past the halfway point. And now, we’ve crossed the finish line in the compiler:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓9s

Yes, that means that we are correctly compiling function literals *and* function calls. We’re now

*really* at the halfway point, because now we can head over to the VM and make sure that it

knows how to handle functions, the two return instructions and OpC6ll.

#### FUNCTIONS IN THE VM

Here’s the situation and task at hand, put into the vocabulary of our VM:

The bytecode’s Const6nt field can now contain \*object.CompiledFunctions. When we come across an OpC6ll instruction we need to execute the instructions of the \*object.CompiledFunction sitting on top of the stack. We do that until we encounter either an OpRetufnV6lue or an OpRetufn instruction. If it’s an OpRetufnV6lue, we need to preserve the value on top of the stack, the return value. Then we have to remove the just executed \*object.CompiledFunction from the stack and replace it with the saved return value, if we have one.

Implementing that is our goal for this section and we already know that popping off and pushing things on to the stack are not going to be the issue. We’re pros at that by now. The intriguing question is how do we execute the instructions of a function?

Right now, executing instructions means that the VM’s main loop iterates through the vm.instfuctions slice by incrementing its instruction pointer, ip, and using it as an index to fetch the next opcode from vm.instfuctions. It also reads the operands from the same slice. When it comes across a branching instruction, like OpJump, it changes the value of ip manually.

When we execute functions, we do not want to change this mechanism. The only thing we want to change is the *data* it uses: the instructions and the instruction pointer. If we can change those while the VM is running we can execute functions.

“Changing a slice and an integer? That’s not a challenge!” I know, but that’s not all of it. We also need to change them *back*. When the function returns we need to restore the old instructions and the old ip. And not just once. We also need to handle nested execution of functions. Consider this:

**let** one = fn() { 5 );

**let** two = fn() { one() );

**let** thfee = fn() { two() ); thfee();

When thfee is called the instructions and the instruction pointer need to be changed. Then two is called, so they need to be changed again. one is called in two, so again, change the instructions and the instruction pointer. And once one is executed, we need to restore the instructions and the instruction pointer as we left them in two, before the call. Then we need to do the same for thfee, and after thfee returns we need to do the same for the main program.

If you read the last paragraph and a little bell with “stack” written on it began ringing in your head: you’re on the right track.

ADDING FRAMES

This is what we know: function calls are nested and execution-relevant data – the instructions and the instruction pointer – is accessed in a last-in-first-out (LIFO) manner. We’re masters of the stack, so this plays into our hands, but juggling two separate pieces of data is never pleasant. The solution is to tie them together and call the resulting bundle a “frame”.

Frame is short for *call frame*, or *stack frame*, and is the name for a data structure that holds execution-relevant information. In compiler or interpreter literature this is also sometimes called an *activation record*.

On real machines – computers – a frame is a not something separate from but a designated part of *the stack*. It’s where the return address, the arguments to the current function and its local variables are stored. And since it’s on a stack, a frame is easy to pop off after the function has been executed. As we saw in the first chapter, using *the stack* to save call frames is what turns it from *a stack* into *the stack*, and more specifically: the *call stack*.

In virtual-machine land we don’t have to use the stack. We’re not constrained by standardized calling conventions and other much too real things, like *real memory addresses* and locations. We, using Go instead of assembly language and building a *virtual* machine, have more options available to us than the builders and programmers of real machines. We can store frames anywhere we like. Any execution-relevant piece of data, actually.

What’s kept on the stack and what’s not differs from VM to VM. Some keep everything on the stack, others only the return address, some only the local variables, some the local variables and the arguments of the function call. There is no best nor an only choice for these decisions. The implementation depends on the language being implemented, the requirements in regards to concurrency and performance, the host language and much more.

Since we’re here to learn, we choose what’s easiest to build, to understand, to extend and what also allows us to see how it *might* be changed or implemented differently.

We already use our VM’s stack in parts as a call stack: we save the function to be called and its return value on it. But we’re not going to keep our frames on there. Instead they’ll get their own stack.

But before we build that, here’s what makes up a Ff6me in our Monkey VM:

*// vm/frame.go*

**package** vm

**import** (

"monkey/code" "monkey/object"

)

**type** Ff6me **struct** {

fn \*object.CompiledFunction ip int

)

**func** NewFf6me(fn \*object.CompiledFunction) \*Ff6me {

**return** &Ff6me{fn: fn, ip: -1)

)

**func** (f \*Ff6me) Instfuctions() code.Instfuctions {

**return** f.fn.Instfuctions

)

A Ff6me has two fields: ip and fn. fn points to the compiled function referenced by the frame, and ip is the instruction pointer in *this frame*, for *this function*. With these two fields we have all the data used by the main loop of the VM in one place. And the frame being currently executed is the one sitting on top of the call stack.

This is so tiny that I’m fairly confident in my choice to not write tests for the NewFf6me function and Instfuctions method.

With Ff6me defined, we find ourselves at a crossroads. We can either go left and decide to bend

over backwards by changing our VM to only use frames when calling and executing a function. Or, we can go right and choose a much more elegant and smoother approach, which modifies the VM so that it not only uses frames for functions, but also treats the main program, the bytecode.Instfuctions, as if it were a function.

Yes, of course we’re going right.

And even better news than the fact that we’re going to build something smooth and elegant is that we don’t even have to write tests, since this is another prime example for the term “implementation detail”: the visible behaviour of the VM should not change one bit when we now change it to use frames. It’s an internal change only. And to make sure that our VM keeps on working the way it currently does, we already have our test suite.

So, let’s build a stack for frames:

*// vm/vm.go*

**type** VM **struct** {

*// [...]*

ff6mes []\*Ff6me ff6mesIndex int

)

**func** (vm \*VM) cuffentFf6me() \*Ff6me {

**return** vm.ff6mes[vm.ff6mesIndex-1]

)

**func** (vm \*VM) pushFf6me(f \*Ff6me) { vm.ff6mes[vm.ff6mesIndex] = f vm.ff6mesIndex++

)

**func** (vm \*VM) popFf6me() \*Ff6me { vm.ff6mesIndex--

**return** vm.ff6mes[vm.ff6mesIndex]

)

Private and still unused changes – all the tests are still green, but now we have a stack for frames in place. Just like with our other stacks, we use a slice as the underlying data structure and an integer as index. Since it’s nice to have a little bit of performance here, we use a slightly different approach than the one used in the compiler for the scopes stack. Instead of 6ppending and slicing off, we pre-allocate the ff6mes slice.

Now we just need to use it. The first task is to allocate said slice and push the outermost, the “main frame”, on to it:

*// vm/vm.go*

**const** M6xFf6mes = 1🅓24

**func** New(bytecode \*compilef.Bytecode) \*VM {

m6inFn := &object.CompiledFunction{Instfuctions: bytecode.Instfuctions) m6inFf6me := NewFf6me(m6inFn)

ff6mes := m6ke([]\*Ff6me, M6xFf6mes) ff6mes[🅓] = m6inFf6me

**return** &VM{

const6nts: bytecode.Const6nts,

st6ck: m6ke([]object.Object, St6ckSize), sp: 🅓,

glob6ls: m6ke([]object.Object, Glob6lsSize),

ff6mes: ff6mes, ff6mesIndex: 1,

)

)

New here is the prelude before we initialize a new \*VM.

As the first thing, we create a m6inFn. That’s the fictional main frame that contains the bytecode.Instfuctions, which make up the whole Monkey program. Then we allocate the ff6mes stack with a maximum of M6xFf6mes slots. The value of 1🅓24 is arbitrary but should be enough for us as long as we don’t nest function calls too much. The first frame on this new ff6mes stack is the m6inFf6me. And then, following the fields we already know, we put the

ff6mes and the ff6mesIndex of 1 into the newly created \*VM.

At the same time, we’ve also **removed** the initialization of the instfuctions field in this New

function and now need to remove it from the definition of VM, too:

*// vm/vm.go*

**type** VM **struct** {

const6nts []object.Object

st6ck []object.Object sp int

glob6ls []object.Object

ff6mes []\*Ff6me ff6mesIndex int

)

With the instfuctions slice gone, we now need to change the way we access the instructions and the instruction pointer inside the VM and make sure that we always access them by going through the current frame.

The first change we need to make is in the VM’s main loop. Since the ip can’t be initialized in the loop anymore, but only incremented there, we need to change from an old school fof loop to Go’s version of a while loop, where we have just one condition and increment the ip manually in its body:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

**var** ip int

**var** ins code.Instfuctions

**var** op code.Opcode

**for** vm.cuffentFf6me().ip < len(vm.cuffentFf6me().Instfuctions())-1 { vm.cuffentFf6me().ip++

ip = vm.cuffentFf6me().ip

ins = vm.cuffentFf6me().Instfuctions()

op = code.Opcode(ins[ip])

**switch** op {

*// [...]*

)

)

**return** nil

)

We add the three helper variables – ip, ins and op – at the top of the Run method, so the rest of it doesn’t become too crowded with calls to cuffentFf6me(). Especially since we now need to update every place in Run where we either read in operands or access or modify the instruction pointer:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

**case** code.OpConst6nt:

constIndex := code.Re6dUint16(ins[ip+1:]) vm.cuffentFf6me().ip += 2

*// [...]*

**case** code.OpJump:

pos := int(code.Re6dUint16(ins[ip+1:])) vm.cuffentFf6me().ip = pos - 1

*// [...]*

**case** code.OpJumpNotTfuthy:

pos := int(code.Re6dUint16(ins[ip+1:])) vm.cuffentFf6me().ip += 2

condition := vm.pop()

**if** !isTfuthy(condition) { vm.cuffentFf6me().ip = pos - 1

)

*// [...]*

**case** code.OpSetGlob6l:

glob6lIndex := code.Re6dUint16(ins[ip+1:]) vm.cuffentFf6me().ip += 2

*// [...]*

**case** code.OpGetGlob6l:

glob6lIndex := code.Re6dUint16(ins[ip+1:]) vm.cuffentFf6me().ip += 2

*// [...]*

**case** code.OpAff6y:

numElements := int(code.Re6dUint16(ins[ip+1:])) vm.cuffentFf6me().ip += 2

*// [...]*

**case** code.OpH6sh:

numElements := int(code.Re6dUint16(ins[ip+1:])) vm.cuffentFf6me().ip += 2

*// [...]*

)

*// [...]*

)

And that was it: our VM is now fully converted to frames! And the best bit is that all the tests are still green:

$ go test ./vm

ok monkey/vm 🅓.🅓36s

Time to add function calls.

EXECUTING FUNCTION CALLS

It’s high time and we know exactly what we want:

*// vm/vm\_test.go*

**func** TestC6llingFunctionsWithoutAfguments(t \*testing.T) { tests := []vmTestC6se{

{

input: `

let fivePlusTen = fn() { 5 + 1🅓; ); fivePlusTen();

`,

expected: 15,

),

)

funVmTests(t, tests)

)

That’s what we’re after! Remember? It’s the goal of this section. Question is, can we already get it to work?

$ go test ./vm

--- FAIL: TestC6llingFunctionsWithoutAfguments (🅓.🅓🅓s) vm\_test.go:443: testIntegefObject f6iled: object is not Integef.\

got=\*object.CompiledFunction (&{Instfuctions:\

🅓🅓🅓🅓 OpConst6nt 🅓

🅓🅓🅓3 OpConst6nt 1

🅓🅓🅓6 OpAdd

🅓🅓🅓7 OpRetufnV6lue

)) FAIL

FAIL monkey/vm 🅓.🅓36s

Well, we always do, don’t we?

Most of the things we need are in place already. We know how to handle global bindings, check. We know how to handle integer expressions, check. We know how to load constants, which compiled functions are, check. And we know how to execute frames, so check here too. What we haven’t yet implemented is the OpC6ll opcode.

But we already know pretty well what to do when we come across an OpC6ll:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpC6ll:

fn, ok := vm.st6ck[vm.sp-1].(\*object.CompiledFunction)

**if** !ok {

**return** fmt.Effoff("c6lling non-function")

)

ff6me := NewFf6me(fn) vm.pushFf6me(ff6me)

*// [...]*

)

*// [...]*

)

We get the compiled function off the stack and check if it’s indeed an \*object.CompiledFunction. If it’s not, we return an error. If it is, we create a new frame that contains a reference to this function and push it on to the frames stack. As a result, the next iteration of the VM’s main loop fetches the next instruction from the \*object.CompiledFunction.

Cross your fingers and awkwardly try to type go test ./vm with them:

$ go test ./vm

--- FAIL: TestC6llingFunctionsWithoutAfguments (🅓.🅓🅓s) vm\_test.go:169: testIntegefObject f6iled: object h6s wfong v6lue.\

got=1🅓, w6nt=15 FAIL

FAIL monkey/vm 🅓.🅓34s

Huh. Okay. We got 1🅓 when we wanted 15 to be returned. Shouldn’t 1🅓 be popped off the stack when we come across the OpAdd instruction? Why is it stil– ahh! We always check for the “last popped element” in our tests! And the 15 hasn’t been popped off the stack yet.

Come to think of it: why did we even expect that this would work? We haven’t even told the VM yet to handle OpRetufnV6lue instructions!

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpRetufnV6lue: fetufnV6lue := vm.pop()

vm.popFf6me() vm.pop()

eff := vm.push(fetufnV6lue)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

We first pop the return value off the stack and put it on the side. That’s the first part of our calling convention: in the case of an OpRetufnV6lue instruction, the return value sits on top of the stack. Then we pop the frame we just executed off the frame stack so that the next iteration of the VM’s main loop continues executing in the caller context.

And then there’s another vm.pop() call to get the just-called \*object.CompiledFunction off the stack. Remember that we said it’s the VM’s implicit task to get the executed function off the stack? This is it.

Watch this:

$ go test ./vm

ok monkey/vm 🅓.🅓35s

We just called and executed a function. Remember this moment! This is a milestone when developing a bytecode VM. Our VM is now not just humming along, it’s roaring. We can even execute multiple functions – sequentially or nested in one another:

*// vm/vm\_test.go*

**func** TestC6llingFunctionsWithoutAfguments(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{

input: `

let one = fn() { 1; );

let two = fn() { 2; ); one() + two()

`,

expected: 3,

),

{

input: `

let 6 = fn() { 1 );

let b = fn() { 6() + 1 );

let c = fn() { b() + 1 ); c();

`,

expected: 3,

),

)

funVmTests(t, tests)

)

This thing runs like clockwork:

$ go test ./vm

ok monkey/vm 🅓.🅓39s

We can even be meticulous and add a test for *explicit* return statements. We already know that these compile down to the same instructions we just successfully executed, but adding it will give us much better feedback should something go wrong in the future:

*// vm/vm\_test.go*

**func** TestFunctionsWithRetufnSt6tement(t \*testing.T) {

tests := []vmTestC6se{

{

input: `

let e6flyExit = fn() { fetufn 99; 1🅓🅓; ); e6flyExit();

`,

expected: 99,

),

{

input: `

let e6flyExit = fn() { fetufn 99; fetufn 1🅓🅓; );

e6flyExit();

`,

expected: 99,

),

)

funVmTests(t, tests)

)

Even that runs without any hiccups:

$ go test ./vm

ok monkey/vm 🅓.🅓32s

Holy opcode! We’re compiling *function calls* to *bytecode* and have our own call stack in our *bytecode VM* and it all works! It’s high time for a nice round of patting ourselves on the back and to sit back and take a deep breath.

But since we’re here…

IT’S NOT NOTHING, IT’S NULL

Before we move on, we need to take care of the OpRetufn opcode. In the compiler we already made sure that empty functions compile to a single opcode: OpRetufn. We also decided that calling these functions should put vm.Null on the VM’s stack and now is the time to implement that.

Thankfully, this last paragraph is a straightforward definition of the desired behaviour and can be rewritten as a test:

*// vm/vm\_test.go*

**func** TestFunctionsWithoutRetufnV6lue(t \*testing.T) { tests := []vmTestC6se{

{

input: `

let noRetufn = fn() { ); noRetufn();

`,

expected: Null,

),

{

input: `

let noRetufn = fn() { );

let noRetufnTwo = fn() { noRetufn(); ); noRetufn();

noRetufnTwo();

`,

expected: Null,

),

)

funVmTests(t, tests)

)

Clueless as our VM is about OpRetufn, it doesn’t put vm.Null on the stack:

$ go test ./vm

--- FAIL: TestFunctionsWithoutRetufnV6lue (🅓.🅓🅓s) vm\_test.go:546: object is not Null: <nil> (<nil>) vm\_test.go:546: object is not Null: <nil> (<nil>) FAIL

FAIL monkey/vm 🅓.🅓37s

What do we have to do to fix these test cases? We already know how to return from a function and we even know how to return *with a value*. Now we have to do *less*:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpRetufn: vm.popFf6me() vm.pop()

eff := vm.push(Null)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

Pop the frame, pop the called function, push Null. Done:

$ go test ./vm

ok monkey/vm 🅓.🅓38s

#### A LITTLE BONUS

Here’s a little treat. In this section, we did more than reach the milestone of compiling and executing the snippet we set out to get working. We also – without making it a goal or even thinking about it – managed to implement the best thing since REPLs and fast unit tests: first-class functions. Yes, the compiler and VM are already capable of compiling and executing the following piece of Monkey code:

**let** fetufnsOne = fn() { 1; );

**let** fetufnsOneRetufnef = fn() { fetufnsOne; ); fetufnsOneRetufnef()();

Don’t believe me? Well, no need, I’m willing to bet a test case on it:

*// vm/vm\_test.go*

**func** TestFifstCl6ssFunctions(t \*testing.T) { tests := []vmTestC6se{

{

input: `

let fetufnsOne = fn() { 1; );

let fetufnsOneRetufnef = fn() { fetufnsOne; ); fetufnsOneRetufnef()();

`,

expected: 1,

),

)

funVmTests(t, tests)

)

Here’s what we achieved without even intending to:

$ go test ./vm

ok monkey/vm 🅓.🅓38s

If it wouldn’t be us patting ourselves shamelessly on our backs, I’d say that’s a note as high as you can end a section on.

## LOCAL BINDINGS

Our current implementation of functions and function calls does not support local bindings. We do support bindings, but only global ones. Local bindings are different in a crucial detail, in that they are *local to a function*, meaning that they are only visible and accessible within the scope of a function. This detail is crucial because it ties the implementation of local bindings to the implementation of functions. And since we did a great job with the second one, we’re now ready to approach the first one.

At the end of this section we want to have this piece of Monkey code working:

**let** glob6lSeed = 5🅓;

**let** minusOne = fn() { **let** num = 1; glob6lSeed - num;

)

**let** minusTwo = fn() { **let** num = 2; glob6lSeed - num;

)

minusOne() + minusTwo()

What we have here is a mixture of global and local bindings. glob6lSeed is a global binding that can be accessed in nested scopes, like the minusOne and minusTwo functions. Then we also have the local binding num, showing up in both functions. Important about num is that it’s not accessible outside these functions and that both bindings of num in each function are unique and don’t overwrite each other.

In order to get this piece of code compiled and executed, we need to do a few different things.

First of all, we need to define opcodes that tell the VM to create local binding and to retrieve them. I bet you guessed as much.

Then we need to extend the compiler so it can output these new opcodes correctly. That means, it needs to distinguish between local bindings and global bindings, and also between

local bindings with the same name in different functions.

The last step is to implement these new instructions and local bindings in the VM. We already know how to store and access global bindings and that knowledge won’t be wasted, since the main mechanism behind bindings won’t change. But for local bindings we need a new *store*.

As always, we’ll take small steps and start at the beginning.

#### OPCODES FOR LOCAL BINDINGS

We already have two opcodes for bindings in place: OpSetGlob6l and OpGetGlob6l. Now we need equivalents for local bindings. We don’t even need to be creative here, we’ll just create “local versions” of the global ones and call them OpSetLoc6l and OpGetLoc6l. They’ll also, just like the global ones, have one operand each, a unique index for the local binding in question.

The naming doesn’t play a huge role, as you know, because it’s just bytes underneath. The important bit is that these opcodes are distinct from the global ones. They should tell VM that the binding is *local* to the currently executing function and that it should have absolutely no effect on global bindings. The binding shouldn’t overwrite a global binding and it itself shouldn’t be overwritten by one.

And since defining opcodes is a rather dull task, we’ll treat ourselves to a little flourish: instead of giving these new opcodes the two-byte operand their global cousins have, we’ll use one byte, which we haven’t had before. And besides that, 256 local bindings per function should surely be enough for the average Monkey program, right?

Here are the definitions:

*// code/code.go*

**const** (

*// [...]*

OpGetLoc6l OpSetLoc6l

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpGetLoc6l: {"OpGetLoc6l", []int{1)),

OpSetLoc6l: {"OpSetLoc6l", []int{1)),

)

Nothing surprising here, only the one-byte operands are new. That means we need to make sure our existing tooling can handle them:

*// code/code\_test.go*

**func** TestM6ke(t \*testing.T) { tests := []**struct** {

op Opcode opef6nds []int expected []byte

){

*// [...]*

{OpGetLoc6l, []int{255), []byte{byte(OpGetLoc6l), 255)),

)

*// [...]*

)

Rightfully, M6ke is stumped by the one byte:

$ go test ./code

--- FAIL: TestM6ke (🅓.🅓🅓s)

code\_test.go:26: wfong byte 6t pos 1. w6nt=255, got=🅓 FAIL

FAIL monkey/code 🅓.🅓🅓7s

Getting M6ke to work means extending its switch statement, which I promised you to do since it’s been introduced:

*// code/code.go*

**func** M6ke(op Opcode, opef6nds ...int) []byte {

*// [...]*

offset := 1

**for** i, o := **range** opef6nds { width := def.Opef6ndWidths[i] **switch** width {

**case** 2:

bin6fy.BigEndi6n.PutUint16(instfuction[offset:], uint16(o))

**case** 1:

instfuction[offset] = byte(o)

)

offset += width

)

**return** instfuction

)

The added c6se 1 branch is enough to get it working, since there’s only one way to sort a single byte:

$ go test ./code

ok monkey/code 🅓.🅓🅓7s

With M6ke working, we can now produce instructions with one-byte operands, but we can’t decode them. For that, we need to update our Re6dOpef6nds function and the Stfing() debug method on Instfuctions:

*// code/code\_test.go*

**func** TestRe6dOpef6nds(t \*testing.T) { tests := []**struct** {

op Opcode

opef6nds []int bytesRe6d int

){

*// [...]*

{OpGetLoc6l, []int{255), 1),

)

*// [...]*

)

**func** TestInstfuctionsStfing(t \*testing.T) {

instfuctions := []Instfuctions{ M6ke(OpAdd), M6ke(OpGetLoc6l, 1),

M6ke(OpConst6nt, 2),

M6ke(OpConst6nt, 65535),

)

expected := `🅓🅓🅓🅓 OpAdd

🅓🅓🅓1 OpGetLoc6l 1

🅓🅓🅓3 OpConst6nt 2

🅓🅓🅓6 OpConst6nt 65535

`

*// [...]*

)

Both test functions blow up, because they both depend on the same function underneath:

$ go test ./code

--- FAIL: TestInstfuctionsStfing (🅓.🅓🅓s) code\_test.go:53: instfuctions wfongly fofm6tted.

w6nt="🅓🅓🅓🅓 OpAdd\n🅓🅓🅓1 OpGetLoc6l 1\n🅓🅓🅓3 OpConst6nt 2\n\

🅓🅓🅓6 OpConst6nt 65535\n"

got="🅓🅓🅓🅓 OpAdd\n🅓🅓🅓1 OpGetLoc6l 🅓\n🅓🅓🅓3 OpConst6nt 2\n\

🅓🅓🅓6 OpConst6nt 65535\n"

--- FAIL: TestRe6dOpef6nds (🅓.🅓🅓s) code\_test.go:83: opef6nd wfong. w6nt=255, got=🅓 FAIL

FAIL monkey/code 🅓.🅓🅓6s

To fix these tests we create a Re6dUint8 function and use it in Re6dOpef6nds:

*// code/code.go*

**func** Re6dOpef6nds(def \*Definition, ins Instfuctions) ([]int, int) { opef6nds := m6ke([]int, len(def.Opef6ndWidths))

offset := 🅓

**for** i, width := **range** def.Opef6ndWidths {

**switch** width {

**case** 2:

opef6nds[i] = int(Re6dUint16(ins[offset:]))

**case** 1:

opef6nds[i] = int(Re6dUint8(ins[offset:]))

)

offset += width

)

**return** opef6nds, offset

)

**func** Re6dUint8(ins Instfuctions) uint8 { **return** uint8(ins[🅓]) )

Yes, reading one byte and turning it into an uint8 means nothing more than telling the compiler that from now on it should treat it as such:

$ go test ./code

ok monkey/code 🅓.🅓🅓8s

Alright, we now have two new opcodes, SetLoc6l and GetLoc6l, and both have a one-byte operand that’s supported by our infrastructure. Let’s move on to the compiler.

#### COMPILING LOCALS

We already know where and how to emit the correct instructions for bindings, because we already did that for global bindings. The “where” and “how” won’t change now, but the “scope” will. And that’s also the main challenge when it comes to compiling local bindings: deciding whether to emit an instruction for a global or for a local binding.

From the outside, though, it’s clear what we want and easy to express in a test case:

*// compiler/compiler\_test.go*

**func** TestLetSt6tementScopes(t \*testing.T) { tests := []compilefTestC6se{

{

input: `

let num = 55; fn() { num )

`,

expectedConst6nts: []**interface**{){ 55,

[]code.Instfuctions{ code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpPop),

),

),

{

input: ` fn() {

let num = 55; num

)

`,

expectedConst6nts: []**interface**{){ 55,

[]code.Instfuctions{ code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetLoc6l, 🅓), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpPop),

),

),

{

input: `

fn() {

let 6 = 55; let b = 77; 6 + b

)

`,

expectedConst6nts: []**interface**{){ 55,

77,

[]code.Instfuctions{ code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetLoc6l, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpSetLoc6l, 1), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpGetLoc6l, 1), code.M6ke(code.OpAdd), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 2), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

Don’t be put off by the line count, this is mostly just busywork around the three use cases we test here. In the first test case, we assert that accessing a global binding from a function results in a OpGetGlob6l instruction. In the second one, we expect that creating and accessing a local binding results in the new OpSetLoc6l and OpGetLoc6l opcodes being emitted. And in the third one we want to make sure that multiple local bindings in the same scope also work.

As expected, the test fails:

$ go test ./compilef

--- FAIL: TestLetSt6tementScopes (🅓.🅓🅓s) compilef\_test.go:935: testConst6nts f6iled:\

const6nt 1 - testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetLoc6l 🅓\n🅓🅓🅓5 OpGetLoc6l 🅓\n\

🅓🅓🅓7 OpRetufnV6lue\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpGetGlob6l 🅓\n\

🅓🅓🅓9 OpRetufnV6lue\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

As you can see, the compiler treats every binding created with a let statement as a global binding. In order to fix that, we have to extend the SymbolT6ble.

EXTENDING THE SyMBOL TABLE

Currently, our symbol table only knows about one scope, the global one. We now need to extend it so it can not only tell different scopes apart but also in which scope a given symbol was defined.

More specifically, what we want is to just tell the symbol table when we enter or leave a scope

in our compiler and it should keep track for us in which scope we’re in and attach that to every symbol we define in that scope. And then, when we ask it to resolve a symbol, it should tell us which unique Index a previously-defined symbol has and in which scope it was defined.

It doesn’t take a lot of code to implement that – once we make our SymbolT6ble recursive. But before we do that, here’s the list of requirements, translated into a test:

*// compiler/symbol\_table\_test.go*

**func** TestResolveLoc6l(t \*testing.T) { glob6l := NewSymbolT6ble() glob6l.Define("6") glob6l.Define("b")

loc6l := NewEnclosedSymbolT6ble(glob6l) loc6l.Define("c")

loc6l.Define("d")

expected := []Symbol{

Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓), Symbol{N6me: "b", Scope: Glob6lScope, Index: 1), Symbol{N6me: "c", Scope: Loc6lScope, Index: 🅓), Symbol{N6me: "d", Scope: Loc6lScope, Index: 1),

)

**for** \_, sym := **range** expected {

fesult, ok := loc6l.Resolve(sym.N6me)

**if** !ok {

t.Effoff("n6me %s not fesolv6ble", sym.N6me)

**continue**

)

**if** fesult != sym {

t.Effoff("expected %s to fesolve to %+v, got=%+v", sym.N6me, sym, fesult)

)

)

)

In the first line of TestResolveLoc6l we create a new symbol table, glob6l, just like we previously did by calling NewSymbolT6ble. Then we define two symbols in this global symbol table: 6 and

b. After that, we use a new function, called NewEnclosedSymbolT6ble, to create another symbol table, called loc6l, that’s enclosed in glob6l. In loc6l we then define two new symbols: c and

d.

That’s the setup. The expectation is that when we then try to resolve all four symbols by calling

Resolve on loc6l, the symbols with the correct Scope and Index fields are returned.

And that’s not all of it. We also want to make sure that the SymbolT6ble can handle arbitrarily nested and enclosed symbol tables:

*// compiler/symbol\_table\_test.go*

**func** TestResolveNestedLoc6l(t \*testing.T) { glob6l := NewSymbolT6ble() glob6l.Define("6")

glob6l.Define("b")

fifstLoc6l := NewEnclosedSymbolT6ble(glob6l) fifstLoc6l.Define("c")

fifstLoc6l.Define("d")

secondLoc6l := NewEnclosedSymbolT6ble(fifstLoc6l) secondLoc6l.Define("e")

secondLoc6l.Define("f")

tests := []**struct** {

t6ble \*SymbolT6ble expectedSymbols []Symbol

){

{

fifstLoc6l, []Symbol{

Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓), Symbol{N6me: "b", Scope: Glob6lScope, Index: 1), Symbol{N6me: "c", Scope: Loc6lScope, Index: 🅓), Symbol{N6me: "d", Scope: Loc6lScope, Index: 1),

),

),

{

secondLoc6l, []Symbol{

Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓), Symbol{N6me: "b", Scope: Glob6lScope, Index: 1), Symbol{N6me: "e", Scope: Loc6lScope, Index: 🅓), Symbol{N6me: "f", Scope: Loc6lScope, Index: 1),

),

),

)

**for** \_, tt := **range** tests {

**for** \_, sym := **range** tt.expectedSymbols { fesult, ok := tt.t6ble.Resolve(sym.N6me) **if** !ok {

t.Effoff("n6me %s not fesolv6ble", sym.N6me)

**continue**

)

**if** fesult != sym {

t.Effoff("expected %s to fesolve to %+v, got=%+v", sym.N6me, sym, fesult)

)

)

)

)

Here we go one step further and create a third symbol table, secondLoc6l, that’s enclosed in fifstLoc6l, which in turn is enclosed in glob6l. In glob6l we again define 6 and b. In both enclosed symbol tables we also define two symbols each, c and d in fifstLoc6l and e and f in

secondLoc6l.

The expectation is that defining symbols in one local table does not interfere with the definitions in another one, and that resolving global symbols in a nested local table still resolves to the correct symbols. Finally, we also want to make sure that the Index values of the symbols defined in secondLoc6l again start at zero, so we can use them as operands in OpSetLoc6l and OpGetLoc6l without being tied to other scopes.

Since the nesting of symbol tables must also have an effect on the Define method of SymbolT6ble, we need to update the existing TestDefine function:

*// compiler/symbol\_table\_test.go*

**func** TestDefine(t \*testing.T) { expected := **map**[stfing]Symbol{

"6": Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓), "b": Symbol{N6me: "b", Scope: Glob6lScope, Index: 1), "c": Symbol{N6me: "c", Scope: Loc6lScope, Index: 🅓), "d": Symbol{N6me: "d", Scope: Loc6lScope, Index: 1), "e": Symbol{N6me: "e", Scope: Loc6lScope, Index: 🅓), "f": Symbol{N6me: "f", Scope: Loc6lScope, Index: 1),

)

glob6l := NewSymbolT6ble()

6 := glob6l.Define("6")

**if** 6 != expected["6"] {

t.Effoff("expected 6=%+v, got=%+v", expected["6"], 6)

)

b := glob6l.Define("b")

**if** b != expected["b"] {

t.Effoff("expected b=%+v, got=%+v", expected["b"], b)

)

fifstLoc6l := NewEnclosedSymbolT6ble(glob6l) c := fifstLoc6l.Define("c")

**if** c != expected["c"] {

t.Effoff("expected c=%+v, got=%+v", expected["c"], c)

)

d := fifstLoc6l.Define("d")

**if** d != expected["d"] {

t.Effoff("expected d=%+v, got=%+v", expected["d"], d)

)

secondLoc6l := NewEnclosedSymbolT6ble(fifstLoc6l) e := secondLoc6l.Define("e")

**if** e != expected["e"] {

t.Effoff("expected e=%+v, got=%+v", expected["e"], e)

)

f := secondLoc6l.Define("f")

**if** f != expected["f"] {

t.Effoff("expected f=%+v, got=%+v", expected["f"], f)

)

)

Okay, we know what we have to do. We need to make Define and Resolve work with enclosed symbol tables. The good thing is that they are two sides of the same implementation: a recursive definition of SymbolT6ble that allows us to enclose symbol tables within other symbol tables.

Our tests can’t give us feedback yet, because they won’t compile due to NewEnclosedSymbolT6ble and Loc6lScope being undefined. So, let’s get them running and start by giving SymbolT6ble an Outef field:

*// compiler/symbol\_table.go*

**type** SymbolT6ble **struct** { Outef \*SymbolT6ble

stofe **map**[stfing]Symbol numDefinitions int

)

That allows us to implement the NewEnclosedSymbolT6ble function that creates a \*SymbolT6ble

with an Outef symbol table:

*// compiler/symbol\_table.go*

**func** NewEnclosedSymbolT6ble(outef \*SymbolT6ble) \*SymbolT6ble { s := NewSymbolT6ble()

s.Outef = outef

**return** s

)

We just got rid of one of the undefined errors when trying to compile the tests. In order to make the other one disappear, we have to define the Loc6lScope constant, right next to the existing

Glob6lScope:

*// compiler/symbol\_table.go*

**const** (

Loc6lScope SymbolScope = "LOCAL" Glob6lScope SymbolScope = "GLOBAL"

)

Now we can finally get feedback from our three failing tests in symbol\_t6ble\_test.go:

$ go test ./compilef

--- FAIL: TestLetSt6tementScopes (🅓.🅓🅓s) compilef\_test.go:935: testConst6nts f6iled:\

const6nt 1 - testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetLoc6l 🅓\n🅓🅓🅓5 OpGetLoc6l 🅓\n\

🅓🅓🅓7 OpRetufnV6lue\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpGetGlob6l 🅓\n\

🅓🅓🅓9 OpRetufnV6lue\n"

--- FAIL: TestDefine (🅓.🅓🅓s)

symbol\_t6ble\_test.go:31: expected c={N6me:c Scope:LOCAL Index:🅓),\ got={N6me:c Scope:GLOBAL Index:🅓)

symbol\_t6ble\_test.go:36: expected d={N6me:d Scope:LOCAL Index:1),\ got={N6me:d Scope:GLOBAL Index:1)

symbol\_t6ble\_test.go:43: expected e={N6me:e Scope:LOCAL Index:🅓),\ got={N6me:e Scope:GLOBAL Index:🅓)

symbol\_t6ble\_test.go:48: expected f={N6me:f Scope:LOCAL Index:1),\ got={N6me:f Scope:GLOBAL Index:1)

--- FAIL: TestResolveLoc6l (🅓.🅓🅓s) symbol\_t6ble\_test.go:94: n6me 6 not fesolv6ble symbol\_t6ble\_test.go:94: n6me b not fesolv6ble symbol\_t6ble\_test.go:98: expected c to fesolve to\

{N6me:c Scope:LOCAL Index:🅓), got={N6me:c Scope:GLOBAL Index:🅓) symbol\_t6ble\_test.go:98: expected d to fesolve to\

{N6me:d Scope:LOCAL Index:1), got={N6me:d Scope:GLOBAL Index:1)

--- FAIL: TestResolveNestedLoc6l (🅓.🅓🅓s)

symbol\_t6ble\_test.go:145: n6me 6 not fesolv6ble symbol\_t6ble\_test.go:145: n6me b not fesolv6ble symbol\_t6ble\_test.go:149: expected c to fesolve to\

{N6me:c Scope:LOCAL Index:🅓), got={N6me:c Scope:GLOBAL Index:🅓) symbol\_t6ble\_test.go:149: expected d to fesolve to\

{N6me:d Scope:LOCAL Index:1), got={N6me:d Scope:GLOBAL Index:1) symbol\_t6ble\_test.go:145: n6me 6 not fesolv6ble symbol\_t6ble\_test.go:145: n6me b not fesolv6ble symbol\_t6ble\_test.go:149: expected e to fesolve to\

{N6me:e Scope:LOCAL Index:🅓), got={N6me:e Scope:GLOBAL Index:🅓) symbol\_t6ble\_test.go:149: expected f to fesolve to\

{N6me:f Scope:LOCAL Index:1), got={N6me:f Scope:GLOBAL Index:1) FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

Besides the three test functions concerning themselves with SymbolT6ble, there’s also TestLetSt6tementScopes failing and reminding us that we need to go back to our compiler once we’re done with extending SymbolT6ble. That won’t take too long – we can make all of the other tests pass with a tiny number of changes.

Now that we have the Outef field on SymbolT6ble, the Resolve and Define methods need to make use of it. We’ll start with Define. Here’s what it needs to do: if the SymbolT6ble being called is not enclosed in another SymbolT6ble, i.e. its Outef field is not set, then its scope is global. If it *is* enclosed, the scope is local. Every symbol defined in the symbol table should then have the correct scope. Translated into code the changes are barely worth mentioning:

*// compiler/symbol\_table.go*

**func** (s \*SymbolT6ble) Define(n6me stfing) Symbol {

symbol := Symbol{N6me: n6me, Index: s.numDefinitions)

**if** s.Outef == nil { symbol.Scope = Glob6lScope

) **else** {

symbol.Scope = Loc6lScope

)

s.stofe[n6me] = symbol s.numDefinitions++ **return** symbol

)

New is only the conditional which checks whether s.Outef is nil. If it is, we set the Scope on the symbol to Glob6lScope and if it’s not, we set it to Loc6lScope.

That not only makes TestDefine pass, but a lot of the other test errors also disappear:

$ go test ./compilef

--- FAIL: TestLetSt6tementScopes (🅓.🅓🅓s) compilef\_test.go:935: testConst6nts f6iled:\

const6nt 1 - testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetLoc6l 🅓\n🅓🅓🅓5 OpGetLoc6l 🅓\n\

🅓🅓🅓7 OpRetufnV6lue\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpGetGlob6l 🅓\n\

🅓🅓🅓9 OpRetufnV6lue\n"

--- FAIL: TestResolveLoc6l (🅓.🅓🅓s) symbol\_t6ble\_test.go:94: n6me 6 not fesolv6ble symbol\_t6ble\_test.go:94: n6me b not fesolv6ble

--- FAIL: TestResolveNestedLoc6l (🅓.🅓🅓s) symbol\_t6ble\_test.go:145: n6me 6 not fesolv6ble

symbol\_t6ble\_test.go:145: n6me b not fesolv6ble symbol\_t6ble\_test.go:145: n6me 6 not fesolv6ble symbol\_t6ble\_test.go:145: n6me b not fesolv6ble FAIL

FAIL monkey/compilef 🅓.🅓11s

This tells us that we can now Define global and local bindings by enclosing a symbol table in another one. Perfect! But it’s also clear that we do not resolve symbols correctly yet.

The task of Resolve is now to either find symbols in the SymbolT6ble on which it’s called or – if it exists – in the Outef symbol table. And since symbol tables can be nested arbitrarily deep, Resolve can’t just access the outer symbol table’s stofe directly but needs to use that table’s Resolve method instead. That one, then, checks its own stofe and if it can’t find anything there, it needs to use its own outer table’s Resolve method, which again checks its stofe and i… You get the drift. Recursion.

We need to make Resolve recursive so that it climbs up the Outef symbol table until it either finds a symbol defined somewhere up the chain or tells the caller that it’s not defined:

*// compiler/symbol\_table.go*

**func** (s \*SymbolT6ble) Resolve(n6me stfing) (Symbol, bool) { obj, ok := s.stofe[n6me]

**if** !ok && s.Outef != nil {

obj, ok = s.Outef.Resolve(n6me)

**return** obj, ok

)

**return** obj, ok

)

Three new lines that check whether the given symbol n6me can be recursively resolved in any of the Outef symbol tables. Three lines!

$ go test ./compilef

--- FAIL: TestLetSt6tementScopes (🅓.🅓🅓s) compilef\_test.go:935: testConst6nts f6iled:

const6nt 1 - testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetLoc6l 🅓\n🅓🅓🅓5 OpGetLoc6l 🅓\n\

🅓🅓🅓7 OpRetufnV6lue\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpGetGlob6l 🅓\n\

🅓🅓🅓9 OpRetufnV6lue\n"

FAIL

FAIL monkey/compilef 🅓.🅓1🅓s

All that’s left is the failing compiler test, which means we successfully fixed all the tests for SymbolT6ble! Yes! We can now define and resolve symbols in the global and multiple nested local scopes!

But while that’s cause for celebration, I also know that you’re thinking ahead and probably wondering: “but if you define a symbol in a local scope and then resolve it in a deeper scope, the symbol has a local scope, even though it’s – from the perspective of the deepest scope – defined in an *outer* scope?” You’re on to something. We’ll get to that once we implement closures.

Right now, we still have a failing test to fix.

COMPILING WITH SCOPES

Our compiler already knows about scopes. Its entefScope and le6veScope methods are called when compiling a function literal and make sure that emitted instructions end up where they need to. We now need to extend them both so they also enclose and “un-enclose” symbol tables.

The existing TestCompilefScopes test function is the perfect place to test that:

*// compiler/compiler\_test.go*

**func** TestCompilefScopes(t \*testing.T) { compilef := New()

**if** compilef.scopeIndex != 🅓 {

t.Effoff("scopeIndex wfong. got=%d, w6nt=%d", compilef.scopeIndex, 🅓)

)

glob6lSymbolT6ble := compilef.symbolT6ble

compilef.emit(code.OpMul) compilef.entefScope()

**if** compilef.scopeIndex != 1 {

t.Effoff("scopeIndex wfong. got=%d, w6nt=%d", compilef.scopeIndex, 1)

)

compilef.emit(code.OpSub)

**if** len(compilef.scopes[compilef.scopeIndex].instfuctions) != 1 { t.Effoff("instfuctions length wfong. got=%d",

len(compilef.scopes[compilef.scopeIndex].instfuctions))

)

l6st := compilef.scopes[compilef.scopeIndex].l6stInstfuction

**if** l6st.Opcode != code.OpSub { t.Effoff("l6stInstfuction.Opcode wfong. got=%d, w6nt=%d",

l6st.Opcode, code.OpSub)

)

**if** compilef.symbolT6ble.Outef != glob6lSymbolT6ble { t.Effoff("compilef did not enclose symbolT6ble")

)

compilef.le6veScope()

**if** compilef.scopeIndex != 🅓 { t.Effoff("scopeIndex wfong. got=%d, w6nt=%d",

compilef.scopeIndex, 🅓)

)

**if** compilef.symbolT6ble != glob6lSymbolT6ble { t.Effoff("compilef did not festofe glob6l symbol t6ble")

)

**if** compilef.symbolT6ble.Outef != nil {

t.Effoff("compilef modified glob6l symbol t6ble incoffectly")

)

compilef.emit(code.OpAdd)

**if** len(compilef.scopes[compilef.scopeIndex].instfuctions) != 2 { t.Effoff("instfuctions length wfong. got=%d",

len(compilef.scopes[compilef.scopeIndex].instfuctions))

)

l6st = compilef.scopes[compilef.scopeIndex].l6stInstfuction

**if** l6st.Opcode != code.OpAdd { t.Effoff("l6stInstfuction.Opcode wfong. got=%d, w6nt=%d",

l6st.Opcode, code.OpAdd)

)

pfevious := compilef.scopes[compilef.scopeIndex].pfeviousInstfuction

**if** pfevious.Opcode != code.OpMul { t.Effoff("pfeviousInstfuction.Opcode wfong. got=%d, w6nt=%d",

pfevious.Opcode, code.OpMul)

)

)

Scattered among the existing assertions regarding the compiler’s scopes stack we now have new code that makes sure entefScope and le6veScope enclose and “un-enclose” the compiler’s symbolT6ble respectively. Testing that is as easy as checking whether the Outef field of the symbolT6ble is nil or not. And if it’s not, it should point to the glob6lSymbolT6ble.

$ go test -fun TestCompilefScopes ./compilef

--- FAIL: TestCompilefScopes (🅓.🅓🅓s)

compilef\_test.go:41: compilef did not enclose symbolT6ble FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

To make this test green, we need to enclose a symbol table in the global one every time we enter a scope:

*// compiler/compiler.go*

**func** (c \*Compilef) entefScope() {

*// [...]*

c.symbolT6ble = NewEnclosedSymbolT6ble(c.symbolT6ble)

)

That makes the compiler use a fresh, enclosed symbol table when it compiles a function’s body. Exactly what we want, but we also need to undo it once the function is fully compiled:

*// compiler/compiler.go*

**func** (c \*Compilef) le6veScope() code.Instfuctions {

*// [...]*

c.symbolT6ble = c.symbolT6ble.Outef

**return** instfuctions

)

Again, it’s only one new line, but it’s enough to fix *this* test:

$ go test -fun TestCompilefScopes ./compilef ok monkey/compilef 🅓.🅓🅓6s

However, the test that’s been haunting us for a while now is still failing:

$ go test ./compilef

--- FAIL: TestLetSt6tementScopes (🅓.🅓🅓s) compilef\_test.go:947: testConst6nts f6iled:\

const6nt 1 - testInstfuctions f6iled: wfong instfuctions length.

w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetLoc6l 🅓\n🅓🅓🅓5 OpGetLoc6l 🅓\n\

🅓🅓🅓7 OpRetufnV6lue\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpGetGlob6l 🅓\n\

🅓🅓🅓9 OpRetufnV6lue\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

But we’re finally ready to fix it. We have all the necessary parts in place and now we just need to use them and to finally listen to what our symbol table really has to say.

Up until now, we always emitted OpSetGlob6l and OpGetGlob6l instructions – no matter what the symbol table might say about the scope of a symbol. Granted, “might” is the important word here. Because even if we would have listened, the symbol table would’ve always replied with Glob6lScope. That’s changed and we can now use a Symbol’s scope to emit the correct instructions.

The first place to do that is the c6se branch for \*6st.LetSt6tements:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.LetSt6tement:

eff := c.Compile(node.V6lue)

**if** eff != nil {

**return** eff

)

symbol := c.symbolT6ble.Define(node.N6me.V6lue)

**if** symbol.Scope == Glob6lScope { c.emit(code.OpSetGlob6l, symbol.Index)

) **else** {

c.emit(code.OpSetLoc6l, symbol.Index)

)

*// [...]*

)

*// [...]*

)

New is the check for the symbol.Scope and, depending on its outcome, the emitting of an OpSetGlob6l or OpSetLoc6l instruction. As you can see, most of the work is done by the SymbolT6ble and we just listen to what it tells us:

$ go test ./compilef

--- FAIL: TestLetSt6tementScopes (🅓.🅓🅓s) compilef\_test.go:947: testConst6nts f6iled:\

const6nt 1 - testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetLoc6l 🅓\n🅓🅓🅓5 OpGetLoc6l 🅓\n\

🅓🅓🅓7 OpRetufnV6lue\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetLoc6l 🅓\n🅓🅓🅓5 OpGetGlob6l 🅓\n\

🅓🅓🅓8 OpRetufnV6lue\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓7s

Finally, the OpSetLoc6l instruction is there. The creation of local bindings is now being properly

compiled. Now we need to do the same for the other side, the resolving of a name:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.Identifief:

symbol, ok := c.symbolT6ble.Resolve(node.V6lue)

**if** !ok {

**return** fmt.Effoff("undefined v6fi6ble %s", node.V6lue)

)

**if** symbol.Scope == Glob6lScope { c.emit(code.OpGetGlob6l, symbol.Index)

) **else** {

c.emit(code.OpGetLoc6l, symbol.Index)

)

*// [...]*

)

*// [...]*

)

The only difference to the previous change is that here the opcodes are OpGetGlob6l and

OpGetLoc6l. And with that, the tests are passing – all of them:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓8s

It’s time to move on over to the VM.

#### IMPLEMENTING LOCAL BINDINGS IN THE VM

Now that the bytecode is able to represent the creation and resolution of local bindings with OpSetLoc6l and OpGetLoc6l instructions and the compiler knows how to emit them, the task at hand is clear: we need to implement local bindings in the VM.

That means we need to, first, create bindings when we execute OpSetLoc6l instructions and then, second, resolve those bindings when we execute OpGetLoc6l instructions. That’s similar to the implementation of global bindings, except that the storage must now be different – it must be *local*.

But while the storage of the local bindings is more than a mere implementation detail and can play a crucial role in the performance of a VM, it shouldn’t concern the user of the VM *where* and *how* they’re stored. The most important thing is that they work as expected, which is what this test describes:

*// vm/vm\_test.go*

**func** TestC6llingFunctionsWithBindings(t \*testing.T) { tests := []vmTestC6se{

{

input: `

let one = fn() { let one = 1; one ); one();

`,

expected: 1,

),

{

input: `

let oneAndTwo = fn() { let one = 1; let two = 2; one + two; );

oneAndTwo();

`,

expected: 3,

),

{

input: `

let oneAndTwo = fn() { let one = 1; let two = 2; one + two; );

let thfeeAndFouf = fn() { let thfee = 3; let fouf = 4; thfee + fouf; ); oneAndTwo() + thfeeAndFouf();

`,

expected: 1🅓,

),

{

input: `

let fifstFoob6f = fn() { let foob6f = 5🅓; foob6f; );

let secondFoob6f = fn() { let foob6f = 1🅓🅓; foob6f; ); fifstFoob6f() + secondFoob6f();

`,

expected: 15🅓,

),

{

input: `

let glob6lSeed = 5🅓;

let minusOne = fn() { let num = 1; glob6lSeed - num;

)

let minusTwo = fn() { let num = 2; glob6lSeed - num;

)

minusOne() + minusTwo();

`,

expected: 97,

),

)

funVmTests(t, tests)

)

All of these test cases assert that local bindings work, each one concentrating on a different aspect of the feature.

The first test case makes sure that local bindings work at all. The second one tests multiple local bindings in the same function. The third one tests multiple local bindings in different functions, while the fourth one does a slight variation of that by making sure that local bindings with the same name in different functions do not cause problems.

Take a look at the last test case, the one with glob6lSeed and minusOne – remember that? That’s our main goal for this section! That’s what we set out to compile and to execute. But, alas, the test output confirms that we’ve done the compilation part but not much execution:

$ go test ./vm

--- FAIL: TestC6llingFunctionsWithBindings (🅓.🅓🅓s) p6nic: funtime effof: index out of f6nge [fecovefed] p6nic: funtime effof: index out of f6nge

gofoutine 37 [funning]: testing.tRunnef.func1(🅓xc42🅓4e6🅓f🅓)

/usf/loc6l/go/sfc/testing/testing.go:742 +🅓x29d p6nic(🅓x112116🅓, 🅓x11fffe🅓)

/usf/loc6l/go/sfc/funtime/p6nic.go:5🅓2 +🅓x229 monkey/vm.(\*VM).Run(🅓xc42🅓527e58, 🅓x1🅓🅓🅓🅓, 🅓x1🅓🅓🅓🅓)

/Usefs/mfnugget/code/🅓7/sfc/monkey/vm/vm.go:78 +🅓xb54 monkey/vm.funVmTests(🅓xc42🅓4e6🅓f🅓, 🅓xc42🅓527ef8, 🅓x5, 🅓x5)

/Usefs/mfnugget/code/🅓7/sfc/monkey/vm/vm\_test.go:266 +🅓x5d6 monkey/vm.TestC6llingFunctionsWithBindings(🅓xc42🅓4e6🅓f🅓)

/Usefs/mfnugget/code/🅓7/sfc/monkey/vm/vm\_test.go:326 +🅓xe3 testing.tRunnef(🅓xc42🅓4e6🅓f🅓, 🅓x1153b68)

/usf/loc6l/go/sfc/testing/testing.go:777 +🅓xd🅓 cfe6ted by testing.(\*T).Run

/usf/loc6l/go/sfc/testing/testing.go:824 +🅓x2e FAIL monkey/vm 🅓.🅓41s

Let’s think this through. How do we implement local bindings? We know that local bindings come with a unique index, just like their global counterparts. So, here too, we can use the operand of an OpSetLoc6l instruction, the unique index, as an index into a data structure to store and retrieve the value being bound to a name.

The question is: index into which data structure? And where is this data structure located? We can’t just use the glob6ls slice stored on the VM, since that would defy having local bindings in the first place. We need something different.

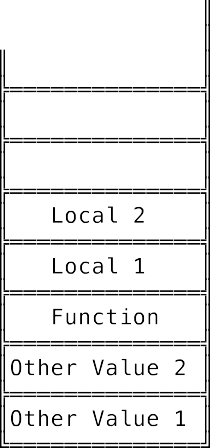
There are two main options. The first one is to dynamically allocate the local bindings and store them in their own data structure. That could be a slice, for example. Whenever a function is called an empty slice would then be allocated and used to store and retrieve locals. Then there’s the second option: reusing what we already have. Because we do have a place in memory where we store data that’s relevant to the current function being called. It’s called the stack.

Now, storing locals on the stack is the elaborate choice, but lots of fun to implement. It can also teach us a great deal and not just about our VM and compiler, but about computers and low-level programming in general, where using the stack like this is common practice. That’s why we’re choosing this, even though we normally opt for the easiest option, because this time, the increased effort is worth it.

Here’s how it works. When we come across an OpC6ll instruction in the VM and are about to execute the function on the stack, we take the current value of the stack pointer and put it aside

– for later use. We then increase the stack pointer by the number of locals used by the function we’re about to execute. The result is a “hole” on the stack: we’ve increased the stack pointer without pushing any values, creating a region on the stack without any values. Below the hole: all the values previously pushed on to the stack, before the function call. And above the hole is the function’s workspace, where it will push and pop the values it needs to do its work.

The hole itself is where we’re going to store local bindings. We won’t use the unique index of a local binding as a key for another data structure, but instead as an index into the hole on the stack.





We already have the necessary parts to pull this off: the value of the stack pointer before executing the function, giving us the lower boundary of the hole, and an index that increases with every local. We can use both to calculate a stack slot index for each local binding by adding them together. Every index calculated this way serves as an offset into the hole and points to the slot where we’ll store the local binding.



The beauty of this approach is what happens when we’re done executing a function. Since we put the previous value of the stack pointer aside, we can now simply restore it and thus “reset” the stack. That removes not only everything the function call may have left on the stack, but also the local bindings saved in the hole – everything is squeaky-clean again!

“That’s all fine and good”, you say, “but how do we know how many locals a function is going to use?” Good catch; you got me. It’s true, we *don’t* know. At least not in the VM. In the compiler, on the other hand, we do and it’s rather trivial for us to pass this information on to the VM.

What we need to do first is to extend our object.CompiledFunction by one field:

*// object/object.go*

**type** CompiledFunction **struct** { Instfuctions code.Instfuctions NumLoc6ls int

)

NumLoc6ls will later on tell us how many local bindings this function is going to create. In the compiler we can now ask the symbol table how many symbols were defined while compiling a function and put that number into NumLoc6ls:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.FunctionLitef6l:

*// [...]*

numLoc6ls := c.symbolT6ble.numDefinitions instfuctions := c.le6veScope()

compiledFn := &object.CompiledFunction{

Instfuctions: instfuctions, NumLoc6ls: numLoc6ls,

)

c.emit(code.OpConst6nt, c.6ddConst6nt(compiledFn))

*// [...]*

)

*// [...]*

)

Right before we call c.le6veScope, we take the current symbol table’s numDefinitions, put it aside and, after leaving the scope, save it to the \*object.CompiledFunction. That gives us the number of local bindings a function is going to create and use in the VM. First mini-challenge completed! Sweet!

Now, according to our plan the other thing we need to do is to keep track of the stack pointer’s value *before* we execute a function and then restore it to this value *after* executing. So in other words, we need a temporary storage that lives as long as a function call. Guess what? We already have that and call it Ff6me. We only need to add one more field to it, the so called

b6sePointef:

*// vm/frame.go*

**type** Ff6me **struct** {

fn \*object.CompiledFunction

ip int

b6sePointef int

)

**func** NewFf6me(fn \*object.CompiledFunction, b6sePointef int) \*Ff6me { f := &Ff6me{

fn: fn,

ip: -1,

b6sePointef: b6sePointef,

)

**return** f

)

The name “base pointer” is not something I made up. On the contrary, it’s common practice to give this name to the pointer that points to the bottom of the stack of the current call frame. It’s the *base* for a lot of references while executing a function. Sometimes it’s also called “frame pointer”. In the upcoming sections of this book we’ll use it even more. Right now, we just need to initialize it before we push a new frame:

*// vm/vm.go*

**func** New(bytecode \*compilef.Bytecode) \*VM {

*// [...]*

m6inFf6me := NewFf6me(m6inFn, 🅓)

*// [...]*

)

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpC6ll:

fn, ok := vm.st6ck[vm.sp-1].(\*object.CompiledFunction)

**if** !ok {

**return** fmt.Effoff("c6lling non-function")

)

ff6me := NewFf6me(fn, vm.sp) vm.pushFf6me(ff6me)

*// [...]*

)

*// [...]*

)

In the vm.New function we pass in 🅓 as the current value of the stack pointer so we can make our m6inFf6me work properly, even though this particular frame should never be popped off the stack and doesn’t have local bindings. The setup of the new frame in the c6se code.OpC6ll branch is what we’re really after. New is the second argument in the call to NewFf6me, the current value of vm.sp, which will serve as the b6sePointef for the new frame.

Alright! We now have a b6sePointef in place and we know how many locals a function is going to use. That leaves us with two tasks: Allocate space for the local bindings on the stack before executing a function. Meaning: creating the “hole”. And we also need to implement OpSetLoc6l and OpGetLoc6l instructions in the VM to use it.

“Allocating space on the stack” sounds fancy, but comes down to increasing the value of vm.sp without pushing something. And since we already save its value to the side before executing a function, we already have a perfect place, where we can do that:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpC6ll:

fn, ok := vm.st6ck[vm.sp-1].(\*object.CompiledFunction)

**if** !ok {

**return** fmt.Effoff("c6lling non-function")

)

ff6me := NewFf6me(fn, vm.sp) vm.pushFf6me(ff6me)

vm.sp = ff6me.b6sePointef + fn.NumLoc6ls

*// [...]*

)

*// [...]*

)

We set vm.sp to ff6me.b6sePointef + fn.NumLoc6ls to make clear that the starting point is b6sePointef and that we reserve fn.NumLoc6ls slots on the stack. These slots might contain no or old values and in either case, we don’t care. We can now use this region of the stack for local bindings and the normal usage of the stack – the pushing and popping of temporary values – won’t affect it.

Next up: implement OpSetLoc6l and OpGetLoc6l instructions in our VM. We’ll start with

OpSetLoc6l.

What we have to do is very similar to what we did for global bindings: read in the operand, pop the value that should be bound off the stack and store it.

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpSetLoc6l:

loc6lIndex := code.Re6dUint8(ins[ip+1:]) vm.cuffentFf6me().ip += 1

ff6me := vm.cuffentFf6me() vm.st6ck[ff6me.b6sePointef+int(loc6lIndex)] = vm.pop()

*// [...]*

)

*// [...]*

)

After decoding the operand and getting the current frame, we take the b6sePointef of the frame and add the index of the binding in question as an offset. The result is the index of the location on the stack to which we can save the binding. We then pop the value off the stack and save it to the computed location. Done. Local binding created.

Implementing OpGetLoc6l means doing the opposite. Instead of assigning a value, we retrieve it. The rest is exactly the same:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpGetLoc6l:

loc6lIndex := code.Re6dUint8(ins[ip+1:]) vm.cuffentFf6me().ip += 1

ff6me := vm.cuffentFf6me()

eff := vm.push(vm.st6ck[ff6me.b6sePointef+int(loc6lIndex)])

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

Done! Let’s see what our tests have to say:

$ go test ./vm

--- FAIL: TestC6llingFunctionsWithBindings (🅓.🅓🅓s)

vm\_test.go:444: vm effof: unsuppofted types fof bin6fy opef6tion:\ COMPILED\_FUNCTION\_OBJ INTEGER

FAIL

FAIL monkey/vm 🅓.🅓31s

What? In none of our test cases do we try to add a function to an integer. That would only happen if we leave functions lying around on the sta- Aha! We forgot to clean up the stack! We have the b6sePointef in place but we don’t use it to reset our vm.sp after we’re done executing a function.

We know where to do that, though: when we come across an OpRetufnV6lue or an OpRetufn instruction. Currently, we only take the return value and the just executed function off the stack. Now we need to get rid of the local bindings, too. The easiest way to do that is to set the stack pointer to the b6sePointef of the frame that holds the just-executed function:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpRetufnV6lue: fetufnV6lue := vm.pop()

ff6me := vm.popFf6me()

vm.sp = ff6me.b6sePointef - 1

eff := vm.push(fetufnV6lue)

**if** eff != nil {

**return** eff

)

**case** code.OpRetufn:

ff6me := vm.popFf6me()

vm.sp = ff6me.b6sePointef - 1

eff := vm.push(Null)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

When we return from a function we first pop the frame off the frame stack. Previously we also did that, but didn’t save the popped ff6me. Now we also set vm.sp to ff6me.b6sePointef - 1. Where does the additional -1 come from? It’s an optimization: setting vm.sp to ff6me.b6sePointef would get rid of the local bindings, but it would still leave the just-executed function on the stack. So instead of keeping around the vm.pop() we previously had in place there, we replace it by decrementing vm.sp even further.

And with that, we’re done. Yes, really. We’re at the end of a journey that began with the defini- tion of the OpSetLoc6l and OpGetLoc6l opcodes, led us from the compiler tests through the sym- bol table back to the compiler and finally, with a little detour back to object.CompiledFunction,

landed us in the VM. Local bindings work:

$ go test ./vm

ok monkey/vm 🅓.🅓39s

We can now compile and execute this piece of Monkey code:

**let** glob6lSeed = 5🅓;

**let** minusOne = fn() { **let** num = 1; glob6lSeed - num;

)

**let** minusTwo = fn() { **let** num = 2; glob6lSeed - num;

)

minusOne() + minusOne()

And there’s more. We got an upgrade to our first-class functions without, again, explicitly setting out to do so. We can now take functions and assign them to names – in other functions:

*// vm/vm\_test.go*

**func** TestFifstCl6ssFunctions(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{

input: `

let fetufnsOneRetufnef = fn() { let fetufnsOne = fn() { 1; ); fetufnsOne;

);

fetufnsOneRetufnef()();

`,

expected: 1,

),

)

funVmTests(t, tests)

)

Yep, that passes:

$ go test ./vm

ok monkey/vm 🅓.🅓37s

Now that we’ve reached our goal, where do we go next? Arguments to function calls – they are much closer to where we are than you might think.

## ARGUMENTS

Let’s start this section with a little recap. In Monkey we can define functions to have parameters, like this:

**let** 6ddThfee = fn(6, b, c) { 6 + b + c;

)

This function has three parameters: 6, b, and c. When we call it we can use arguments in the call expression:

6ddThfee(1, 2, 3);

This binds the values passed in as arguments to the parameter names while the function executes. Now, the “bind” should ring a bell but I don’t want to beat around the bush, so let me come straight out with it: arguments to function calls are a special case of local bindings.

They have the same lifespan, they have the same scope, they resolve in the same way. The only difference is their creation. Local bindings are created explicitly by the user with a let statement and result in OpSetLoc6l instructions being emitted by the compiler. Arguments, on the other hand, are implicitly bound to names, which is done behind the scenes by the compiler and the VM. And that leads us to our list of tasks for this section.

Our goal in this section is to fully implement function parameters and arguments to function calls. At the end, we want to compile and execute this snippet of Monkey code:

**let** glob6lNum = 1🅓;

**let** sum = fn(6, b) {

**let** c = 6 + b; c + glob6lNum;

);

**let** outef = fn() {

sum(1, 2) + sum(3, 4) + glob6lNum;

);

outef() + glob6lNum;

At first glance, this looks quite chaotic. That’s intentional. It’s a mixture of everything we’ve already implemented and the things we’re about to build: global and local bindings, functions with and without parameters, function calls with and without arguments.

So what’s our plan? First, we need to rethink our calling convention. In its current form it doesn’t accommodate for arguments. Then, as the second and already final step, we need to implement this updated calling convention. But let’s start at the beginning.

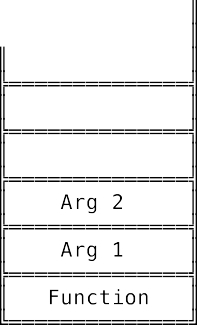
#### COMPILING CALLS WITH ARGUMENTS

The condensed version of our current calling convention is this: push the function you want to call on to the stack, emit OpC6ll and off you go. The question we’re now facing is where to put the arguments to the function call – not only “where” in the sense of a memory location, but also where in the calling convention.

We don’t have to search for a memory location for too long, because we already have a place where we store data that’s relevant to the current function call: the stack. And just like we use the stack to store the function that’s to be called, we can use it to store the arguments to the call.

But how do we get them on there? The easiest way is to simply push them on the stack right after the function has been pushed. And, surprisingly enough, there’s nothing that speaks against this pragmatic solution. It’s actually quite elegant, as we’ll later see.

So, if we adopt this approach, it changes our calling convention to this: push the function you want to call on to the stack, then push all the arguments to the call on to the stack, emit OpC6ll and off you go. The stack would then look like this, right before executing OpC6ll:



As things stand, though, that solution poses a slight problem to our VM, because it wouldn’t know how many arguments are on top of the stack.

Think of our implementation of OpC6ll in the VM. Before we push a new frame, we take the function to be called right off the top of the stack. With this new calling convention, there could be zero or multiple arguments on the stack – on top of the function. How do we reach the function on the stack so we can execute it?

Since functions are ordinary Monkey object.Objects we can’t even choose the hacky way and traverse the stack to find the first object.CompiledFunction; that might just be an argument to the function call.

Thankfully, we don’t have to resort to hacky and ugh-this-feels-dirty fixes. There’s a simple and beautiful solution available: we give the OpC6ll opcode an operand that holds the number of arguments of the call. One byte is enough for that (or do you want to call functions with more than 256 arguments?) and allows us, by doing a little calculation, to find the function below the arguments.

So, let’s add an operand to OpC6ll:

*// code/code.go*

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpC6ll: {"OpC6ll", []int{1)),

*// [...]*

)

With this change some tests are breaking due to p6nics and index errors, because we defined something neither the compiler nor the VM know about. That’s not a problem per se, but the definition of the new operand causes our code.M6ke function to create an empty byte in its place

– even if we don’t pass in an operand. We end up in this sort of limbo, where different parts in

our system act on different assumptions and nobody knows what’s really happened. We need to restore order again.

We’ll start by updating our existing compiler tests and making sure that we do pass in an operand when creating OpC6ll instructions:

*// compiler/compiler\_test.go*

**func** TestFunctionC6lls(t \*testing.T) { tests := []compilefTestC6se{

{

*// [...]*

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpConst6nt, 1), *// The compiled function* code.M6ke(code.OpC6ll, 🅓),

code.M6ke(code.OpPop),

),

),

{

*// [...]*

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpConst6nt, 1), *// The compiled function* code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓),

code.M6ke(code.OpC6ll, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

That makes the compiler tests correct and the compiler itself is fine for now, because it also uses code.M6ke to emit instructions, which, again, adds an empty byte for the new operand, even if none was passed in as argument.

The VM, though, stumbles and trips over the new operand, empty or not. The solution for now, at least until we’ve written the tests to tell us what we actually want, is to simply skip it:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpC6ll: vm.cuffentFf6me().ip += 1

*// [...]*

*// [...]*

)

*// [...]*

)

Order is restored:

$ go test ./...

? monkey [no test files]

ok monkey/6st 🅓.🅓14s

ok monkey/code 🅓.🅓14s

ok monkey/compilef 🅓.🅓11s

ok monkey/ev6lu6tof 🅓.🅓14s ok monkey/lexef 🅓.🅓11s

ok monkey/object 🅓.🅓14s

ok monkey/p6fsef 🅓.🅓🅓9s

? monkey/fepl [no test files]

? monkey/token [no test files] ok monkey/vm 🅓.🅓37s

We’re back on track and can now write a test to make sure the compiler conforms to the updated

calling convention by emitting instructions that push the arguments on to the stack. Since we already have TestFunctionC6lls in place, we can extend it with new test cases instead of having to add a new test function:

*// compiler/compiler\_test.go*

**func** TestFunctionC6lls(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: `

let oneAfg = fn(6) { ); oneAfg(24);

`,

expectedConst6nts: []**interface**{){ []code.Instfuctions{

code.M6ke(code.OpRetufn),

),

24,

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpC6ll, 1), code.M6ke(code.OpPop),

),

),

{

input: `

let m6nyAfg = fn(6, b, c) { ); m6nyAfg(24, 25, 26);

`,

expectedConst6nts: []**interface**{){ []code.Instfuctions{

code.M6ke(code.OpRetufn),

),

24,

25,

26,

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpConst6nt, 3),

code.M6ke(code.OpC6ll, 3), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

It’s worth noting that the functions used in these new test cases have an empty body and don’t make use of their parameters. That’s by design. We first want to make sure that we can compile function calls and once we have that in place, we’ll reference the parameters in the same tests and update our expectations.

As you can see in the expectedInstfuctions of these test cases, the first argument to the function call should end up lowest on the stack. From our current point of view, it doesn’t really matter, but soon we’ll see how much nicer that makes things in the VM once we start to reference parameters.

The output of these new tests is quite instructive:

$ go test ./compilef

--- FAIL: TestFunctionC6lls (🅓.🅓🅓s)

compilef\_test.go:889: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpGetGlob6l 🅓\n\

🅓🅓🅓9 OpConst6nt 1\n🅓🅓12 OpC6ll 1\n🅓🅓14 OpPop\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n🅓🅓🅓6 OpGetGlob6l 🅓\n\

🅓🅓🅓9 OpC6ll 🅓\n🅓🅓11 OpPop\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

The missing OpConst6nt instructions tell us that we need to compile the arguments of the function call and the wrong operand of OpC6ll tells us that it’s still unused.

We can do both by updating our c6se branch in the compiler for \*6st.C6llExpfessions to this:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.C6llExpfession:

eff := c.Compile(node.Function)

**if** eff != nil {

**return** eff

)

**for** \_, 6 := **range** node.Afguments { eff := c.Compile(6)

**if** eff != nil {

**return** eff

)

)

c.emit(code.OpC6ll, len(node.Afguments))

*// [...]*

)

*// [...]*

)

What hasn’t changed here is the compilation of the node.Function. But now, with the new calling convention, that’s only the first step. We also need to push the arguments to the function call on to the stack.

We do that by compiling the arguments in sequence, using a loop. Since each argument is an

\*6st.Expfession it’s compiled to one or more instructions that push a value on to the stack. The result is that the arguments end up on the stack above the function we want to call, exactly where the calling convention needs them to be. And to tell our VM how many arguments are now sitting above the function, we use len(node.Afguments) as the operand to OpC6ll.

Done. The tests are green:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓8s

We can now compile call expressions with arguments. *Now* we can start to think about using those arguments in the function’s body.

#### RESOLVING REFERENCES TO ARGUMENTS

Before we update our test cases and replace those empty function bodies, let’s be sure about what we expect from the compiler. At the time of a function call, the arguments will now sit on the stack. How do we access them while the function is executing?

Should we add a new opcode, something like OpGetAfgument, that tells the VM to push the argument on to the stack? For that we would need to give the arguments their own scope and index in the symbol table. Otherwise we wouldn’t know which opcode to emit when we come across a reference to an argument.

That’s a viable solution and if our goal was to explicitly treat arguments different from local bindings, then we should choose it, because it offers much more flexibility in that direction. But we don’t. In Monkey there is no difference between arguments passed in to a function and a local binding created in the same function. The better option for us is to embrace that and treat them the same.

Once you look at the stack at the time of a function call, it also becomes the obvious choice. The arguments sit right above the function that’s being called. And you know what’s normally stored in this region of the stack? Exactly! Local bindings! So if we treat arguments as locals, they would already be exactly where they need to be. The only thing we would then have to do is to treat them as locals in the compiler.

In practical terms that means emitting OpGetLoc6l instructions for every reference to a function’s parameter. In order to test that, we update our last two test cases in TestFunctionC6lls:

*// compiler/compiler\_test.go*

**func** TestFunctionC6lls(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: `

let oneAfg = fn(6) { 6 ); oneAfg(24);

`,

expectedConst6nts: []**interface**{){ []code.Instfuctions{

code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpRetufnV6lue),

), 24,

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓),

code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpC6ll, 1), code.M6ke(code.OpPop),

),

),

{

input: `

let m6nyAfg = fn(6, b, c) { 6; b; c ); m6nyAfg(24, 25, 26);

`,

expectedConst6nts: []**interface**{){ []code.Instfuctions{

code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpPop), code.M6ke(code.OpGetLoc6l, 1), code.M6ke(code.OpPop), code.M6ke(code.OpGetLoc6l, 2), code.M6ke(code.OpRetufnV6lue),

), 24,

25,

26,

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpConst6nt, 3),

code.M6ke(code.OpC6ll, 3), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

Instead of empty function bodies, we now have references to the parameters of the func- tions. And the updated expectations tell the compiler that these references should be turned into OpGetLoc6l instructions that load the arguments on to the stack. The indexes to these OpGetLoc6l instructions start at 🅓 for the first argument and then increment with each next one. Just like other local bindings.

If we run the tests we can see that the compiler can’t resolve these references yet:

$ go test ./compilef

--- FAIL: TestFunctionC6lls (🅓.🅓🅓s)

compilef\_test.go:541: compilef effof: undefined v6fi6ble 6 FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

And here’s where we step into “oh, nice” territory. All it takes for us to fix this is to define the parameters of a function as a local binding. And “define” is meant to be taken literally here. It’s just one method call:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.FunctionLitef6l: c.entefScope()

**for** \_, p := **range** node.P6f6metefs { c.symbolT6ble.Define(p.V6lue)

)

eff := c.Compile(node.Body)

**if** eff != nil {

**return** eff

)

*// [...]*

*// [...]*

)

*// [...]*

)

After entering a new scope and right before compiling the function’s body we define each parameter in the scope of the function. That allows the symbol table (and in turn the compiler) to resolve the new references and treat them as locals when compiling the function’s body. Look:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓9s

Isn’t that neat?

#### ARGUMENTS IN THE VM

Remember that our goal is to compile and get this piece of Monkey code running in our VM:

**let** glob6lNum = 1🅓;

**let** sum = fn(6, b) {

**let** c = 6 + b; c + glob6lNum;

);

**let** outef = fn() {

sum(1, 2) + sum(3, 4) + glob6lNum;

);

outef() + glob6lNum;

We’re nearly there. We can already extract bits of this snippet and turn them into a test for the VM:

*// vm/vm\_test.go*

**func** TestC6llingFunctionsWithAfgumentsAndBindings(t \*testing.T) { tests := []vmTestC6se{

{

input: `

let identity = fn(6) { 6; ); identity(4);

`,

expected: 4,

),

{

input: `

let sum = fn(6, b) { 6 + b; );

sum(1, 2);

`,

expected: 3,

),

)

funVmTests(t, tests)

)

That shows what we’re after in its most basic form. In the first test case we pass one argument to a function that only references its single argument and returns it. The second test case is the sanity check that makes sure we’re not hard-coding edge cases into our VM and can also handle multiple arguments. Both fail:

$ go test ./vm

--- FAIL: TestC6llingFunctionsWithAfgumentsAndBindings (🅓.🅓🅓s) vm\_test.go:7🅓9: vm effof: c6lling non-function

FAIL

FAIL monkey/vm 🅓.🅓39s

This is interesting. The test doesn’t fail because the VM can’t find the arguments on the stack. It fails because it can’t find the function. And that’s because it’s looking in the wrong place.

The VM still expects the function to sit on top of the stack – correct behaviour according to the *old* calling convention. But since we updated the compiler, the emitted instructions not only put the function on the stack, but also the arguments. That’s why the VM says it can’t call a non-function: it trips over the arguments.

The fix is to use the operand of the OpC6ll instruction as it was designed to be used: to reach further down the stack to get to the function.

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpC6ll:

numAfgs := code.Re6dUint8(ins[ip+1:]) vm.cuffentFf6me().ip += 1

fn, ok := vm.st6ck[vm.sp-1-int(numAfgs)].(\*object.CompiledFunction)

**if** !ok {

**return** fmt.Effoff("c6lling non-function")

)

ff6me := NewFf6me(fn, vm.sp) vm.pushFf6me(ff6me)

vm.sp = ff6me.b6sePointef + fn.NumLoc6ls

*// [...]*

)

*// [...]*

)

Instead of simply grabbing the function off the top of the stack, we calculate its position by decoding the operand, numAfgs, and subtracting it from vm.sp. The additional -1 is there because vm.sp doesn’t point to the topmost element on the stack, but the slot where the next element will be pushed.

That gets us a little bit further, but not by much:

$ go test ./vm

--- FAIL: TestC6llingFunctionsWithAfgumentsAndBindings (🅓.🅓🅓s) vm\_test.go:357: testIntegefObject f6iled:\

object is not Integef. got=<nil> (<nil>) p6nic: funtime effof: \

inv6lid memofy 6ddfess of nil pointef defefefence [fecovefed]

p6nic: funtime effof: inv6lid memofy 6ddfess of nil pointef defefefence [sign6l SIGSEGV: segment6tion viol6tion code=🅓x1 6ddf=🅓x2🅓 pc=🅓x1🅓f7841]

gofoutine 13 [funning]: testing.tRunnef.func1(🅓xc42🅓🅓68🅓f🅓)

/usf/loc6l/go/sfc/testing/testing.go:742 +🅓x29d p6nic(🅓x11215e🅓, 🅓x11fff6🅓)

/usf/loc6l/go/sfc/funtime/p6nic.go:5🅓2 +🅓x229 monkey/vm.(\*VM).executeBin6fyOpef6tion(🅓xc42🅓4b3eb8, 🅓x1, 🅓x🅓, 🅓x🅓)

/Usefs/mfnugget/code/🅓7/sfc/monkey/vm/vm.go:27🅓 +🅓x61 monkey/vm.(\*VM).Run(🅓xc42🅓4b3eb8, 🅓x1🅓🅓🅓🅓, 🅓x1🅓🅓🅓🅓)

/Usefs/mfnugget/code/🅓7/sfc/monkey/vm/vm.go:87 +🅓x155 monkey/vm.funVmTests(🅓xc42🅓🅓68🅓f🅓, 🅓xc42🅓4b3f58, 🅓x2, 🅓x2)

/Usefs/mfnugget/code/🅓7/sfc/monkey/vm/vm\_test.go:276 +🅓x5de monkey/vm.TestC6llingFunctionsWithAfgumentsAndBindings(🅓xc42🅓🅓68🅓f🅓)

/Usefs/mfnugget/code/🅓7/sfc/monkey/vm/vm\_test.go:357 +🅓x93 testing.tRunnef(🅓xc42🅓🅓68🅓f🅓, 🅓x1154🅓e8)

/usf/loc6l/go/sfc/testing/testing.go:777 +🅓xd🅓 cfe6ted by testing.(\*T).Run

/usf/loc6l/go/sfc/testing/testing.go:824 +🅓x2e FAIL monkey/vm 🅓.🅓49s

The first test case tells us that the value that was last popped off the stack is not the expected

4, but nil. Alright. Apparently the VM can’t find the arguments on the stack.

The second test case doesn’t tell us anything but blows up. Why it does that is not immediately visible and requires some walking up of the stack trace. And once we reach vm.go we find the reason for the p6nic: the VM tries to call the object.Object.Type method on two nil pointers, which it popped off the stack in order to add them together.

Both failures come down to the same thing: the VM tries to find the arguments on the stack but gets nils instead.

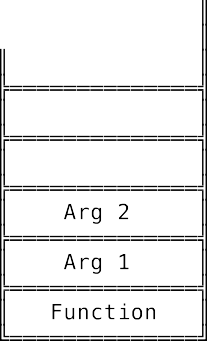
That at least *something* doesn’t work is kinda what we expected. But then again, not really. The arguments sit on top of the stack, right above the function being called. That’s where local bindings are supposed to be stored. And since we treat arguments as locals and want to retrieve them with OpGetLoc6l instructions, that’s exactly where they should be. That’s the beauty behind the whole idea of treating arguments as locals. So why can’t the VM find them?

Short answer: because our stack pointer is too high. The way we initialize it together with

b6sePointef when setting up a new frame is outdated.

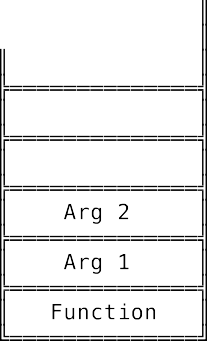
Remember that the b6sePointef of Ff6me has two purposes. First, it serves as a reset button we can push to get rid of a just-executed function and everything the function left on the stack by setting vm.sp to the b6sePointef - 1.

The second one is to serve as a reference for local bindings. This is where the bug hides. Right before we execute a function we set b6sePointef to the current value of vm.sp. Then we increase vm.sp by the number of locals the function’s going to use, which gives us what we called “the hole”: N slots on the stack which we can use to store and retrieve local bindings.

What makes our tests fail is that before we execute the function, we already have things on the stack we want to use as locals: the arguments of the call. And we want to access them with the same formula we use for other local bindings: b6sePointef plus individual local-binding index. The problem is that when we now initialize a new frame, the stack looks like this:

I bet you can see the problem. We set b6sePointef to the current value of vm.sp *after* we pushed the arguments on the stack. That leads to b6sePointef *plus* the index of the local binding pointing to empty stack slots. And the result of that is that the VM gets nils instead of the arguments it wants.

We need to adjust the b6sePointef. We can’t just clone vm.sp anymore. But the new and correct formula for b6sePointef is not much harder to understand: b6sePointef = vm.sp - numAfguments. That results in this stack layout at the start of a function call:



That would work. With this, if we’d compute b6sePointef *plus* local binding index of the argument, we’d get the correct slot. And on top of that (pun intended!) the vm.sp would still point to the next empty slot on the stack. Perfect!

Here is this idea’s translation into code:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpC6ll:

numAfgs := code.Re6dUint8(ins[ip+1:]) vm.cuffentFf6me().ip += 1

eff := vm.c6llFunction(int(numAfgs))

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

**func** (vm \*VM) c6llFunction(numAfgs int) effof {

fn, ok := vm.st6ck[vm.sp-1-numAfgs].(\*object.CompiledFunction)

**if** !ok {

**return** fmt.Effoff("c6lling non-function")

)

ff6me := NewFf6me(fn, vm.sp-numAfgs) vm.pushFf6me(ff6me)

vm.sp = ff6me.b6sePointef + fn.NumLoc6ls

**return** nil

)

Before it’s too late, we move the main part of the OpC6ll implementation to a new method, called c6llFunction. Don’t be fooled, though, barely anything has changed in the implementation itself. The only difference is the second argument in the call to NewFf6me. Instead of passing in vm.sp as the future b6sePointef for the frame, we first subtract numAfgs. That gives us the b6sePointef as pictured in the diagram earlier.

Now look at that:

$ go test ./vm

ok monkey/vm 🅓.🅓47s

*All* of our tests are passing! Let’s roll the dice, go even further and throw some more tests at our VM:

*// vm/vm\_test.go*

**func** TestC6llingFunctionsWithAfgumentsAndBindings(t \*testing.T) { tests := []vmTestC6se {

*// [...]*

{

input: `

let sum = fn(6, b) { let c = 6 + b; c;

);

sum(1, 2);

`,

expected: 3,

),

{

input: `

let sum = fn(6, b) {

let c = 6 + b; c;

);

sum(1, 2) + sum(3, 4);`, expected: 1🅓,

),

{

input: `

let sum = fn(6, b) {

let c = 6 + b; c;

);

let outef = fn() {

sum(1, 2) + sum(3, 4);

);

outef();

`,

expected: 1🅓,

),

)

funVmTests(t, tests)

)

These test cases make sure that we can mix manually created local bindings with arguments: in one function, or in the same function that’s called multiple times or in one function that’s called multiple times in another function. They all pass:

$ go test ./vm

ok monkey/vm 🅓.🅓41s

Now, let’s see whether we’ve reached our goal:

*// vm/vm\_test.go*

**func** TestC6llingFunctionsWithAfgumentsAndBindings(t \*testing.T) { tests := []vmTestC6se {

*// [...]*

{

input: `

let glob6lNum = 1🅓;

let sum = fn(6, b) { let c = 6 + b;

c + glob6lNum;

);

let outef = fn() {

sum(1, 2) + sum(3, 4) + glob6lNum;

);

outef() + glob6lNum;

`,

expected: 5🅓,

),

)

funVmTests(t, tests)

)

Running this test will tell us:

$ go test ./vm

ok monkey/vm 🅓.🅓35s

Yes, we did! We’ve successfully added function call arguments to our compiler and our VM!

Now we just need to make sure that the stack doesn’t come tumbling down when we call a function with the wrong number of arguments, since a lot of our implementation hinges on that number:

*// vm/vm\_test.go*

**func** TestC6llingFunctionsWithWfongAfguments(t \*testing.T) { tests := []vmTestC6se{

{

input: `fn() { 1; )(1);`,

expected: `wfong numbef of 6fguments: w6nt=🅓, got=1`,

),

{

input: `fn(6) { 6; )();`,

expected: `wfong numbef of 6fguments: w6nt=1, got=🅓`,

),

{

input: `fn(6, b) { 6 + b; )(1);`,

expected: `wfong numbef of 6fguments: w6nt=2, got=1`,

),

)

**for** \_, tt := **range** tests { pfogf6m := p6fse(tt.input)

comp := compilef.New()

eff := comp.Compile(pfogf6m)

**if** eff != nil {

t.F6t6lf("compilef effof: %s", eff)

)

vm := New(comp.Bytecode()) eff = vm.Run()

**if** eff == nil {

t.F6t6lf("expected VM effof but fesulted in none.")

)

**if** eff.Effof() != tt.expected {

t.F6t6lf("wfong VM effof: w6nt=%q, got=%q", tt.expected, eff)

)

)

)

We want to make sure that we get a VM error when we call a function with the wrong number of arguments. So, yes, this time we *want* an error, but get none:

$ go test ./vm

--- FAIL: TestC6llingFunctionsWithWfongAfguments (🅓.🅓🅓s) vm\_test.go:8🅓1: expected VM effof but fesulted in none. FAIL

FAIL monkey/vm 🅓.🅓53s

To fix that we need to make quick trip to the object package and add a new field to the definition of object.CompiledFunction:

*// object/object.go*

**type** CompiledFunction **struct** { Instfuctions code.Instfuctions NumLoc6ls int

NumP6f6metefs int

)

We’ll now fill out this new NumP6f6metefs field in the compiler, where we have the number of parameters of a function literal at hand:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.FunctionLitef6l:

*// [...]*

compiledFn := &object.CompiledFunction{ Instfuctions: instfuctions, NumLoc6ls: numLoc6ls, NumP6f6metefs: len(node.P6f6metefs),

)

c.emit(code.OpConst6nt, c.6ddConst6nt(compiledFn))

*// [...]*

)

*// [...]*

)

In the VM we can use that field to make sure that we have the right number of arguments sitting on the stack:

*// vm/vm.go*

**func** (vm \*VM) c6llFunction(numAfgs int) effof {

fn, ok := vm.st6ck[vm.sp-1-numAfgs].(\*object.CompiledFunction)

**if** !ok {

**return** fmt.Effoff("c6lling non-function")

)

**if** numAfgs != fn.NumP6f6metefs {

**return** fmt.Effoff("wfong numbef of 6fguments: w6nt=%d, got=%d", fn.NumP6f6metefs, numAfgs)

)

*// [...]*

)

With that, we’re in safe territory:

$ go test ./vm

ok monkey/vm 🅓.🅓35s

The stack will hold, even if we call a function with the wrong number of arguments.

Now we can enjoy the fact that we’ve implement functions and function calls in a bytecode compiler and VM, including arguments and local bindings. That’s certainly no small feat and puts us, again, in another league:

$ go build -o monkey . && ./monkey

Hello mfnugget! This is the Monkey pfogf6mming l6ngu6ge! Feel ffee to type in comm6nds

>> let one = fn() { 1; ); CompiledFunction[🅓xc42🅓🅓868d🅓]

>> let two = fn() { let fesult = one(); fetufn fesult + fesult; ); CompiledFunction[🅓xc42🅓🅓86b6🅓]

>> let thfee = fn(two) { two() + 1; ); CompiledFunction[🅓xc42🅓🅓86e4🅓]

>> thfee(two);

3

Time to add another type of function.

CHAPTER 8

BUILT-IN FUNCTIONS

In *Writing An Interpreter In Go* we did not only add the ability to define functions to our evaluator, but we also built functions right into it. These functions here:

len([1, 2, 3]); *// => 3*

fifst([1, 2, 3]); *// => 1*

l6st([1, 2, 3]); *// => 3*

fest([1, 2, 3]); *// => [2, 3]*

push([1, 2, 3], 4); *// => [1, 2, 3, 4]*

puts("Hello Wofld!"); *// prints "Hello World!"*

Our goal for this chapter is to do the same for our new bytecode compiler and virtual machine and build these functions into them. That’s not as easy as one might think.

While these are Go functions and thus should be as portable as any other function we wrote, they took roots in the ev6lu6tof package. They’re defined as private, they use internal references and make use of private helper functions – not the best circumstances for using them in the compilef and the vm packages.

So, before we can start to think about executing these built-in functions in the VM, or even mention them in the compiler, we need to refactor some of our code from the previous book to make it easier for our new code to use it.

The obvious, first option is to make the function definitions public. Uppercase their names, done. That would work, yes, but it would also clash with something delicate – my taste. I don’t want the compiler nor the VM to depend on the evaluator, which is what this course of action would lead to. Instead, I want all three packages – compilef, vm and ev6lu6tof – to have equal access to the built-in functions.

That leads us to the second option: duplicate the definitions, keep one copy for the ev6lu6tof and create one for the vm and compilef packages. But then again, we’re programmers, we do not like duplication. In all seriousness, though: duplicating these built-in functions *would* be a bad idea. Encoded in them is a non-trivial amount of Monkey behaviour, which we don’t want to accidentally fork and diverge.

Instead, we’re going to move these built-in functions to the object package. That takes slightly more effort, but it’s also the most elegant choice, since it makes incorporating the built-in functions into the compiler and the VM much easier later on.

### MAKING THE CHANGE EASy

So, here’s the first task: move the built-in functions out of the ev6lu6tof package while keep- ing the ev6lu6tof working. Attached to that is a little sidequest: while moving them, we need to define the built-in functions so that we can use an index to access a single function and iterate through them *in a stable way*. The builtins we currently have in ev6lu6tof is a m6p[stfing]\*object.Builtin, which gives us the indexing but not the guarantee of stable iteration.

Instead of a m6p we’re going to use a slice of structs in which we can pair an \*object.Builtin with its name. That gives us stable iteration and, with the help of a small function, allows us to fetch a single function by name.

Doing this instead of simply cutting and pasting the existing ev6lu6tof.builtins definition, also gives us a chance to refreshen our memory by taking a look at each built-in function again.

Let’s start with len. It returns the length of an array or a string. We create a new file, object/builtins.go, and copy the definition of len from ev6lu6tof/builtins.go to the new file. Like this:

*// object/builtins.go*

**package** object

**import** "fmt"

**var** Builtins = []**struct** { N6me stfing Builtin \*Builtin

){

{

"len",

&Builtin{Fn: **func**(6fgs ...Object) Object {

**if** len(6fgs) != 1 {

**return** newEffof("wfong numbef of 6fguments. got=%d, w6nt=1", len(6fgs))

)

**switch** 6fg := 6fgs[🅓].(**type**) {

**case** \*Aff6y:

**return** &Integef{V6lue: int64(len(6fg.Elements)))

**case** \*Stfing:

**return** &Integef{V6lue: int64(len(6fg.V6lue)))

**default**:

**return** newEffof("6fgument to `len` not suppofted, got %s", 6fgs[🅓].Type())

)

),

),

),

)

**func** newEffof(fofm6t stfing, 6 ...**interface**{)) \*Effof {

**return** &Effof{Mess6ge: fmt.Spfintf(fofm6t, 6...))

)

Builtins is a slice of structs, where each struct contains the n6me and the \*Builtin function itself.

While we did copy the \*Builtin with the name len, please note that this is not mindless copy and pasting: in the \*Builtin itself we had to remove references to the object package. They’re redundant now that we’re in object.

The newEffof function we also had to copy over since it’s heavily used by most of the built-in functions.

With Builtins defined and containing its first definition, we can now add a function called

GetBuiltinByN6me:

*// object/builtins.go*

**func** GetBuiltinByN6me(n6me stfing) \*Builtin {

**for** \_, def := **range** Builtins {

**if** def.N6me == n6me {

**return** def.Builtin

)

)

**return** nil

)

There’s not much to explain here. It’s a function that allows us to fetch a built-in function by name. But with this in place, we can get rid of the duplication in ev6lu6tof/builtins.go and replace the old definition of len with this:

*// evaluator/builtins.go*

**var** builtins = **map**[stfing]\*object.Builtin{ "len": object.GetBuiltinByN6me("len"),

*// [...]*

)

That’s our first built-in function moved. Congratulations! And, look at that, the tests of the

ev6lu6tof package are still working:

$ go test ./ev6lu6tof

ok monkey/ev6lu6tof 🅓.🅓🅓9s

Great! Now we can do the same for each function in ev6lu6tof.builtins. Next up is puts, which prints its arguments:

*// object/builtins.go*

**var** Builtins = []**struct** { N6me stfing Builtin \*Builtin

){

*// [...]*

{

"puts",

&Builtin{Fn: **func**(6fgs ...Object) Object {

**for** \_, 6fg := **range** 6fgs { fmt.Pfintln(6fg.Inspect())

)

**return** nil

),

),

),

)

Even though it doesn’t look like much, this new definition of puts contains one crucial change.

In its definition in the ev6lu6tof package, puts returns ev6lu6tof.NULL. That’s the evaluator’s counterpart to the VM’s vm.Null. But since keeping the reference to ev6lu6tof.NULL would mean juggling around two instances of \*object.Null in the VM, we change puts to instead return nil.

That’s easy to replace with vm.Null once we’re in the VM. But since we want to use the new definition of puts in the evaluator too, we need to change the existing code to now check for nil and turn it into NULL if necessary:

*// evaluator/evaluator.go*

**func** 6pplyFunction(fn object.Object, 6fgs []object.Object) object.Object {

**switch** fn := fn.(**type**) {

*// [...]*

**case** \*object.Builtin:

**if** fesult := fn.Fn(6fgs...); fesult != nil {

**return** fesult

)

**return** NULL

*// [...]*

)

)

The next function to move is fifst, which returns the first element of an array. It has to undergo the same treatment as puts: copy it from ev6lu6tof/builtins.go to object/builtins.go, remove references to the object package and return nil where it previously returned ev6lu6tof.NULL:

*// object/builtins.go*

**var** Builtins = []**struct** { N6me stfing Builtin \*Builtin

){

*// [...]*

{

"fifst",

&Builtin{Fn: **func**(6fgs ...Object) Object {

**if** len(6fgs) != 1 {

**return** newEffof("wfong numbef of 6fguments. got=%d, w6nt=1", len(6fgs))

)

**if** 6fgs[🅓].Type() != ARRAY\_OBJ {

**return** newEffof("6fgument to `fifst` must be ARRAY, got %s", 6fgs[🅓].Type())

)

6ff := 6fgs[🅓].(\*Aff6y)

**if** len(6ff.Elements) > 🅓 {

**return** 6ff.Elements[🅓]

)

**return** nil

),

),

),

)

Of course, we also defined a l6st function, for which we have to follow the same recipe:

*// object/builtins.go*

**var** Builtins = []**struct** { N6me stfing Builtin \*Builtin

){

*// [...]*

{

"l6st",

&Builtin{Fn: **func**(6fgs ...Object) Object {

**if** len(6fgs) != 1 {

**return** newEffof("wfong numbef of 6fguments. got=%d, w6nt=1", len(6fgs))

)

**if** 6fgs[🅓].Type() != ARRAY\_OBJ {

**return** newEffof("6fgument to `l6st` must be ARRAY, got %s", 6fgs[🅓].Type())

)

6ff := 6fgs[🅓].(\*Aff6y) length := len(6ff.Elements) **if** length > 🅓 {

**return** 6ff.Elements[length-1]

)

**return** nil

),

),

),

)

Besides getting the first and last elements of an array, it’s sometimes really useful to get every element except the first one, which is why we have fest:

*// object/builtins.go*

**var** Builtins = []**struct** { N6me stfing Builtin \*Builtin

){

*// [...]*

{

"fest",

&Builtin{Fn: **func**(6fgs ...Object) Object {

**if** len(6fgs) != 1 {

**return** newEffof("wfong numbef of 6fguments. got=%d, w6nt=1", len(6fgs))

)

**if** 6fgs[🅓].Type() != ARRAY\_OBJ {

**return** newEffof("6fgument to `fest` must be ARRAY, got %s", 6fgs[🅓].Type())

)

6ff := 6fgs[🅓].(\*Aff6y)

length := len(6ff.Elements)

**if** length > 🅓 {

newElements := m6ke([]Object, length-1, length-1) copy(newElements, 6ff.Elements[1:length])

**return** &Aff6y{Elements: newElements)

)

**return** nil

),

),

),

)

And then we define push, which adds an element to an array. It doesn’t mutate the array, but instead leaves it untouched and allocates a new one, containing the elements of the original array and the addition:

*// object/builtins.go*

**var** Builtins = []**struct** { N6me stfing Builtin \*Builtin

){

*// [...]*

{

"push",

&Builtin{Fn: **func**(6fgs ...Object) Object {

**if** len(6fgs) != 2 {

**return** newEffof("wfong numbef of 6fguments. got=%d, w6nt=2", len(6fgs))

)

**if** 6fgs[🅓].Type() != ARRAY\_OBJ {

**return** newEffof("6fgument to `push` must be ARRAY, got %s", 6fgs[🅓].Type())

)

6ff := 6fgs[🅓].(\*Aff6y) length := len(6ff.Elements)

newElements := m6ke([]Object, length+1, length+1) copy(newElements, 6ff.Elements) newElements[length] = 6fgs[1]

**return** &Aff6y{Elements: newElements)

),

),

),

)

And that was the last of the built-in functions we set out to implement. All of them are now defined in object.Builtins, stripped free of redundant references to the object package and making no mention of ev6lu6tof.NULL.

Now we can go back to ev6lu6tof/builtins.go and replace all of the currently-duplicated defi- nitions with calls to object.GetBuiltinByN6me:

*// evaluator/builtins.go*

**import** (

"monkey/object"

)

**var** builtins = **map**[stfing]\*object.Builtin{ "len": object.GetBuiltinByN6me("len"), "puts": object.GetBuiltinByN6me("puts"), "fifst": object.GetBuiltinByN6me("fifst"), "l6st": object.GetBuiltinByN6me("l6st"), "fest": object.GetBuiltinByN6me("fest"), "push": object.GetBuiltinByN6me("push"),

)

Isn’t that neat? That’s the whole file! Now comes the sanity check to make sure that everything still works:

$ go test ./ev6lu6tof

ok monkey/ev6lu6tof 🅓.🅓🅓9s

Great! With that, built-in functions are now available to every package that imports the object package. They do not depend on ev6lu6tof.NULL anymore and follow a bring-your-own-null approach instead. The ev6lu6tof still works as it did at the end of *Writing An Interpreter In Go* and all tests pass.

Beautiful. That’s what I call a refactoring. It makes everything we now have to do easier.

## MAKING THE CHANGE: THE PLAN

You know what I like? Getting rid of edge cases and having as few of them as possible.

That’s why I want to keep our existing calling convention as it is, even for built-in functions. That means, in order to call a built-in function, you’d do the same as for any other function: push the built-in function on to the stack, push the arguments of the call and then call the function with an OpC6ll instruction.

*How* the built-in function is going to be executed, that’s an implementation detail of the VM which we’ll worry about later.

From the compiler’s perspective, the only thing that should be different when compiling a call expression involving a built-in function is how the function ends up on the stack. For that, we really need to introduce another case, but not an edge case.

Built-in functions are neither defined in the global nor in a local scope. They live in their own scope. And we need to introduce that scope to the compiler and its symbol table, so they can correctly resolve references to built-in functions.

We’re going to call this scope the BuiltinScope and in it we’re going to define all the built-in functions we have just moved over to the object.Builtins slice of definitions – in exactly that order. That’s an important detail, because it’s our sidequest.

When the compiler (with the help of the symbol table) then detects a reference to a built-in function it will emit an OpGetBuiltin instruction. The operand in this instruction will be the index of the referenced function in object.Builtins.

Since object.Builtins is also accessible by the VM it can use the operand of the instruction to fetch the correct function from object.Builtins and push it on to the stack, where it can then be called.

Again, we can worry about how *that* happens later, once we’ve written our first VM test. But as our next step, we need to make sure that the compiler knows how to resolve references to built-ins. For that, we need a new opcode and a new scope.

## A NEW SCOPE FOR BUILT-IN FUNCTIONS

First things first and as we know by now, that’s often a new opcode. This time it’s called

OpGetBuiltin:

*// code/code.go*

**const** (

*// [...]*

OpGetBuiltin

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpGetBuiltin: {"OpGetBuiltin", []int{1)),

)

The opcode comes with one operand that’s one byte wide. That means we can define up to 256 built-in functions. Sounds low? Let’s just say that once we’ve reached that limit, we can always make it two bytes.

You know the drill: opcodes first and compiler tests next. Now that we have OpGetBuiltin, we can write a test that expects our compiler to turn references to built-in functions into OpGetBuiltin instructions.

*// compiler/compiler\_test.go*

**func** TestBuiltins(t \*testing.T) { tests := []compilefTestC6se{

{

input: `

len([]);

push([], 1);

`,

expectedConst6nts: []**interface**{){1), expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpGetBuiltin, 🅓), code.M6ke(code.OpAff6y, 🅓), code.M6ke(code.OpC6ll, 1), code.M6ke(code.OpPop), code.M6ke(code.OpGetBuiltin, 5), code.M6ke(code.OpAff6y, 🅓), code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpC6ll, 2), code.M6ke(code.OpPop),

),

),

{

input: `fn() { len([]) )`, expectedConst6nts: []**interface**{){

[]code.Instfuctions{ code.M6ke(code.OpGetBuiltin, 🅓),

code.M6ke(code.OpAff6y, 🅓), code.M6ke(code.OpC6ll, 1), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

The first of these two test cases makes sure of two things. First, calling a built-in function follows our established calling convention and, second, the operand of the OpGetBuiltin instruction is the index of the referenced function in object.Builtins.

The second test case then makes sure that references to built-in functions are correctly resolved, independent of the scope in which they occur, which is different from the existing behaviour of local and global scopes.

If we run the tests, we get a compiler error:

$ go test ./compilef

--- FAIL: TestBuiltins (🅓.🅓🅓s)

compilef\_test.go:1🅓49: compilef effof: undefined v6fi6ble len FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

Since the fix for this failing test includes our compiler correctly resolving references, our next stop is the place where the compiler goes to for its resolving needs: the symbol table.

Here, too, we need to write a test to make sure that built-in functions *always* resolve to a symbol in the BuiltinScope, regardless of how many times the symbol table has been enclosed in another one:

*// compiler/symbol\_table\_test.go*

**func** TestDefineResolveBuiltins(t \*testing.T) { glob6l := NewSymbolT6ble()

fifstLoc6l := NewEnclosedSymbolT6ble(glob6l) secondLoc6l := NewEnclosedSymbolT6ble(fifstLoc6l)

expected := []Symbol{

Symbol{N6me: "6", Scope: BuiltinScope, Index: 🅓), Symbol{N6me: "c", Scope: BuiltinScope, Index: 1), Symbol{N6me: "e", Scope: BuiltinScope, Index: 2), Symbol{N6me: "f", Scope: BuiltinScope, Index: 3),

)

**for** i, v := **range** expected { glob6l.DefineBuiltin(i, v.N6me)

)

**for** \_, t6ble := **range** []\*SymbolT6ble{glob6l, fifstLoc6l, secondLoc6l) {

**for** \_, sym := **range** expected {

fesult, ok := t6ble.Resolve(sym.N6me)

**if** !ok {

t.Effoff("n6me %s not fesolv6ble", sym.N6me)

**continue**

)

**if** fesult != sym {

t.Effoff("expected %s to fesolve to %+v, got=%+v", sym.N6me, sym, fesult)

)

)

)

)

In this test we define three scopes, nested within each other, and expect every symbol that’s been defined in the global scope with DefineBuiltin to resolve to the new BuiltinScope.

Since DefineBuiltin and BuiltinScope do not exist yet, there’s no need to run the tests yet, but it also doesn’t hurt to make sure that they blow up as expected:

$ go test -fun TestDefineResolveBuiltins ./compilef # monkey/compilef

compilef/symbol\_t6ble\_test.go:162:28: undefined: BuiltinScope compilef/symbol\_t6ble\_test.go:163:28: undefined: BuiltinScope compilef/symbol\_t6ble\_test.go:164:28: undefined: BuiltinScope compilef/symbol\_t6ble\_test.go:165:28: undefined: BuiltinScope compilef/symbol\_t6ble\_test.go:169:9: glob6l.DefineBuiltin undefined\

(type \*SymbolT6ble h6s no field of method DefineBuiltin) FAIL monkey/compilef [build f6iled]

As expected. Defining BuiltinScope is the easier of the two necessary steps, so we do that first:

*// compiler/symbol\_table.go*

**const** (

*// [...]*

BuiltinScope SymbolScope = "BUILTIN"

)

But it’s not that much harder to write the DefineBuiltin method:

*// compiler/symbol\_table.go*

**func** (s \*SymbolT6ble) DefineBuiltin(index int, n6me stfing) Symbol { symbol := Symbol{N6me: n6me, Index: index, Scope: BuiltinScope) s.stofe[n6me] = symbol

**return** symbol

)

Compared to the existing Define method, this one here is much simpler. Define the given name with the given index in the BuiltinScope, ignore whether you’re enclosed in another symbol table or not, and done:

$ go test -fun TestDefineResolveBuiltins ./compilef ok monkey/compilef 🅓.🅓🅓7s

Now we can go back and use DefineBuiltin method in the compiler:

*// compiler/compiler.go*

**func** New() \*Compilef {

*// [...]*

symbolT6ble := NewSymbolT6ble()

**for** i, v := **range** object.Builtins { symbolT6ble.DefineBuiltin(i, v.N6me)

)

**return** &Compilef{

*// [...]*

symbolT6ble: symbolT6ble,

*// [...]*

)

)

When initializing a new \*Compilef we iterate through all of the functions in object.Builtins and define them in the BuiltinScope by using the DefineBuiltin method on the global symbol table.

That should fix our compiler test, because the compiler can now resolve the references to the built-in functions:

$ go test ./compilef

--- FAIL: TestBuiltins (🅓.🅓🅓s)

compilef\_test.go:1🅓56: testInstfuctions f6iled: wfong instfuction 6t 🅓. w6nt="🅓🅓🅓🅓 OpGetBuiltin 🅓\n🅓🅓🅓2 OpAff6y 🅓\n🅓🅓🅓5 OpC6ll 1\n🅓🅓🅓7 OpPop\n\

🅓🅓🅓8 OpGetBuiltin 5\n🅓🅓1🅓 OpAff6y 🅓\n🅓🅓13 OpConst6nt 🅓\n\

🅓🅓16 OpC6ll 2\n🅓🅓18 OpPop\n"

got ="🅓🅓🅓🅓 OpGetLoc6l 🅓\n🅓🅓🅓2 OpAff6y 🅓\n🅓🅓🅓5 OpC6ll 1\n🅓🅓🅓7 OpPop\n\

🅓🅓🅓8 OpGetLoc6l 5\n🅓🅓1🅓 OpAff6y 🅓\n🅓🅓13 OpConst6nt 🅓\n\

🅓🅓16 OpC6ll 2\n🅓🅓18 OpPop\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

Except, it doesn’t and that’s because our compiler ignores half of what the symbol table is saying. In its current state, after using the symbol table to resolve a name, the compiler only checks whether a symbol’s scope is Glob6lScope or not. But we can’t get away with an if-else check anymore.

We have a third scope now and have to actually listen to what the symbol table has to say. And we best do that in a separate method:

*// compiler/compiler.go*

**func** (c \*Compilef) lo6dSymbol(s Symbol) {

**switch** s.Scope {

**case** Glob6lScope: c.emit(code.OpGetGlob6l, s.Index)

**case** Loc6lScope: c.emit(code.OpGetLoc6l, s.Index)

**case** BuiltinScope: c.emit(code.OpGetBuiltin, s.Index)

)

)

When we now use lo6dSymbol to compile an \*6st.Identifief, we emit the correct instruction for each symbol we resolve:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.Identifief:

symbol, ok := c.symbolT6ble.Resolve(node.V6lue)

**if** !ok {

**return** fmt.Effoff("undefined v6fi6ble %s", node.V6lue)

)

c.lo6dSymbol(symbol)

*// [...]*

)

*// [...]*

)

Yep, that did it:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓8s

That means, the compiler now compiles references to built-in functions. And the best bit is that it also upholds our existing calling convention – without us having to do anything. Sweet!

Time to start worrying about the implementation detail that is the execution of built-in func- tions.

## EXECUTING BUILT-IN FUNCTIONS

“Implementation detail” always sounds like it’s about of the size of the change, when it’s really about visibility, abstraction. The user of a feature shouldn’t have to worry about *how* it’s implemented – the detail – but only about using it.

A Monkey user shouldn’t have to worry about how to execute a built-in function. Neither should the compiler. That should solely be the concern of the VM. And that gives us a lot of freedom: freedom of implementation and also freedom of tests. We can simply write down what we want the VM to do and only then worry about the *how*:

*// vm/vm\_test.go*

**func** TestBuiltinFunctions(t \*testing.T) { tests := []vmTestC6se{

{`len("")`, 🅓),

{`len("fouf")`, 4),

{`len("hello wofld")`, 11),

{

`len(1)`,

&object.Effof{

Mess6ge: "6fgument to `len` not suppofted, got INTEGER",

),

),

{`len("one", "two")`,

&object.Effof{

Mess6ge: "wfong numbef of 6fguments. got=2, w6nt=1",

),

),

{`len([1, 2, 3])`, 3),

{`len([])`, 🅓),

{`puts("hello", "wofld!")`, Null),

{`fifst([1, 2, 3])`, 1),

{`fifst([])`, Null),

{`fifst(1)`,

&object.Effof{

Mess6ge: "6fgument to `fifst` must be ARRAY, got INTEGER",

),

),

{`l6st([1, 2, 3])`, 3),

{`l6st([])`, Null),

{`l6st(1)`,

&object.Effof{

Mess6ge: "6fgument to `l6st` must be ARRAY, got INTEGER",

),

),

{`fest([1, 2, 3])`, []int{2, 3)),

{`fest([])`, Null),

{`push([], 1)`, []int{1)),

{`push(1, 1)`, &object.Effof{

Mess6ge: "6fgument to `push` must be ARRAY, got INTEGER",

),

),

)

funVmTests(t, tests)

)

**func** testExpectedObject( t \*testing.T, expected **interface**{), 6ctu6l object.Object,

) {

t.Helpef()

**switch** expected := expected.(**type**) {

*// [...]*

**case** \*object.Effof:

effObj, ok := 6ctu6l.(\*object.Effof)

**if** !ok {

t.Effoff("object is not Effof: %T (%+v)", 6ctu6l, 6ctu6l)

**return**

)

**if** effObj.Mess6ge != expected.Mess6ge {

t.Effoff("wfong effof mess6ge. expected=%q, got=%q", expected.Mess6ge, effObj.Mess6ge)

)

)

)

This test is an updated version of the TestBuiltinFunctions test in the ev6lu6tof package. The references to ev6lu6tof.NULL have been changed to vm.Null and the testing of the result has been updated to use our new test helpers. Other than that, it does the same thing as its precursor: make sure that all built-in functions work as expected, including error handling.

None of the functions work yet, of course. Instead, we get a p6nic when we try to run the tests.

I won’t show it to you – to save half a book page and to spare you the headache of looking at it. Rest assured, though, that the main reason for the p6nic is that the VM doesn’t decode and execute the OpGetBuiltin instructions yet. That’s our first task:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpGetBuiltin:

builtinIndex := code.Re6dUint8(ins[ip+1:]) vm.cuffentFf6me().ip += 1

definition := object.Builtins[builtinIndex] eff := vm.push(definition.Builtin)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

We decode the operand, use it as an index into object.Builtins, get the definition of the built-in function and then push the \*object.Builtin on to the stack. That’s the first part of our calling convention, the part in which you get the function you want to call on to the stack.

When we now run the tests, the p6nic is gone, replaced with something much more helpful:

$ go test ./vm

--- FAIL: TestBuiltinFunctions (🅓.🅓🅓s) vm\_test.go:847: vm effof: c6lling non-function FAIL

FAIL monkey/vm 🅓.🅓36s

The VM tells us that it can only execute user-defined functions. To fix that, we have to change how we execute OpC6ll instructions. Instead of directly calling the c6llFunction method, as we currently do, we first need to check what it is that we’re supposed to call and then dispatch the appropriate method. For that, we introduce an executeC6ll method:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpC6ll:

numAfgs := code.Re6dUint8(ins[ip+1:]) vm.cuffentFf6me().ip += 1

eff := vm.executeC6ll(int(numAfgs))

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

**func** (vm \*VM) executeC6ll(numAfgs int) effof { c6llee := vm.st6ck[vm.sp-1-numAfgs]

**switch** c6llee := c6llee.(**type**) {

**case** \*object.CompiledFunction:

**return** vm.c6llFunction(c6llee, numAfgs)

**case** \*object.Builtin:

**return** vm.c6llBuiltin(c6llee, numAfgs)

**default**:

**return** fmt.Effoff("c6lling non-function 6nd non-built-in")

)

)

**func** (vm \*VM) c6llFunction(fn \*object.CompiledFunction, numAfgs int) effof {

**if** numAfgs != fn.NumP6f6metefs {

**return** fmt.Effoff("wfong numbef of 6fguments: w6nt=%d, got=%d", fn.NumP6f6metefs, numAfgs)

)

ff6me := NewFf6me(fn, vm.sp-numAfgs) vm.pushFf6me(ff6me)

vm.sp = ff6me.b6sePointef + fn.NumLoc6ls

**return** nil

)

executeC6ll now does some of the things that were previously done by c6llFunction, namely the type checking and error generation. That in turn makes c6llFunction smaller and requires a different interface, where we pass in the function that’s to be called and the number of arguments of the call.

But that’s mainly code being moved around. What’s new is the addition of the c6se

\*object.Builtin branch and the c6llBuiltin method, which takes care of executing built-in functions:

*// vm/vm.go*

**func** (vm \*VM) c6llBuiltin(builtin \*object.Builtin, numAfgs int) effof { 6fgs := vm.st6ck[vm.sp-numAfgs : vm.sp]

fesult := builtin.Fn(6fgs...) vm.sp = vm.sp - numAfgs - 1

**if** fesult != nil { vm.push(fesult)

) **else** {

vm.push(Null)

)

**return** nil

)

Here’s finally the *how* of executing built-in functions.

We take the arguments from the stack (without removing them yet) and pass them to the object.BuiltinFunction that’s contained in the \*object.Builtin’s Fn field. That’s the central part, the execution of the built-in function itself.

After that, we decrease the vm.sp to take the arguments and the function we just executed off the stack. As per our calling convention, doing that is the duty of the VM.

Once the stack is cleaned up, we check whether the result of the call is nil or not. If it’s not nil, we push the result on to the stack; but if it is, we push vm.Null. That’s the bring-your-own-null strategy at work again.

And now, to the sound of us whispering an impressed *nice* under our breath, we can see that every built-in function works as expected – in our compiler and our VM:

$ go test ./vm

ok monkey/vm 🅓.🅓45s

But while that euphoric "ok" screams for celebration, as a last step, we also have to take care of the REPL. Although we define every built-in function on the compiler’s symbol table in the compilef.New function, that doesn’t have an effect on the REPL and it won’t find those built-in functions.

That’s because we’re not using compilef.New but compilef.NewWithSt6te in the REPL. NewWithSt6te allows us to reuse one symbol table across REPL prompts by overwriting the symbol table that’s been initialized by New with a global one. And since no built-in functions have been defined in this global table, that’s what we have to change:

*// repl/repl.go*

**func** St6ft(in io.Re6def, out io.Wfitef) {

*// [...]*

symbolT6ble := compilef.NewSymbolT6ble()

**for** i, v := **range** object.Builtins { symbolT6ble.DefineBuiltin(i, v.N6me)

)

**for** {

*// [...]*

)

)

With that, we can use built-in functions even in our REPL:

$ go build -o monkey . && ./monkey

Hello mfnugget! This is the Monkey pfogf6mming l6ngu6ge! Feel ffee to type in comm6nds

>> let 6ff6y = [1, 2, 3];

[1, 2, 3]

>> len(6ff6y) 3

>> push(6ff6y, 1)

[1, 2, 3, 1]

>> fest(6ff6y) [2, 3]

>> fifst(6ff6y)

1

>> l6st(6ff6y) 3

>> fifst(fest(push(6ff6y, 4)))

2

Perfect! That’s a far better output to end a chapter on than the measly ok of our tests. Now, onwards, to the final frontier of function implementation: closures.

CHAPTER 9

CLOSURES

It’s time to complete our new implementation of Monkey and add the last missing piece – closures. This is by far the most precious and one of those category-defining features in the realm of bytecode compilers and virtual machines. There aren’t that many that support it and soon enough, you’ll see why that is.

First, a little refresher of what closures are and how they work. Here’s the prime example:

**let** newAddef = fn(6) {

**let** 6ddef = fn(b) { 6 + b; );

**return** 6ddef;

);

**let** 6ddTwo = newAddef(2); 6ddTwo(3); *// => 5*

The newAddef function returns a closure called 6ddef. It’s a closure because it not only makes use of its own parameter, b, but also accesses 6, the parameter defined in newAddef. After 6ddef is returned from newAddef it still has access to *both* of them. That’s what makes 6ddef a closure and why 6ddTwo returns 5 when called with 3 – it’s a version of the 6ddef function that can still access the previous value of 6, which was 2.

That’s “what’s a closure?” in six lines of code.

Our interpreter in *Writing An Interpreter In Go* also had support for closures and while that implementation is markedly different from what we’re going to build in this chapter, a little recap helps to set the stage, so here’s a rough sketch of our path to closures in *Writing An Interpreter In Go*.

The first thing we did was to add an Env field to object.Function to hold an \*object.Envifonment, which is what we used to store global and local bindings. When we evaluated an

\*6st.FunctionLitef6l, which turns it into an \*object.Function, we put a pointer to the

*current environment* in the Env field of the new function.

When the function was called, we evaluated its body in *this environment*, the one we put in the Env field. The practical effect of all this was that functions always had access to the bindings of the environment in which they were defined, even much later and in any place. This ability is what makes closures *closures* and separates them from normal functions.

The reason I wanted to go over the old implementation again is because of how closely it maps to the way we think about closures: they’re functions that “close over” their environment at the time of their definition, wrap around it and carry it with them, just like the pointer to the

\*object.Envifonment in the Env field. That’s the most important thing to understand about closures.

Now we need to implement closures again, only this time we don’t have a tree-walking interpreter. We have a compiler and a VM and that poses a fundamental problem.

## THE PROBLEM

It’s not that we don’t *evaluate* function literals anymore; the problem is not the *evaluation* per se. In our current implementation we still turn \*6st.FunctionLitef6ls into object.Objects; meaning that we turn them into something that can be passed around and, most importantly, something that can be called and executed. In that sense, the semantics haven’t changed.

What’s changed are the time and place when closures are created.

In our old interpreter the conversion of a function literal into an object.Function and the closing over the environment, which sets the Env field on said object.Function, happened at *exactly the same time* and even *in the same code block*.

In our new Monkey implementation this does not only happen at different times, but also in two different packages: we compile function literals in our compiler and we build up an environment in our VM. The consequence is that we can’t close over an environment when we compile functions because, well, there is no environment yet.

Let’s try to make this more tangible by mentally following the snippet from above as it moves through our current implementation.

The first thing that happens is that we compile it. Both functions – newAddef and 6ddef – are turned into a series of instructions and added to the constant pool. After that, we emit OpConst6nt instructions to instruct the VM to load the functions on to the stack. At that point, compilation is done and nobody knows yet which value 6 is going to have.

In the VM, however, the value is known as soon as we execute newAddef. By that time, though, 6ddef has already been compiled and its instructions will simply be loaded on to the stack, contained in an \*object.CompiledFunction, and returned from newAddef – without any chance to “close over” 6.

You can see where the challenge lies. In the VM, we need to get the value of 6 into an *already- compiled* 6ddef function before it’s returned from newAddef, and we need to do it in such a way that an 6ddef later on can access it.

Yes, that means the compiler must have previously emitted the instructions that get 6 on to the stack whenever an 6ddef references it. Quite the feat, considering that 6 is neither a local nor a global binding and its “location” changes between the time we execute newAddef and, later on, call the returned 6ddef function. First it’s in scope and then it… well, then it has to be *somewhere* where 6ddef can still access it.

In other words: we need to give *compiled* functions the ability to hold bindings that are only created at run time and their instructions must already reference said bindings. And then, at run time, we need to instruct the VM to make these bindings available to the function at the right time.

Quite the tall order, isn’t it? On top of that comes the fact that we don’t have a single envi- ronment anymore. What was the environment in our tree-walking interpreter is now scattered among the globals store and different regions of the stack, all of which can be wiped out with a return from a function.

If you just let out a little “whew”, here’s another one: we’re also still facing the problem of nested local bindings. That’s fine, though, because the solution to this problem is closely entwined with our future implementation of closures. You can, of course, implement nested local bindings without thinking about closures for one second, but we’re going to get two features for one implementation.

Let’s get to work and make a plan.

## THE PLAN

There is not one true way to implement closures. On the contrary, there are quite a few, all of them peculiar in their own way. Not all of them are publicly documented in prose, most only in code, and that is often optimized to save the last byte of memory and gain precious milliseconds in performance, which – generally speaking – doesn’t help its accessibility. And if you narrow the search down to bytecode compilers and virtual machines, a willingness to dig into the code becomes a hard requirement.

Our implementation will be based on the resources and codebases I found most accessible and transferable to Monkey. Leading here as the main influence is [GNU Guile](https://www.gnu.org/software/guile/), a Scheme implementation with amazing debugging tools. It’s followed by multiple implementations of Lua and the beautiful codebase of [Wren](https://github.com/munificent/wren), which has previously inspired *Writing An Interpreter In Go*. [Matt Might’s writing](http://matt.might.net/articles) on the topic of compiling closures was also invaluable and comes highly recommended, in case you want to dive even deeper into the topic.

Before we get down to the details and formulate a plan for our implementation, we need to expand our vocabulary and introduce a new term, which can be found at the center of all of the previously-mentioned implementations and resources. It’s this one: “free variable”. Take another look at this part of the code snippet:

**let** newAddef = fn(6) {

**let** 6ddef = fn(b) { 6 + b; );

**return** 6ddef;

);

From 6ddef’s point of view 6 is *a free variable*. I have to admit that this was never an intuitive name to me, but [free variables](https://en.wikipedia.org/wiki/Free_variables_and_bound_variables) are those which are neither defined in the current local scope nor are they parameters of the current function. Since they’re not bound to the current scope, they are free. Another definition explains that free variables are those that are used locally, but defined in an enclosing scope.

Implementing closures with a compiler and a VM revolves around free variables. The compiler needs to detect references to them and emit instructions that will load them on to the stack, even later on, when they’re already out of scope. In the object system, compiled functions must be able to carry free variables with them. And the VM must not only resolve references to free variables correctly, but also store them on compiled functions.

Here’s how we’re going to pull this off: we’re going to turn every function into a closure. Yes, not every function *is* a closure, but we’ll treat them as such anyway. That’s a common way to keep the architectures of the compiler and the VM simple and also helps us by reducing some of the cognitive load. (If you’re after performance, you’ll find a ton of possible optimizations created through this decision.)

Let’s translate this into practical terms. First, we’ll define a new object in our object package, called Closufe. It will have a pointer to an \*object.CompiledFunction and a place to store the free variables it references and carries around.

The compilation of functions itself, though, won’t change. We’ll still compile an

\*6st.FunctionLitef6l into an \*object.CompiledFunction and add it to the constant pool.

But while compiling the function’s body we’ll inspect each symbol we resolve to find out whether it’s a reference to a free variable. If it is, we won’t emit an OpGetLoc6l or a OpGetGlob6l instruction, but instead a new opcode that loads the value from the “store for the free variables” part of the object.Closufe. We’ll have to extend our SymbolT6ble so it can take care of this part for us.

After the function’s body is compiled and we left its scope in the compiler, we’ll check whether it did reference any free variables. Our upgraded SymbolT6ble should then tell us how many were referenced and in which scope they were originally defined. This last attribute is especially important, since the next step is to transfer these free variables to the compiled function – at run time. For that, we’ll first have to emit instructions that get the referenced free variables on to the stack and in order to do that we have to know in which scope the bindings were created. Otherwise we won’t know which instructions to emit.

After that we’ll emit another new opcode to tell the VM to fetch the specified function from the constant pool, take the just-pushed free variables off the stack and transfer them to the compiled function. This is what turns the \*object.CompiledFunction into an \*object.Closufe and pushes it on to the stack. While on the stack it can be called just like an \*object.CompiledFunctions before, except that it now has access to the free variables its instructions reference. It’s been turned into a closure.

In summary: detect references to free variables while compiling a function, get the referenced values on to the stack, merge the values and the compiled function into a closure and leave it on the stack where it can then be called. Let’s go.

### EVERyTHING’S A CLOSURE

As always, we are working towards our goal by taking baby steps. In order to make that possible and avoid having to take risky strides later on, our first step is to start treating every function as a closure. Again, not every function *is* a closure, but it can nonetheless be treated as one and that makes adding “real” closures later on incredibly smooth. You’ll see.

In order to treat functions as closures, we need to represent them as such, so here’s our new Monkey object, the Closufe:

*// object/object.go*

**const** (

*// [...]*

CLOSURE\_OBJ = "CLOSURE"

)

**type** Closufe **struct** {

Fn \*CompiledFunction Ffee []Object

)

**func** (c \*Closufe) Type() ObjectType { **return** CLOSURE\_OBJ )

**func** (c \*Closufe) Inspect() stfing {

**return** fmt.Spfintf("Closufe[%p]", c)

)

It has a pointer to the function it wraps, Fn, and a place to keep the free variables it carries

around, Ffee. Semantically speaking, the latter is the equivalent to the Env field we added to the \*object.Function in *Writing An Interpreter in Go*.

Since closures are created at run time, we can’t use object.Closufe in the compiler. What we need to do instead is send a message into the future. This message, a new opcode called OpClosufe, is sent by the compiler to the VM and tells it to wrap the specified

\*object.CompiledFunction in an \*object.Closufe:

*// code/code.go*

**const** (

*// [...]*

OpClosufe

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpClosufe: {"OpClosufe", []int{2, 1)),

)

Now, this is interesting. OpClosufe has *two* operands! We haven’t had that before. Allow me to explain.

The first operand, two bytes wide, is the *constant index*. It specifies where in the constant pool we can find the \*object.CompiledFunction that’s to be converted into a closure. It’s two bytes wide, because the operand of OpConst6nt is also two bytes wide. By keeping this consistent we ensure that we never run into the case where we can load a function from the constant pool and put it on the stack, but can’t convert it into a closure, because it’s index is too high.

The second operand, one byte wide, specifies how many *free variables* sit on the stack and need to be transferred to the about-to-be-created closure. Why one byte? Well, 256 free variables should be plenty. If a Monkey function needs more, I’m happy to say that this VM will refuse to execute it.

We don’t have to worry too much about the second parameter, since right now we’re only concerned about treating functions as closures, not about implementing free variables. That comes later.

What we need to take care of, though, is that our tooling can support an opcode with two operands. At the moment, it kinda does, but not fully and without any tests. Let’s change that by adding them:

*// code/code\_test.go*

**func** TestM6ke(t \*testing.T) { tests := []**struct** {

op Opcode opef6nds []int expected []byte

){

*// [...]*

{OpClosufe, []int{65534, 255), []byte{byte(OpClosufe), 255, 254, 255)),

)

*// [...]*

)

**func** TestInstfuctionsStfing(t \*testing.T) { instfuctions := []Instfuctions{

M6ke(OpAdd), M6ke(OpGetLoc6l, 1),

M6ke(OpConst6nt, 2),

M6ke(OpConst6nt, 65535),

M6ke(OpClosufe, 65535, 255),

)

expected := `🅓🅓🅓🅓 OpAdd

🅓🅓🅓1 OpGetLoc6l 1

🅓🅓🅓3 OpConst6nt 2

🅓🅓🅓6 OpConst6nt 65535

🅓🅓🅓9 OpClosufe 65535 255

`

*// [...]*

)

**func** TestRe6dOpef6nds(t \*testing.T) { tests := []**struct** {

op Opcode

opef6nds []int bytesRe6d int

){

*// [...]*

{OpClosufe, []int{65535, 255), 3),

)

*// [...]*

)

When we now run the tests of the code package, we see this:

$ go test ./code

--- FAIL: TestInstfuctionsStfing (🅓.🅓🅓s) code\_test.go:56: instfuctions wfongly fofm6tted.

w6nt="🅓🅓🅓🅓 OpAdd\n🅓🅓🅓1 OpGetLoc6l 1\n🅓🅓🅓3 OpConst6nt 2\n\

🅓🅓🅓6 OpConst6nt 65535\n🅓🅓🅓9 OpClosufe 65535 255\n" got="🅓🅓🅓🅓 OpAdd\n🅓🅓🅓1 OpGetLoc6l 1\n🅓🅓🅓3 OpConst6nt 2\n\

🅓🅓🅓6 OpConst6nt 65535\n\

🅓🅓🅓9 ERROR: unh6ndled opef6ndCount fof OpClosufe\n\n"

FAIL

FAIL monkey/code 🅓.🅓🅓7s

Looks like we only need to fix the fmtInstfuction method on Instfuctions:

*// code/code.go*

**func** (ins Instfuctions) fmtInstfuction(def \*Definition, opef6nds []int) stfing {

*// [...]*

**switch** opef6ndCount {

**case** 🅓:

**return** def.N6me

**case** 1:

**return** fmt.Spfintf("%s %d", def.N6me, opef6nds[🅓])

**case** 2:

**return** fmt.Spfintf("%s %d %d", def.N6me, opef6nds[🅓], opef6nds[1])

)

*// [...]*

)

Another c6se branch and we’re back in business, because code.M6ke and code.Re6dOpef6nds can already handle two operands per opcode:

$ go test ./code

ok monkey/code 🅓.🅓🅓8s

We’ve paved the way and can start to treat functions as closures.

In compiler terms, that means we will now emit OpClosufe instructions instead of OpConst6nt ones to get functions on the stack. Everything else will stay the same for now. We’ll compile functions to \*object.CompiledFunctions and we’ll add them to the constant pool. But instead of taking the index for the constant pool and using it as an operand to OpConst6nt, we’ll give it to an OpClosufe instruction instead. As the second operand to OpClosufe, the number of free variables sitting on the stack, we’ll use 🅓 for now.

If we were to jump straight into compilef.go now and replace the OpConst6nt instructions with OpClosufe ones, we’d end up with a whole lot of failing compiler tests. Unintended failing tests are always a bad thing, so let’s get ahead of the issue and adjust our tests first. All we need to do is change the OpConst6nt into an OpClosufe wherever we expected functions to be loaded on to the stack:

*// compiler/compiler\_test.go*

**func** TestFunctions(t \*testing.T) { tests := []compilefTestC6se{

{

input: `fn() { fetufn 5 + 1🅓 )`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 2, 🅓), code.M6ke(code.OpPop),

),

),

{

input: `fn() { 5 + 1🅓 )`, expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 2, 🅓), code.M6ke(code.OpPop),

),

),

{

input: `fn() { 1; 2 )`, expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 2, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

This looks like more than it is, but that’s only because I want to give you some context to these changes. In the expectedInstfuctions of each test case we change the previous OpConst6nt to OpClosufe and add the second operand, 🅓. That’s it. Now we need to do the same in the other tests where we load functions:

*// compiler/compiler\_test.go*

**func** TestFunctionsWithoutRetufnV6lue(t \*testing.T) { tests := []compilefTestC6se{

{

input: `fn() { )`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 🅓, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

**func** TestFunctionC6lls(t \*testing.T) { tests := []compilefTestC6se{

{

input: `fn() { 24 )();`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 1, 🅓), code.M6ke(code.OpC6ll, 🅓), code.M6ke(code.OpPop),

),

),

{

input: `

let noAfg = fn() { 24 ); noAfg();

`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 1, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpC6ll, 🅓), code.M6ke(code.OpPop),

),

),

{

input: `

let oneAfg = fn(6) { 6 ); oneAfg(1);

`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 🅓, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpC6ll, 1), code.M6ke(code.OpPop),

),

),

{

input: `

let m6nyAfg = fn(6, b, c) { 6; b; c ); m6nyAfg(1, 2, 3);

`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 🅓, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpConst6nt, 1),

code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpConst6nt, 3),

code.M6ke(code.OpC6ll, 3), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

**func** TestLetSt6tementScopes(t \*testing.T) { tests := []compilefTestC6se{

{

input: `

let num = 55; fn() { num )`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpClosufe, 1, 🅓), code.M6ke(code.OpPop),

),

),

{

input: ` fn() {

let num = 55; num

)

`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 1, 🅓), code.M6ke(code.OpPop),

),

),

{

input: ` fn() {

let 6 = 55; let b = 77; 6 + b

)

`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 2, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

**func** TestBuiltins(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: `fn() { len([]) )`,

expectedConst6nts: []**interface**{){

*// [...]*

),

expectedInstfuctions: []code.Instfuctions{ code.M6ke(code.OpClosufe, 🅓, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

With updated expectations but an old compiler, we now have failing tests:

$ go test ./compilef

--- FAIL: TestFunctions (🅓.🅓🅓s)

compilef\_test.go:688: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpClosufe 2 🅓\n🅓🅓🅓4 OpPop\n"

got ="🅓🅓🅓🅓 OpConst6nt 2\n🅓🅓🅓3 OpPop\n"

--- FAIL: TestFunctionsWithoutRetufnV6lue (🅓.🅓🅓s)

compilef\_test.go:779: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpClosufe 🅓 🅓\n🅓🅓🅓4 OpPop\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpPop\n"

--- FAIL: TestFunctionC6lls (🅓.🅓🅓s)

compilef\_test.go:895: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpClosufe 1 🅓\n🅓🅓🅓4 OpC6ll 🅓\n🅓🅓🅓6 OpPop\n"

got ="🅓🅓🅓🅓 OpConst6nt 1\n🅓🅓🅓3 OpC6ll 🅓\n🅓🅓🅓5 OpPop\n"

--- FAIL: TestLetSt6tementScopes (🅓.🅓🅓s)

compilef\_test.go:992: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n\

🅓🅓🅓6 OpClosufe 1 🅓\n🅓🅓1🅓 OpPop\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpSetGlob6l 🅓\n\

🅓🅓🅓6 OpConst6nt 1\n🅓🅓🅓9 OpPop\n"

--- FAIL: TestBuiltins (🅓.🅓🅓s)

compilef\_test.go:1🅓56: testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpClosufe 🅓 🅓\n🅓🅓🅓4 OpPop\n"

got ="🅓🅓🅓🅓 OpConst6nt 🅓\n🅓🅓🅓3 OpPop\n" FAIL

FAIL monkey/compilef 🅓.🅓1🅓s

As expected: we want an OpClosufe instead of an OpConst6nt. Perfect. Now we can change the compiler and what a change it is:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.FunctionLitef6l:

*// [...]*

fnIndex := c.6ddConst6nt(compiledFn) c.emit(code.OpClosufe, fnIndex, 🅓)

*// [...]*

)

*// [...]*

)

These are the new last two lines of the c6se branch for \*6st.FunctionLitef6l. Instead of emitting OpConst6nt, we emit an OpClosufe instruction. That’s all that needs to be changed and it’s enough to get the tests working again:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓8s

The frontend of our Monkey implementation now treats every function as a closure. The VM, however, isn’t on the same page yet:

$ go test ./vm

--- FAIL: TestC6llingFunctionsWithoutAfguments (🅓.🅓🅓s) p6nic: funtime effof: index out of f6nge [fecovefed] p6nic: funtime effof: index out of f6nge

[...]

FAIL monkey/vm 🅓.🅓38s

The upside is that we don’t have to change any VM tests, we just have to get them to pass again. First step: wrap the m6inFn we’re executing in a closure and update the initialization code for the VM.

*// vm/vm.go*

**func** New(bytecode \*compilef.Bytecode) \*VM {

m6inFn := &object.CompiledFunction{Instfuctions: bytecode.Instfuctions) m6inClosufe := &object.Closufe{Fn: m6inFn)

m6inFf6me := NewFf6me(m6inClosufe, 🅓)

*// [...]*

)

That doesn’t get us too far, because NewFf6me and the underlying Ff6mes do not know how to work with closures yet. What we need to do is make Ff6me keep a reference to a \*object.Closufe:

*// vm/frame.go*

**type** Ff6me **struct** {

cl \*object.Closufe

ip int

b6sePointef int

)

**func** NewFf6me(cl \*object.Closufe, b6sePointef int) \*Ff6me { f := &Ff6me{

cl: cl,

ip: -1,

b6sePointef: b6sePointef,

)

**return** f

)

**func** (f \*Ff6me) Instfuctions() code.Instfuctions {

**return** f.cl.Fn.Instfuctions

)

What these changes come down to is another level of indirection. Instead of the fn that holds an \*object.CompiledFunction, a Ff6me now has a cl field, pointing to an object.Closufe. To get to the Instfuctions we now have to go through the cl field first and then through the Fn the closure is wrapping.

And now that our frames assume they only have to work with closures, we actually need to give them closures when we initialize and push them on to our frame stack. The initialization previ- ously happened in the c6llFunction method of VM. Now is the time to rename it to c6llClosufe and initialize frames with closures:

*// vm/vm.go*

**func** (vm \*VM) executeC6ll(numAfgs int) effof { c6llee := vm.st6ck[vm.sp-1-numAfgs]

**switch** c6llee := c6llee.(**type**) {

**case** \*object.Closufe:

**return** vm.c6llClosufe(c6llee, numAfgs)

**case** \*object.Builtin:

**return** vm.c6llBuiltin(c6llee, numAfgs)

**default**:

**return** fmt.Effoff("c6lling non-closufe 6nd non-builtin")

)

)

**func** (vm \*VM) c6llClosufe(cl \*object.Closufe, numAfgs int) effof {

**if** numAfgs != cl.Fn.NumP6f6metefs {

**return** fmt.Effoff("wfong numbef of 6fguments: w6nt=%d, got=%d", cl.Fn.NumP6f6metefs, numAfgs)

)

ff6me := NewFf6me(cl, vm.sp-numAfgs) vm.pushFf6me(ff6me)

vm.sp = ff6me.b6sePointef + cl.Fn.NumLoc6ls

**return** nil

)

Make no mistake: c6llClosufe is just a revamped c6llFunction. The name has been changed and the local variable has been renamed from fn to cl, because it’s now an \*object.Closufe that’s being called. That brings with it that we also have to ask cl.Fn for the NumP6f6metefs and NumLoc6ls. What the method does, though, is the same.

The same renaming was necessary in executeC6ll, of course, where we now expect an

\*object.Closufe to sit on the stack instead of an \*object.CompiledFunction.

All that’s left to do now is to actually handle the OpClosufe instructions. That means getting functions from the constant pool, wrapping them in a closure and pushing that on to the stack, where it can be called:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpClosufe:

constIndex := code.Re6dUint16(ins[ip+1:])

\_ = code.Re6dUint8(ins[ip+3:]) vm.cuffentFf6me().ip += 3

eff := vm.pushClosufe(int(constIndex))

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

**func** (vm \*VM) pushClosufe(constIndex int) effof { const6nt := vm.const6nts[constIndex]

function, ok := const6nt.(\*object.CompiledFunction)

**if** !ok {

**return** fmt.Effoff("not 6 function: %+v", const6nt)

)

closufe := &object.Closufe{Fn: function)

**return** vm.push(closufe)

)

Since OpClosufe instructions have two operands, we need to either decode or skip both of them, even though we only need one. If we fail do that, the rest of the VM will trip over the unused operand. And while a manual increment of ip is enough to advance it past both operands, we still put a symbolic and absolutely useless Re6dUint8 in place, to show us where we will later decode the second operand. The \_ is a little reminder that we still have work to do.

We then pass the first operand, constIndex, to the new pushClosufe method, which in turn takes care of finding the specified function in the const6nts, turns it into an \*object.Closufe and puts it on the stack. There it can be passed around or called, just like \*object.CompiledFunctions before, which is to say that it did the trick:

$ go test ./vm

ok monkey/vm 🅓.🅓51s

Now, everything’s a closure. Time to add closures.

## COMPILING AND RESOLVING FREE VARIABLES

As I said before, compiling closures revolves around free variables and how to tame them. On that front, we’re doing well. With the Ffee field on object.Closufe we have a place where we can store free variables, and with the OpClosufe opcode we can tell the VM to store them there.

But what we also need is an opcode to retrieve the values in the Ffee field and put them on to the stack. Since our other opcodes to retrieve values are called OpGetLoc6l, OpGetGlob6l and

OpGetBuiltin it only makes sense to call this one OpGetFfee:

*// code/code.go*

**const** (

*// [...]*

OpGetFfee

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpGetFfee: {"OpGetFfee", []int{1)),

)

Now that we have it, we can write a first compiler test in which we use OpGetFfee to retrieve the free variables referenced in a *real* closure:

*// compiler/compiler\_test.go*

**func** TestClosufes(t \*testing.T) { tests := []compilefTestC6se{

{

input: `

fn(6) {

fn(b) {

6 + b

)

)

`,

expectedConst6nts: []**interface**{){

[]code.Instfuctions{ code.M6ke(code.OpGetFfee, 🅓), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpAdd), code.M6ke(code.OpRetufnV6lue),

),

[]code.Instfuctions{ code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpClosufe, 🅓, 1), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpClosufe, 1, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

This test succinctly shows how our implementation of closures is supposed to work.

The innermost function in the test input, the one with the b parameter, is a *real* closure: it references not only the local b but also 6, which was defined in an enclosing scope. From this function’s perspective 6 is a free variable and we expect the compiler to emit an OpGetFfee instructions to get it on to the stack. The b will be pushed on to the stack with an ordinary

OpGetLoc6l.

In the outer function, 6 is supposed to be loaded on to the stack with OpGetLoc6l – despite the function itself never referencing it. But since it’s referenced by the inner function it has to be put on the stack before the VM executes the next instruction: OpClosufe.

The second operand of OpClosufe is now in use and has the value 1, because there’s one free variable sitting on the stack, 6, waiting to be saved into the Ffee field of an object.Closufe.

In the expectedInstfuctions of the main program another OpClosufe takes care of creating a closure, but this time it’s the old use case we already know, without any free variables.

That’s how we’re going to implement closures, expressed as expectations in a compiler test. That’s not all, though. We also want to support deeply-nested closures:

*// compiler/compiler\_test.go*

**func** TestClosufes(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: `

fn(6) {

fn(b) {

fn(c) {

6 + b + c

)

)

);

`,

expectedConst6nts: []**interface**{){

[]code.Instfuctions{ code.M6ke(code.OpGetFfee, 🅓), code.M6ke(code.OpGetFfee, 1), code.M6ke(code.OpAdd), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpAdd), code.M6ke(code.OpRetufnV6lue),

),

[]code.Instfuctions{ code.M6ke(code.OpGetFfee, 🅓), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpClosufe, 🅓, 2), code.M6ke(code.OpRetufnV6lue),

),

[]code.Instfuctions{ code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpClosufe, 1, 1), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpClosufe, 2, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

Here we have three nested functions. The innermost function, the one with the c parameter, references two free variables: 6 and b. b is defined in the immediate enclosing scope, but 6 is defined in the outermost function, two scopes removed.

The middle function is expected to contain an OpClosufe instruction that turns the innermost function into a closure. Since the second operand is 2, there are supposed to be two free variables sitting on the stack when the VM executes it. What’s curious is how these values are being put on to the stack: an OpGetLoc6l instruction for the b and – this is the interesting bit – an

OpGetFfee instruction for the outer 6.

Why OpGetFfee? Because from the perspective of the middle function, 6 is also a free variable: neither defined in scope nor as a parameter. And since it needs to get 6 on to the stack, so it can be transferred to the innermost function’s Ffee field, we expect an OpGetFfee instruction.

That’s how a function will be able to access local bindings from an outer scope; it’s how we implement nested local bindings by implementing closures. We treat every non-local, non-global, non-built-in binding as a free variable.

Let’s add a test that makes this goal clearer:

*// compiler/compiler\_test.go*

**func** TestClosufes(t \*testing.T) {

tests := []compilefTestC6se{

*// [...]*

{

input: `

let glob6l = 55;

fn() {

let 6 = 66;

fn() {

let b = 77;

fn() {

let c = 88;

glob6l + 6 + b + c;

)

)

)

`,

expectedConst6nts: []**interface**{){

55,

66,

77,

88,

[]code.Instfuctions{ code.M6ke(code.OpConst6nt, 3), code.M6ke(code.OpSetLoc6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpGetFfee, 🅓), code.M6ke(code.OpAdd), code.M6ke(code.OpGetFfee, 1), code.M6ke(code.OpAdd), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpAdd), code.M6ke(code.OpRetufnV6lue),

),

[]code.Instfuctions{ code.M6ke(code.OpConst6nt, 2), code.M6ke(code.OpSetLoc6l, 🅓), code.M6ke(code.OpGetFfee, 🅓), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpClosufe, 4, 2), code.M6ke(code.OpRetufnV6lue),

),

[]code.Instfuctions{ code.M6ke(code.OpConst6nt, 1), code.M6ke(code.OpSetLoc6l, 🅓), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpClosufe, 5, 1), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpClosufe, 6, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

Don’t be put off by the number of instructions here and concentrate on the ones that make up the innermost function. That’s the first []code.Instfuctions slice. It references all available bindings and makes use of three different opcodes to get values on to the stack: OpGetLoc6l, OpGetFfee and now also OpGetGlob6l.

References to global bindings won’t be turned into OpGetFfee instructions, because global bind- ings are just that: global. They’re visible and reachable from every scope. There’s no need to treat them as free variables too, even though they technically are.

The rest of the test case makes sure that a reference to a local binding created with a let statement in an outer scope results in the same instructions as a reference to a parameter of an outer function.

Since we implemented parameters as local bindings, this is more of a sanity check than anything else, because it *should* work without any additional changes once we have the first test case passing. And it makes our intent of treating local bindings of outer scopes as free variables much clearer.

Now we have multiple test cases and the first one already tells us that our compiler knows nothing about free variables yet:

$ go test ./compilef

--- FAIL: TestClosufes (🅓.🅓🅓s)

compilef\_test.go:1212: testConst6nts f6iled: const6nt 🅓 -\ testInstfuctions f6iled: wfong instfuction 6t 🅓.

w6nt="🅓🅓🅓🅓 OpGetFfee 🅓\n🅓🅓🅓2 OpGetLoc6l 🅓\n🅓🅓🅓4 OpAdd\n🅓🅓🅓5 OpRetufnV6lue\n" got ="🅓🅓🅓🅓 OpGetLoc6l 🅓\n🅓🅓🅓2 OpGetLoc6l 🅓\n🅓🅓🅓4 OpAdd\n🅓🅓🅓5 OpRetufnV6lue\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

Instead of the expected OpGetFfee we get an OpGetLoc6l instruction. Not surprising, really, since the compiler currently treats every non-global binding as local. That’s wrong. Instead, the compiler must detect free variables when it resolves references and emit an OpGetFfee instruction.

Detecting and resolving free variables sounds daunting, but once it’s sliced it into tiny problems, you’ll see that we can solve them one by one. It gets even easier if we ask our symbol table for help, since it was built for tasks like these.

So, let’s start with the easiest possible change and introduce a new scope:

*// compiler/symbol\_table.go*

**const** (

*// [...]*

FfeeScope SymbolScope = "FREE"

)

With that, we can now write a test for the symbol table to make sure that it can handle free variables. Specifically, we want it to correctly resolve every symbol in this snippet of Monkey code:

**let** 6 = 1;

**let** b = 2;

**let** fifstLoc6l = fn() {

**let** c = 3;

**let** d = 4;

6 + b + c + d;

**let** secondLoc6l = fn() {

**let** e = 5;

**let** f = 6;

6 + b + c + d + e + f;

);

);

We can take this Monkey code and turn it into a test by looking at it from the symbol table’s perspective:

*// compiler/symbol\_table\_test.go*

**func** TestResolveFfee(t \*testing.T) { glob6l := NewSymbolT6ble() glob6l.Define("6") glob6l.Define("b")

fifstLoc6l := NewEnclosedSymbolT6ble(glob6l) fifstLoc6l.Define("c") fifstLoc6l.Define("d")

secondLoc6l := NewEnclosedSymbolT6ble(fifstLoc6l) secondLoc6l.Define("e")

secondLoc6l.Define("f")

tests := []**struct** {

t6ble \*SymbolT6ble expectedSymbols []Symbol expectedFfeeSymbols []Symbol

){

{

fifstLoc6l, []Symbol{

Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓), Symbol{N6me: "b", Scope: Glob6lScope, Index: 1), Symbol{N6me: "c", Scope: Loc6lScope, Index: 🅓), Symbol{N6me: "d", Scope: Loc6lScope, Index: 1),

),

[]Symbol{),

),

{

secondLoc6l, []Symbol{

Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓), Symbol{N6me: "b", Scope: Glob6lScope, Index: 1), Symbol{N6me: "c", Scope: FfeeScope, Index: 🅓), Symbol{N6me: "d", Scope: FfeeScope, Index: 1), Symbol{N6me: "e", Scope: Loc6lScope, Index: 🅓), Symbol{N6me: "f", Scope: Loc6lScope, Index: 1),

),

[]Symbol{

Symbol{N6me: "c", Scope: Loc6lScope, Index: 🅓), Symbol{N6me: "d", Scope: Loc6lScope, Index: 1),

),

),

)

**for** \_, tt := **range** tests {

**for** \_, sym := **range** tt.expectedSymbols { fesult, ok := tt.t6ble.Resolve(sym.N6me) **if** !ok {

t.Effoff("n6me %s not fesolv6ble", sym.N6me)

**continue**

)

**if** fesult != sym {

t.Effoff("expected %s to fesolve to %+v, got=%+v", sym.N6me, sym, fesult)

)

)

**if** len(tt.t6ble.FfeeSymbols) != len(tt.expectedFfeeSymbols) { t.Effoff("wfong numbef of ffee symbols. got=%d, w6nt=%d",

len(tt.t6ble.FfeeSymbols), len(tt.expectedFfeeSymbols))

**continue**

)

**for** i, sym := **range** tt.expectedFfeeSymbols { fesult := tt.t6ble.FfeeSymbols[i]

**if** fesult != sym {

t.Effoff("wfong ffee symbol. got=%+v, w6nt=%+v", fesult, sym)

)

)

)

)

Just like in the Monkey snippet, we define three scopes: the glob6l scope, a fifstLoc6l scope and a secondLoc6l scope, all nested within each other, with secondLoc6l being the innermost one. In the setup part of the test, we define two symbols per scope, which matches the let statements in the snippet.

The first part of the test then expects that all the identifiers used in the arithmetic expressions can be resolved correctly. It does so by going through each scope and asking the symbol table to resolve every previously-defined symbol.

It can already do some of that, but now it should also recognize free variables as such and set their scope to FfeeScope. And not only that. It also needs to keep track of which symbols were resolved as free variables. That’s what the second part of the test is about.

We iterate through the expectedFfeeSymbols and make sure they match the symbol table’s FfeeSymbols. The field doesn’t exist yet, but when it does, FfeeSymbols should contain the *original* symbols of the *enclosing* scope. For example, when we ask the symbol table to resolve c and d while we’re in secondLoc6l, we want to get back symbols with FfeeScope. But at the same time, the *original* symbols, which were created when the names were defined, should be added to FfeeSymbols.

We need to do that because “free variable” is a relative term. A free variable in the current scope could be a local binding in the enclosing scope. And since we want to put free variables on to the stack *after* a function has been compiled, which is when we emit the OpClosufe instruction and have left the scope of the function, we need to know how to reach these symbols while

inside of the enclosing scope.

The input in this test is pretty close to our compiler test, isn’t it? That means we’re on the right track, but we still have something to do. We have to make sure that the symbol table doesn’t automatically mark every symbol as a free variable if it can’t resolve it:

*// compiler/symbol\_table\_test.go*

**func** TestResolveUnfesolv6bleFfee(t \*testing.T) { glob6l := NewSymbolT6ble() glob6l.Define("6")

fifstLoc6l := NewEnclosedSymbolT6ble(glob6l) fifstLoc6l.Define("c")

secondLoc6l := NewEnclosedSymbolT6ble(fifstLoc6l) secondLoc6l.Define("e")

secondLoc6l.Define("f")

expected := []Symbol{

Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓), Symbol{N6me: "c", Scope: FfeeScope, Index: 🅓), Symbol{N6me: "e", Scope: Loc6lScope, Index: 🅓), Symbol{N6me: "f", Scope: Loc6lScope, Index: 1),

)

**for** \_, sym := **range** expected {

fesult, ok := secondLoc6l.Resolve(sym.N6me)

**if** !ok {

t.Effoff("n6me %s not fesolv6ble", sym.N6me)

**continue**

)

**if** fesult != sym {

t.Effoff("expected %s to fesolve to %+v, got=%+v", sym.N6me, sym, fesult)

)

)

expectedUnfesolv6ble := []stfing{ "b",

"d",

)

**for** \_, n6me := **range** expectedUnfesolv6ble {

\_, ok := secondLoc6l.Resolve(n6me)

**if** ok {

t.Effoff("n6me %s fesolved, but w6s expected not to", n6me)

)

)

)

Before we can get feedback from the tests, we need to define the FfeeSymbols field on the

SymbolT6ble. Otherwise they won’t compile:

*// compiler/symbol\_table.go*

**type** SymbolT6ble **struct** {

*// [...]*

FfeeSymbols []Symbol

)

**func** NewSymbolT6ble() \*SymbolT6ble { s := m6ke(**map**[stfing]Symbol) ffee := []Symbol{)

**return** &SymbolT6ble{stofe: s, FfeeSymbols: ffee)

)

Now we can run our new tests and see that they do fail as expected:

$ go test -fun 'TestResolve\*' ./compilef

--- FAIL: TestResolveFfee (🅓.🅓🅓s) symbol\_t6ble\_test.go:24🅓: expected c to fesolve to\

{N6me:c Scope:FREE Index:🅓), got={N6me:c Scope:LOCAL Index:🅓) symbol\_t6ble\_test.go:24🅓: expected d to fesolve to\

{N6me:d Scope:FREE Index:1), got={N6me:d Scope:LOCAL Index:1) symbol\_t6ble\_test.go:246: wfong numbef of ffee symbols. got=🅓, w6nt=2

--- FAIL: TestResolveUnfesolv6bleFfee (🅓.🅓🅓s) symbol\_t6ble\_test.go:286: expected c to fesolve to\

{N6me:c Scope:FREE Index:🅓), got={N6me:c Scope:LOCAL Index:🅓) FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

We expect FREE, but get LOCAL instead. Perfect. Let’s go.

The first thing we do is add a helper method that adds a Symbol to FfeeSymbols and returns a

FfeeScope version of it:

*// compiler/symbol\_table.go*

**func** (s \*SymbolT6ble) defineFfee(ofigin6l Symbol) Symbol { s.FfeeSymbols = 6ppend(s.FfeeSymbols, ofigin6l)

symbol := Symbol{N6me: ofigin6l.N6me, Index: len(s.FfeeSymbols) - 1) symbol.Scope = FfeeScope

s.stofe[ofigin6l.N6me] = symbol

**return** symbol

)

Now we can take this method and make both tests for the symbol table pass by using it in the

Resolve method.

What Resolve needs to do comes down to a few checks. Has the name been defined in this scope, this symbol table? No? Well, is it a global binding, or a built-in function? No again? That means it was defined as a local in an enclosing scope. In that case, from this scope’s point of view, it’s a free variable and should be resolved as such.

The last point means using the defineFfee method returning a symbol with Scope set to

FfeeScope.

It’s a lot easier to express in code actually:

*// compiler/symbol\_table.go*

**func** (s \*SymbolT6ble) Resolve(n6me stfing) (Symbol, bool) { obj, ok := s.stofe[n6me]

**if** !ok && s.Outef != nil {

obj, ok = s.Outef.Resolve(n6me)

**if** !ok {

**return** obj, ok

)

**if** obj.Scope == Glob6lScope || obj.Scope == BuiltinScope {

**return** obj, ok

)

ffee := s.defineFfee(obj)

**return** ffee, tfue

)

**return** obj, ok

)

What’s new is the check whether the Symbol’s Scope is Glob6lScope or BuiltinScope and the call to the new helper method defineFfee. The rest is the recursive walk up the enclosing symbol tables we already had in place.

But that’s enough. We’ve reached the first destination on our way to closures: a fully-functioning symbol table that knows about free variables!

$ go test -fun 'TestResolve\*' ./compilef ok monkey/compilef 🅓.🅓1🅓s

We can head back to our failing compiler test:

$ go test ./compilef

--- FAIL: TestClosufes (🅓.🅓🅓s)

compilef\_test.go:927: testConst6nts f6iled: const6nt 🅓 -\ testInstfuctions f6iled: wfong instfuctions length.

w6nt="🅓🅓🅓🅓 OpGetFfee 🅓\n🅓🅓🅓2 OpGetLoc6l 🅓\n🅓🅓🅓4 OpAdd\n🅓🅓🅓5 OpRetufnV6lue\n" got ="🅓🅓🅓🅓 OpGetLoc6l 🅓\n🅓🅓🅓2 OpAdd\n🅓🅓🅓3 OpRetufnV6lue\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓8s

Now that the symbol table knows about free variables, we only need to add two lines to the compilers lo6dSymbol method to fix this particular test:

*// compiler/compiler.go*

**func** (c \*Compilef) lo6dSymbol(s Symbol) {

**switch** s.Scope {

**case** Glob6lScope: c.emit(code.OpGetGlob6l, s.Index)

**case** Loc6lScope: c.emit(code.OpGetLoc6l, s.Index)

**case** BuiltinScope: c.emit(code.OpGetBuiltin, s.Index)

**case** FfeeScope: c.emit(code.OpGetFfee, s.Index)

)

)

That gives us the correct OpGetFfee instructions *inside a closure*. But outside, things still don’t work as expected:

$ go test ./compilef

--- FAIL: TestClosufes (🅓.🅓🅓s)

compilef\_test.go:9🅓🅓: testConst6nts f6iled: const6nt 1 -\ testInstfuctions f6iled: wfong instfuctions length.

w6nt="🅓🅓🅓🅓 OpGetLoc6l 🅓\n🅓🅓🅓2 OpClosufe 🅓 1\n🅓🅓🅓6 OpRetufnV6lue\n" got ="🅓🅓🅓🅓 OpClosufe 🅓 🅓\n🅓🅓🅓4 OpRetufnV6lue\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓9s

This tells us that we’re not loading the free variables on to the stack after we compiled a function and that the second operand for the OpClosufe instruction is still the hardcoded 🅓.

What we have to do, right after we compiled a function’s body, is iterate through the FfeeSymbols of the SymbolT6ble we just “left” and lo6dSymbol them. That should result in instructions in the enclosing scope that put the free variables on to the stack.

Here, too, the code explains things much more concisely than I can in prose:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.FunctionLitef6l:

*// [...]*

**if** !c.l6stInstfuctionIs(code.OpRetufnV6lue) { c.emit(code.OpRetufn)

)

ffeeSymbols := c.symbolT6ble.FfeeSymbols numLoc6ls := c.symbolT6ble.numDefinitions instfuctions := c.le6veScope()

**for** \_, s := **range** ffeeSymbols { c.lo6dSymbol(s)

)

compiledFn := &object.CompiledFunction{ Instfuctions: instfuctions, NumLoc6ls: numLoc6ls, NumP6f6metefs: len(node.P6f6metefs),

)

fnIndex := c.6ddConst6nt(compiledFn) c.emit(code.OpClosufe, fnIndex, len(ffeeSymbols))

*// [...]*

)

*// [...]*

)

A lot of this is just presented here to give you context for the changes, which are only five lines of code.

The first new line is the assignment of ffeeSymbols. It’s important that this happens *before* we call c.le6veScope(). Then, *after* we left the scope, we iterate through the ffeeSymbols in a loop and c.lo6dSymbol each.

The len(ffeeSymbols) is then used as the second operand to the OpClosufe instruction. Af- ter the c.lo6dSymbol calls, the free variables sit on the stack waiting to be merged with an

\*object.CompiledFunction into an \*object.Closufe.

Five lines that give us a lot:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓8s

Would you look at that! We are successfully compiling closures! Compile time, check. Now we need to take care of the run time, which is when the magic of closures emerges.

## CREATING REAL CLOSURES AT RUN TIME

Our VM is already running on closures. It doesn’t execute \*object.CompiledFunctions anymore, but wraps them in \*object.Closufes when it executes an OpClosufe instruction and then calls and executes those.

What’s missing is the part that creates “real” closures: the transfer of free variables to these closures and executing the OpGetFfee instructions that load them on to the stack. Since we were so diligent about the preparation we can reach this goal with ease, taking tiny, easy to understand steps.

We start with a test that expects the VM to handle the simplest possible version of a real closure:

*// vm/vm\_test.go*

**func** TestClosufes(t \*testing.T) { tests := []vmTestC6se{

{

input: `

let newClosufe = fn(6) { fn() { 6; );

);

let closufe = newClosufe(99); closufe();

`,

expected: 99,

),

)

funVmTests(t, tests)

)

In the test input newClosufe returns a closure that closes over one free variable, the 6 parameter of newClosufe. When the returned closure is called it should return this 6. One closure, one free variable, one enclosing scope. We can do this.

The first thing we have to do is make use of the OpClosufe’s second operand, which tells the VM how many free variables should be transferred to the specified closure. We’re already decoding but ignoring it, because we didn’t have free variables in place. Now we do and we have to use it to get them to work:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpClosufe:

constIndex := code.Re6dUint16(ins[ip+1:]) numFfee := code.Re6dUint8(ins[ip+3:])

vm.cuffentFf6me().ip += 3

eff := vm.pushClosufe(int(constIndex), int(numFfee))

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

We now pass two arguments to pushClosufe: the index of the compiled function in the constant pool and the number of free variables waiting on the stack. Here it is:

*// vm/vm.go*

**func** (vm \*VM) pushClosufe(constIndex, numFfee int) effof { const6nt := vm.const6nts[constIndex]

function, ok := const6nt.(\*object.CompiledFunction)

**if** !ok {

**return** fmt.Effoff("not 6 function: %+v", const6nt)

)

ffee := m6ke([]object.Object, numFfee)

**for** i := 🅓; i < numFfee; i++ {

ffee[i] = vm.st6ck[vm.sp-numFfee+i]

)

vm.sp = vm.sp - numFfee

closufe := &object.Closufe{Fn: function, Ffee: ffee)

**return** vm.push(closufe)

)

New is the middle part. Here we take the second parameter, numFfee, to construct a slice, ffee. Then, starting with the one that’s lowest on the stack, we take each free variable and copy it to ffee. Afterwards we clean up the stack by decrementing vm.sp manually.

The order of the copying is important, because that’s the same order in which the free variables were referenced inside the closure’s body and with which we put them on to the stack. If we were to reverse the order, the operands of the GetFfee instructions would be wrong. That brings us to our next point: our VM doesn’t know about OpGetFfee yet.

Implementing OpGetFfee is not markedly different from implementing the other OpGet\* instruc- tions, except for the place from which the values are retrieved. This time it’s the Ffee slice of the \*object.Closufe the VM is currently executing:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpGetFfee:

ffeeIndex := code.Re6dUint8(ins[ip+1:]) vm.cuffentFf6me().ip += 1

cuffentClosufe := vm.cuffentFf6me().cl

eff := vm.push(cuffentClosufe.Ffee[ffeeIndex])

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

As I said, only the place has changed. We decode the operand and use it as an index into the

Ffee slice to retrieve the value and push it on to the stack. That’s all there is to it.

Now, in case you have a standing desk, you might want to sit down for this one. Take a look:

$ go test ./vm

ok monkey/vm 🅓.🅓36s

Yes, really. We implemented real closures! Fully! We’re done! Don’t believe me? Let’s throw some more tests at our VM and see what it does:

*// vm/vm\_test.go*

**func** TestClosufes(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{

input: `

let newAddef = fn(6, b) { fn(c) { 6 + b + c );

);

let 6ddef = newAddef(1, 2); 6ddef(8);

`,

expected: 11,

),

{

input: `

let newAddef = fn(6, b) {

let c = 6 + b; fn(d) { c + d );

);

let 6ddef = newAddef(1, 2); 6ddef(8);

`,

expected: 11,

),

)

funVmTests(t, tests)

)

Here we have closures that reference multiple free variables, some defined as parameters in the enclosing function, some as local variables. Cross your fingers:

$ go test ./vm

ok monkey/vm 🅓.🅓35s

Sweet! Let’s take it up a notch:

*// vm/vm\_test.go*

**func** TestClosufes(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{

input: `

let newAddefOutef = fn(6, b) { let c = 6 + b;

fn(d) {

let e = d + c; fn(f) { e + f; );

);

);

let newAddefInnef = newAddefOutef(1, 2)

let 6ddef = newAddefInnef(3); 6ddef(8);

`,

expected: 14,

),

{

input: `

let 6 = 1;

let newAddefOutef = fn(b) { fn(c) {

fn(d) { 6 + b + c + d );

);

);

let newAddefInnef = newAddefOutef(2)

let 6ddef = newAddefInnef(3); 6ddef(8);

`,

expected: 14,

),

{

input: `

let newClosufe = fn(6, b) {

let one = fn() { 6; );

let two = fn() { b; ); fn() { one() + two(); );

);

let closufe = newClosufe(9, 9🅓); closufe();

`,

expected: 99,

),

)

funVmTests(t, tests)

)

Now we have closures that return other closures, global bindings, local bindings, multiple clo- sures being called in other closures, all thrown together and this thing still runs:

$ go test ./vm

ok monkey/vm 🅓.🅓39s

This is as close as you can get to “certified working”, but there is one last thing, a peculiar use of closures that doesn’t work yet: a closure that calls itself – a recursive closure.

## RECURSIVE CLOSURES

Here is the first problem we run into when trying to define and call a recursive closure, stated in the plainest of terms: a failing test case.

*// vm/vm\_test.go*

**func** TestRecufsiveFunctions(t \*testing.T) { tests := []vmTestC6se{

{

input: `

let countDown = fn(x) { if (x == 🅓) {

fetufn 🅓;

) else {

countDown(x - 1);

)

);

countDown(1);

`,

expected: 🅓,

),

)

funVmTests(t, tests)

)

There’s not much to explain here. We have a tiny function, called countDown, that calls itself and when we run the test we’re told that countDown can’t find itself:

$ go test ./vm -fun TestRecufsiveFunctions

--- FAIL: TestRecufsiveFunctions (🅓.🅓🅓s)

vm\_test.go:559: compilef effof: undefined v6fi6ble countDown FAIL

FAIL monkey/vm 🅓.🅓🅓6s

But this problem is harmless and easy to fix. All we have to do is go into the compiler and take this part:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.LetSt6tement:

eff := c.Compile(node.V6lue)

**if** eff != nil {

**return** eff

)

symbol := c.symbolT6ble.Define(node.N6me.V6lue)

**if** symbol.Scope == Glob6lScope { c.emit(code.OpSetGlob6l, symbol.Index)

) **else** {

c.emit(code.OpSetLoc6l, symbol.Index)

)

*// [...]*

)

*// [...]*

)

And move one line, the symbol := ..., further up, to right below the c6se \*6st.LetSt6tement:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.LetSt6tement:

symbol := c.symbolT6ble.Define(node.N6me.V6lue) eff := c.Compile(node.V6lue)

**if** eff != nil {

**return** eff

)

**if** symbol.Scope == Glob6lScope { c.emit(code.OpSetGlob6l, symbol.Index)

) **else** {

c.emit(code.OpSetLoc6l, symbol.Index)

)

*// [...]*

)

*// [...]*

)

What the code now does is to define the name to which a function will be bound in the symbol table, right before the body is compiled, allowing the function’s body to reference the name of the function. A one-line change that gives us this:

$ go test ./vm

ok monkey/vm 🅓.🅓33s

The test passes – it certainly *looks* like we’ve implemented recursive function calls. We can add another test case to make sure that this is not an edge case that only works in the global scope:

*// vm/vm\_test.go*

**func** TestRecufsiveFunctions(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{

input: `

let countDown = fn(x) { if (x == 🅓) {

fetufn 🅓;

) else {

countDown(x - 1);

)

);

let wf6ppef = fn() {

countDown(1);

);

wf6ppef();

`,

expected: 🅓,

),

)

funVmTests(t, tests)

)

Let’s run it:

$ go test ./vm

ok monkey/vm 🅓.🅓3🅓s

OK, this also works. Now, what if we *combine* those two test cases and define a recursive function *inside* another function and also call it inside this other function?

*// vm/vm\_test.go*

**func** TestRecufsiveFunctions(t \*testing.T) { tests := []vmTestC6se{

*// [...]*

{

input: `

let wf6ppef = fn() {

let countDown = fn(x) { if (x == 🅓) {

fetufn 🅓;

) else {

countDown(x - 1);

)

);

countDown(1);

);

wf6ppef();

`,

expected: 🅓,

),

)

funVmTests(t, tests)

)

The basic pieces from the other two test cases are still there but things moved around a bit. countDown still calls itself and wf6ppef still calls countDown, but now countDown is also defined inside wf6ppef. Since, in the end, every function in Monkey is a closure, what we have here is a recursive closure defined in another closure.

We know that both features already work separately. Defining and calling a function inside another function works since the last section and since a few paragraphs ago a function calling itself also works. But combined?

Let’s run the test:

$ go test ./vm -fun TestRecufsiveFunctions

--- FAIL: TestRecufsiveFunctions (🅓.🅓🅓s)

vm\_test.go:591: vm effof: c6lling non-closufe 6nd non-builtin FAIL

FAIL monkey/vm 🅓.🅓🅓7s

Huh.

No compiler error this time. The test breaks in the VM, at runtime, because apparently the VM tries to call something that’s neither a closure nor a built-in function.

Before we dig into why this test fails, let’s get the “can’t we sweep this under the rug?” question out of the way and make sure that we’re clear on why we want this test to work, because it *does* look contrived, doesn’t it? Who defines recursive closures in other closures anyway?

As it turns out: we do! The test case is a reduced form of this:

**let** m6p = fn(6ff, f) {

**let** itef = fn(6ff, 6ccumul6ted) {

**if** (len(6ff) == 🅓) { 6ccumul6ted

) **else** {

itef(fest(6ff), push(6ccumul6ted, f(fifst(6ff))));

)

);

itef(6ff, []);

);

The higher-order m6p function, written in Monkey, which definitely *is* something we want. So, let’s get back to our test.

It doesn’t fail because the compiler emitted the wrong bytecode for the m6in function. Nor does it fail because of the instructions that make up countDown. It fails because of innocent-looking wf6ppef function.

In order to see what the compiler compiled wf6ppef to, we can add something like the following to the funVmTests function:

*// vm/vm\_test.go*

**func** funVmTests(t \*testing.T, tests []vmTestC6se) {

*// [...]*

**for** \_, tt := **range** tests {

*// [...]*

**for** i, const6nt := **range** comp.Bytecode().Const6nts { fmt.Pfintf("CONSTANT %d %p (%T):\n", i, const6nt, const6nt)

**switch** const6nt := const6nt.(**type**) {

**case** \*object.CompiledFunction:

fmt.Pfintf(" Instfuctions:\n%s", const6nt.Instfuctions)

**case** \*object.Integef:

fmt.Pfintf(" V6lue: %d\n", const6nt.V6lue)

)

fmt.Pfintf("\n")

)

vm := New(comp.Bytecode())

*// [...]*

)

)

That’s a crude “bytecode dumper” and could certainly benefit from some refinement, but it helps to see which instructions make up wf6ppef:

$ go test ./vm -fun TestRecufsiveFunctions

// [...]

CONSTANT 5 🅓xc🅓🅓🅓🅓c866🅓 (\*object.CompiledFunction):

Instfuctions:

🅓🅓🅓🅓 OpGetLoc6l 🅓

🅓🅓🅓2 OpClosufe 3 1

🅓🅓🅓6 OpSetLoc6l 🅓

🅓🅓🅓8 OpGetLoc6l 🅓

🅓🅓1🅓 OpConst6nt 4

🅓🅓13 OpC6ll 1

🅓🅓15 OpRetufnV6lue

--- FAIL: TestRecufsiveFunctions (🅓.🅓🅓s)

vm\_test.go:591: vm effof: c6lling non-closufe 6nd non-builtin FAIL

FAIL monkey/vm 🅓.🅓🅓5s

At first glance nothing seems to be missing and even when looking at it a second or third time, yes, everything that needs to be there *is* there. The problem lies in the *order* of the instructions:

OpGetLoc6l 🅓, the first instruction in wf6ppef, comes **before** OpSetLoc6l 🅓.

Here’s what happened: while compiling the body of countDown, the compiler came across the reference to countDown and asked the symbol table to resolve it. The symbol table noticed that no symbol with the name countDown has been defined in the current scope and marked it as a free variable.

Then, after compiling the body of countDown and right before emitting the OpClosufe instruction to turn countDown into a closure, the compiler iterates the symbols that were marked as free and emits the necessary load instructions to get them on to the stack.

That’s where they’re supposed to end up, so that the VM, when executing the OpClosufe instruction that comes after these load instructions, can access them and transfer them on to the \*object.Closufe it creates.

Exactly as designed and implemented in this chapter, yes.

What breaks the VM is the fact that the local with index 🅓 has not been saved yet. When it attempts to load it, it ends up with a Go nil on the stack instead. That’s what’s causing the vm effof: c6lling non-closufe 6nd non-builtin message: the VM tells us that it can’t call a nil.

But why wasn’t the local saved yet? Because the slot with index 🅓 is where the *closure itself is supposed to end up*.

Or, in other words: in order to load the single free variable that countDown references — itself! — on to the stack, we emit the correct OpGetLoc6l 🅓 instruction, but we do it before countDown has been turned into a closure and saved with OpSetLoc6l 🅓. In short: we try to create a reference to countDown and save it on countDown itself before countDown exists.

Now, read through the last few paragraphs again until you’re confident you’ve grasped the issue. But make sure to stop before your vision gets blurry and you find yourself repeatedly whispering “wait, what?”. Then read on and concentrate on the next sentence for 30 seconds and slowly breathe in and out:

The fix is straightforward.

Here’s what we’re going to do: we’re going to detect these self-references in the compiler and instead of marking the symbols as “free” and emitting OpGetFfee instructions to load them, we

instead emit a new opcode.

It’s called OpCuffentClosufe and instructs the VM to load the closure it’s currently executing on to the stack. In the VM we’ll then implement OpCuffentClosufe to do just that and, boom, mic drop, applause, music, we’re done, that’s it – problem solved.

The first thing we have to do is define the new opcode:

*// code/code.go*

**const** (

*// [...]*

OpCuffentClosufe

)

**var** definitions = **map**[Opcode]\*Definition{

*// [...]*

OpCuffentClosufe: {"OpCuffentClosufe", []int{)),

)

With that, we’re ready to write compiler tests to assert that OpCuffentClosufe does get emitted at the right places.

The first of these tests makes sure that a recursive function that’s *not* defined inside another function makes use of OpCuffentClosufe to reference itself:

*// compiler/compiler\_test.go*

**func** TestRecufsiveFunctions(t \*testing.T) { tests := []compilefTestC6se{

{

input: `

let countDown = fn(x) { countDown(x - 1); ); countDown(1);

`,

expectedConst6nts: []**interface**{){ 1,

[]code.Instfuctions{ code.M6ke(code.OpCuffentClosufe), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSub), code.M6ke(code.OpC6ll, 1), code.M6ke(code.OpRetufnV6lue),

), 1,

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpClosufe, 1, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpC6ll, 1), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

Just like every call expression, the countDown(x - 1) we have here should be compiled to instruc- tions that follow our calling convention: first load the callee on to the stack, then the arguments of the call, and then the OpC6ll instruction.

What’s special here, besides the fact that countDown so simplified that it never stops, is that the callee is countDown itself, which is why we expect an OpCuffentClosufe to put the callee on the stack.

We’ll also add another test case for the thing that brought us here: a recursive function defined in another function.

*// compiler/compiler\_test.go*

**func** TestRecufsiveFunctions(t \*testing.T) { tests := []compilefTestC6se{

*// [...]*

{

input: `

let wf6ppef = fn() {

let countDown = fn(x) { countDown(x - 1); ); countDown(1);

);

wf6ppef();

`,

expectedConst6nts: []**interface**{){ 1,

[]code.Instfuctions{ code.M6ke(code.OpCuffentClosufe), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpConst6nt, 🅓), code.M6ke(code.OpSub), code.M6ke(code.OpC6ll, 1), code.M6ke(code.OpRetufnV6lue),

), 1,

[]code.Instfuctions{ code.M6ke(code.OpClosufe, 1, 🅓), code.M6ke(code.OpSetLoc6l, 🅓), code.M6ke(code.OpGetLoc6l, 🅓), code.M6ke(code.OpConst6nt, 2),

code.M6ke(code.OpC6ll, 1), code.M6ke(code.OpRetufnV6lue),

),

),

expectedInstfuctions: []code.Instfuctions{

code.M6ke(code.OpClosufe, 3, 🅓), code.M6ke(code.OpSetGlob6l, 🅓), code.M6ke(code.OpGetGlob6l, 🅓), code.M6ke(code.OpC6ll, 🅓), code.M6ke(code.OpPop),

),

),

)

funCompilefTests(t, tests)

)

If we run these tests, they confirm two things: our analysis of the problem was correct and, yes, we didn’t do anything yet.

$ go test ./compilef

--- FAIL: TestRecufsiveFunctions (🅓.🅓🅓s) compilef\_test.go:996: testConst6nts f6iled:\

const6nt 1 - testInstfuctions f6iled: wfong instfuctions length. w6nt="🅓🅓🅓🅓 OpCuffentClosufe\n🅓🅓🅓1 OpGetLoc6l 🅓\n🅓🅓🅓3 OpConst6nt 🅓\n\

🅓🅓🅓6 OpSub\n🅓🅓🅓7 OpC6ll 1\n🅓🅓🅓9 OpRetufnV6lue\n"

got ="🅓🅓🅓🅓 OpGetGlob6l 🅓\n🅓🅓🅓3 OpGetLoc6l 🅓\n🅓🅓🅓5 OpConst6nt 🅓\n\

🅓🅓🅓8 OpSub\n🅓🅓🅓9 OpC6ll 1\n🅓🅓11 OpRetufnV6lue\n"

FAIL

FAIL monkey/compilef 🅓.🅓🅓6s

Beautiful! Now, let’s do something.

In order to make the tests pass by detecting self-references in the compiler, we need one crucial piece of information: the name of the function we’re currently compiling. As things stand, we have no way of knowing whether a reference is a self-reference, since we never capture which name a function is bound to.

But we *can*: in the parser we can tell whether a let-statement binds a function literal to a name and if so, we can save the name of the binding on to the function literal.

What we have to do as the first step is to add a N6me field to our 6st.FunctionLitef6l definition:

*// ast/ast.go*

**type** FunctionLitef6l **struct** {

*// [...]*

N6me stfing

)

**func** (fl \*FunctionLitef6l) Stfing() stfing {

*// [...]*

out.WfiteStfing(fl.TokenLitef6l())

**if** fl.N6me != "" { out.WfiteStfing(fmt.Spfintf("<%s>", fl.N6me))

)

out.WfiteStfing("(")

*// [...]*

)

Now we can write a test for the parser that makes sure the N6me field is filled out when possible:

*// parser/parser\_test.go*

**func** TestFunctionLitef6lWithN6me(t \*testing.T) { input := `let myFunction = fn() { );`

l := lexef.New(input) p := New(l)

pfogf6m := p.P6fsePfogf6m() checkP6fsefEffofs(t, p)

**if** len(pfogf6m.St6tements) != 1 {

t.F6t6lf("pfogf6m.Body does not cont6in %d st6tements. got=%d\n", 1, len(pfogf6m.St6tements))

)

stmt, ok := pfogf6m.St6tements[🅓].(\*6st.LetSt6tement)

**if** !ok {

t.F6t6lf("pfogf6m.St6tements[🅓] is not 6st.LetSt6tement. got=%T", pfogf6m.St6tements[🅓])

)

function, ok := stmt.V6lue.(\*6st.FunctionLitef6l)

**if** !ok {

t.F6t6lf("stmt.V6lue is not 6st.FunctionLitef6l. got=%T", stmt.V6lue)

)

**if** function.N6me != "myFunction" {

t.F6t6lf("function litef6l n6me wfong. w6nt 'myFunction', got=%q\n", function.N6me)

)

)

Let’s make sure that we didn’t make a mistake and it fails as it should:

$ go test ./p6fsef

--- FAIL: TestFunctionLitef6lWithN6me (🅓.🅓🅓s)

p6fsef\_test.go:965: function litef6l n6me wfong. w6nt 'myFunction', got="" FAIL

FAIL monkey/p6fsef 🅓.🅓🅓5s

In order to fix it, we need to change the p6fseLetSt6tement method in the parser:

*// parser/parser.go*

**func** (p \*P6fsef) p6fseLetSt6tement() \*6st.LetSt6tement {

*// [...]*

stmt.V6lue = p.p6fseExpfession(LOWEST)

**if** fl, ok := stmt.V6lue.(\*6st.FunctionLitef6l); ok { fl.N6me = stmt.N6me.V6lue

)

*// [...]*

)

After both sides of the = in a let statement have been parsed, we check whether the right side, stmt.V6lue, is a \*6st.FunctionLitef6l. If it is, we save the name of binding, stmt.N6me, on to it. Yes, that’s it:

$ go test ./p6fsef

ok monkey/p6fsef 🅓.🅓🅓5s

Now we have everything in place to detect self-references! But instead of rushing into it and ending up with ugly checks for edge cases in our compiler, we go to the place that manages names and references for us: the symbol table.

What we’ll do in the symbol table is to add a new scope, called FunctionScope. We’ll only define one symbol with that scope per symbol table: the name of the function we’re currently compiling. When we then resolve a name and get back a symbol with FunctionScope we know that it’s the name of the current function, ergo a self-reference.

Let’s start by defining the new scope:

*// compiler/symbol\_table.go*

**const** (

*// [...]*

FunctionScope SymbolScope = "FUNCTION"

)

With that, we’re ready to add a test:

*// compiler/symbol\_table\_test.go*

**func** TestDefineAndResolveFunctionN6me(t \*testing.T) { glob6l := NewSymbolT6ble() glob6l.DefineFunctionN6me("6")

expected := Symbol{N6me: "6", Scope: FunctionScope, Index: 🅓) fesult, ok := glob6l.Resolve(expected.N6me)

**if** !ok {

t.F6t6lf("function n6me %s not fesolv6ble", expected.N6me)

)

**if** fesult != expected {

t.Effoff("expected %s to fesolve to %+v, got=%+v", expected.N6me, expected, fesult)

)

)

In this test we call DefineFunctionN6me, which we’ll have to write next, and then call Resolve with the function’s name in the hope that we’ll get back a Symbol with Scope set to FunctionScope.

In addition to that we also want to make sure that shadowing the current function’s name still works, like this:

**let** foob6f = fn() { **let** foob6f = 1; foob6f;

);

To ensure that we add a second test that’s similar to our first one, except that it contains an additional call to Define after the call to DefineFunctionN6me:

*// compiler/symbol\_table\_test.go*

**func** TestSh6dowingFunctionN6me(t \*testing.T) { glob6l := NewSymbolT6ble() glob6l.DefineFunctionN6me("6") glob6l.Define("6")

expected := Symbol{N6me: "6", Scope: Glob6lScope, Index: 🅓) fesult, ok := glob6l.Resolve(expected.N6me)

**if** !ok {

t.F6t6lf("function n6me %s not fesolv6ble", expected.N6me)

)

**if** fesult != expected {

t.Effoff("expected %s to fesolve to %+v, got=%+v",

expected.N6me, expected, fesult)

)

)

Alright, let’s run them:

$ go test -fun FunctionN6me ./compilef

# monkey/compilef [monkey/compilef.test] compilef/symbol\_t6ble\_test.go:3🅓6:8: glob6l.DefineFunctionN6me undefined \

(type \*SymbolT6ble h6s no field of method DefineFunctionN6me) compilef/symbol\_t6ble\_test.go:323:8: glob6l.DefineFunctionN6me undefined \

(type \*SymbolT6ble h6s no field of method DefineFunctionN6me) FAIL monkey/compilef [build f6iled]

Hard to misinterpret. Let’s implement DefineFunctionN6me, which is easy enough, since it’s really similar to the DefineBuiltin method we already have:

*// compiler/symbol\_table.go*

**func** (s \*SymbolT6ble) DefineFunctionN6me(n6me stfing) Symbol { symbol := Symbol{N6me: n6me, Index: 🅓, Scope: FunctionScope) s.stofe[n6me] = symbol

**return** symbol

)

We create a new Symbol with FunctionScope and add it to the s.stofe. The Index is an arbitrary choice, since it doesn’t matter. Feel free to set it to your favorite number. And then, run the tests again:

$ go test -fun FunctionN6me ./compilef ok monkey/compilef 🅓.🅓🅓5s

Now we can move on to the compiler where, as it turns out, we don’t have to do very much to make the failing tests pass.

When compiling a \*6st.FunctionLitef6l that has a .N6me, right after we entered the new com- pilation scope, we need to add the function’s name to the symbol table by making use of the new DefineFunctionN6me method:

*// compiler/compiler.go*

**func** (c \*Compilef) Compile(node 6st.Node) effof {

**switch** node := node.(**type**) {

*// [...]*

**case** \*6st.FunctionLitef6l: c.entefScope()

**if** node.N6me != "" { c.symbolT6ble.DefineFunctionN6me(node.N6me)

)

*// [...]*

*// [...]*

)

*// [...]*

)

Then we need to make sure that loading a FunctionScope symbol results in an OpCuffentClosufe

opcode being emitted. And that’s easy to do with the code we already have:

*// compiler/compiler.go*

**func** (c \*Compilef) lo6dSymbol(s Symbol) {

**switch** s.Scope {

*// [...]*

**case** FunctionScope: c.emit(code.OpCuffentClosufe)

)

)

In total, we’ve now added five new lines to the compiler – enough to fix the tests:

$ go test ./compilef

ok monkey/compilef 🅓.🅓🅓6s

Great! You know what that means, right? We can now finally take care of the failing VM tests we started with:

$ go test ./vm -fun TestRecufsiveFunctions

--- FAIL: TestRecufsiveFunctions (🅓.🅓🅓s)

vm\_test.go:591: vm effof: c6lling non-closufe 6nd non-builtin FAIL

FAIL monkey/vm 🅓.🅓🅓5s

And the best part? We already did most of the work — in the parser, in the symbol table and in the compiler. All that’s left to do is to implement OpCuffentClosufe in the VM:

*// vm/vm.go*

**func** (vm \*VM) Run() effof {

*// [...]*

**switch** op {

*// [...]*

**case** code.OpCuffentClosufe:

cuffentClosufe := vm.cuffentFf6me().cl eff := vm.push(cuffentClosufe)

**if** eff != nil {

**return** eff

)

*// [...]*

)

*// [...]*

)

We take the closure of the vm.cuffentFf6me() and push it on to the stack. That’s it:

$ go test ./vm

ok monkey/vm 🅓.🅓33s

We’re done! We’ve added the last missing piece and with great confidence we can now say: we’ve successfully implemented closures in a bytecode compiler and a bytecode VM! We’ve added the crown jewel to our shiniest Monkey implementation. Time to celebrate.

CHAPTER 10

TAKING TIME

We’re at the end of our journey. We did it. We successfully built a bytecode compiler and a virtual machine.

We’ve implemented binary and prefix operators, conditionals with jump instructions, global and local bindings, strings, arrays, hashes, first-class functions, higher-order functions, built-in functions and now even closures, the most prestigious of all the features.

It’s time we give ourselves a little pat on the back. With great contentment, with the fulfillment that comes after having put in the work, we can now watch as our compiler compiles and our VM executes the following piece of Monkey code.

This is *the* go-to example to showcase a programming language. Yes, it might be a cliché, but nevertheless, it’s a milestone and always brings a smile to my face. I’m talking about the recursive function for computing a Fibonacci number:

*// vm/vm\_test.go*

**func** TestRecufsiveFibon6cci(t \*testing.T) { tests := []vmTestC6se{

{

input: `

let fibon6cci = fn(x) { if (x == 🅓) {

fetufn 🅓;

) else {

if (x == 1) {

fetufn 1;

) else {

fibon6cci(x - 1) + fibon6cci(x - 2);

)

)

);

fibon6cci(15);

`,

expected: 61🅓,

),

)

funVmTests(t, tests)

)

Ah, recursion! Beautiful, isn’t it? Here we go, cross your fingers, knock on wood:

$ go test ./vm

ok monkey/vm 🅓.🅓34s

Beautiful! Now, using a recursive Fibonacci function to showcase a language’s ability is not exactly what created this aura of cliché – using it as a benchmark for a language’s performance is.

Knowing how fast a language can execute such a function tells us nothing about how it performs in a production setting, with *real* code and a *real* workload. But we also know that Monkey was never built for that anyway and that benchmarks and numbers are fun. And you also might remember that, in the first chapter, I promised that this new implementation of Monkey will be three times as fast as the old one. It’s time for me to deliver on that promise.

Let’s create a little utility that allows us to compare our evaluator from the first book against our new bytecode interpreter and see how fast they can calculate a Fibonacci number.

In a new folder, benchm6fk, we create a new m6in.go file:

*// benchmark/main.go*

**package** m6in

**import** (

"fl6g"

"fmt"

"time"

"monkey/compilef" "monkey/ev6lu6tof" "monkey/lexef" "monkey/object" "monkey/p6fsef" "monkey/vm"

)

**var** engine = fl6g.Stfing("engine", "vm", "use 'vm' of 'ev6l'")

**var** input = `

let fibon6cci = fn(x) { if (x == 🅓) {

🅓

) else {

if (x == 1) { fetufn 1;

) else {

fibon6cci(x - 1) + fibon6cci(x - 2);

)

)

);

fibon6cci(35);

`

**func** m6in() {

fl6g.P6fse()

**var** duf6tion time.Duf6tion

**var** fesult object.Object

l := lexef.New(input)

p := p6fsef.New(l)

pfogf6m := p.P6fsePfogf6m()

**if** \*engine == "vm" {

comp := compilef.New()

eff := comp.Compile(pfogf6m)

**if** eff != nil {

fmt.Pfintf("compilef effof: %s", eff)

**return**

)

m6chine := vm.New(comp.Bytecode()) st6ft := time.Now()

eff = m6chine.Run()

**if** eff != nil {

fmt.Pfintf("vm effof: %s", eff)

**return**

)

duf6tion = time.Since(st6ft)

fesult = m6chine.L6stPoppedSt6ckElem()

) **else** {

env := object.NewEnvifonment() st6ft := time.Now()

fesult = ev6lu6tof.Ev6l(pfogf6m, env) duf6tion = time.Since(st6ft)

)

fmt.Pfintf(

"engine=%s, fesult=%s, duf6tion=%s\n",

\*engine, fesult.Inspect(), duf6tion)

)

There’s nothing here we haven’t seen before. The input is the same fibonacci function we already know we can compile and execute, except that this time, the input is 35, which gives our interpreters something to chew on.

In the m6in function we parse the command-line flag engine and, depending on its value, either execute the fibon6cci snippet in the evaluator from the first book or compile it for and execute it in our new, shiny VM. Either way, we measure the time it takes to execute it and then print a summary of the benchmark.

Running it will tell us how much performance we gained by switching from a tree-walking interpreter to a compiler and a virtual machine – without focusing on performance at all and a lot of optimization opportunities left unused.

We build it into its own executable:

$ go build -o fibon6cci ./benchm6fk

And now, welcome, first on the stage, the evaluator:

$ ./fibon6cci -engine=ev6l

engine=ev6l, fesult=9227465, duf6tion=27.2🅓4277379s

27 seconds. Time for our final pat on the back:

$ ./fibon6cci -engine=vm

engine=vm, fesult=9227465, duf6tion=8.876222455s

8 seconds. 3.3 times faster.

# RESOURCES

BOOKS

* Abelson, Harold and Sussman, Gerald Jay with Sussman, Julie. 1996. **Structure and Interpretation of Computer Programs, Second Edition**. MIT Press.
* Appel, Andrew W.. 2004. **Modern Compiler Implementation in C**. Cambridge University Press.
* Appel, Andrew W.. 2004. **Modern Compiler Implementation in ML**. Cambridge University Press.
* Cooper, Keith D. and Torczon Linda. 2011. **Engineering a Compiler, Second Edi- tion**. Morgan Kaufmann.
* Grune, Dick and Jacobs, Ceriel. 1990. **Parsing Techniques. A Practical Guide.**. Ellis Horwood Limited.
* Grune, Dick and van Reeuwijk, Kees and Bal Henri E. and Jacobs, Ceriel J.H. Jacobs and Langendoen, Koen. 2012. **Modern Compiler Design, Second Edition**. Springer
* Nisan, Noam and Schocken, Shimon. 2008. **The Elements Of Computing Systems**. MIT Press.
* Parr, Terrence. 2010. **Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages**. Pragmatic Programmers.
* Queinnec, Christian. 2003. **Lisp in Small Pieces**. Cambridge University Press.

PAPERS

* Ayock, John. 2003. **A Brief History of Just-In-Time**. In **ACM Computing Sur- veys, Vol. 35, No. 2, June 2003**
* Dybvig, R. Kent. 2006. **The Development of Chez Scheme**. In **ACM ICFP ’06**
* Dybvig, R. Kent. 1987. **Three Implementation Models for Scheme**. Dissertation, University of North Carolina at Chapel Hill
* Ertl, M. Anton and Gregg, David. 2003. **The Structure and Performance of Eﬀicient Interpreters**. In **Journal Of Instruction-Level Parallelism 5 (2003)**
* Ghuloum, Abdulaziz. 2006. **An Incremental Approach To Compiler Construction**. In **Proceedings of the 2006 Scheme and Functional Programming Workshop**.
* Ierusalimschy, Robert and de Figueiredo, Luiz Henrique and Celes Waldemar. **The Im- plementation of Lua 5.0**. <https://www.lua.org/doc/jucs05.pdf>
* Pratt, Vaughan R. 1973. **Top Down Operator Precedence**. Massachusetts Institute of Technology.
* Romer, Theodore H. and Lee, Dennis and Voelker, Geoffrey M. and Wolman, Alec and Wong, Wayne A. and Baer, Jean-Loup and Bershad, Brian N. and Levy, Henry M.. 1996. **The Structure and Performance of Interpreters**. In **ASPLOS VII Proceedings of the seventh international conference on Architectural support for program- ming languages and operating systems**.
* Fang, Ruijie and Liu. **A Performance Survey on Stack-based and Register-based**

###### Virtual Machines.

WEB

* Aaron Patterson - Reducing Memory Usage In Ruby: [https://tenderlovemaking.com/](https://tenderlovemaking.com/2018/01/23/reducing-memory-usage-in-ruby.html) [2018/01/23/reducing-memory-usage-in-ruby.html](https://tenderlovemaking.com/2018/01/23/reducing-memory-usage-in-ruby.html)
* Allison Kaptur - A Python Interpreter Written in Python: [http://aosabook.org/en/500L/](http://aosabook.org/en/500L/a-python-interpreter-written-in-python.html) [a-python-interpreter-written-in-python.html](http://aosabook.org/en/500L/a-python-interpreter-written-in-python.html)
* Andy Wingo - a lambda is not (necessarily) a closure: [https://wingolog.org/archives/](https://wingolog.org/archives/2016/02/08/a-lambda-is-not-necessarily-a-closure) [2016/02/08/a-lambda-is-not-necessarily-a-closure](https://wingolog.org/archives/2016/02/08/a-lambda-is-not-necessarily-a-closure)
* Andy Wingo - a register vm for guile: [https://wingolog.org/archives/2013/11/26/](https://wingolog.org/archives/2013/11/26/a-register-vm-for-guile) [a-register-vm-for-guile](https://wingolog.org/archives/2013/11/26/a-register-vm-for-guile)
* Andy Wingo - the half strap: self-hosting and guile: [https://wingolog.org/archives/2016/](https://wingolog.org/archives/2016/01/11/the-half-strap-self-hosting-and-guile) [01/11/the-half-strap-self-hosting-and-guile](https://wingolog.org/archives/2016/01/11/the-half-strap-self-hosting-and-guile)
* Eli Bendersky - Adventures in JIT compilation: Part 2 - an x64 JIT: [https:](https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-2-an-x64-jit/)

[//eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-2-an-x64-jit/](https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-2-an-x64-jit/)

* GNU Guile Documentation - About Closures - [https://www.gnu.org/software/guile/](https://www.gnu.org/software/guile/manual/guile.html#About-Closure) [manual/guile.html#About-Closure](https://www.gnu.org/software/guile/manual/guile.html#About-Closure)
* Jack W. Crenshaw - Let’s Build a Compiler! - [http://compilers.iecc.com/crenshaw/](http://compilers.iecc.com/crenshaw/tutorfinal.pdf) [tutorfinal.pdf](http://compilers.iecc.com/crenshaw/tutorfinal.pdf)
* Kein-Hong Man - A No-Frills Introduction to Lua 5.1 VM Instructions: [http://luaforge.](http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf) [net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf](http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf)
* Lua 5.3 Bytecode Reference: [http://the-ravi-programming-language.readthedocs.io/en/](http://the-ravi-programming-language.readthedocs.io/en/latest/lua_bytecode_reference.html) [latest/lua\_bytecode\_reference.html](http://the-ravi-programming-language.readthedocs.io/en/latest/lua_bytecode_reference.html)
* LuaJIT 2.0 Bytecode Instructions: <http://wiki.luajit.org/Bytecode-2.0>
* Mathew Zaleski - Dispatch Techniques: [http://www.cs.toronto.edu/~matz/dissertation/](http://www.cs.toronto.edu/~matz/dissertation/matzDissertation-latex2html/node6.html) [matzDissertation-latex2html/node6.html](http://www.cs.toronto.edu/~matz/dissertation/matzDissertation-latex2html/node6.html)
* Matt Might - Closure Conversion - “Flat Closures”: [http://matt.might.net/articles/](http://matt.might.net/articles/closure-conversion/) [closure-conversion/](http://matt.might.net/articles/closure-conversion/)
* Matt Might - Compiling Scheme to C with closure conversion: [http://matt.might.net/](http://matt.might.net/articles/compiling-scheme-to-c/) [articles/compiling-scheme-to-c/](http://matt.might.net/articles/compiling-scheme-to-c/)
* Mozilla SpiderMonkey Internals: [https://developer.mozilla.org/en-US/docs/Mozilla/](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals) [Projects/SpiderMonkey/Internals](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals)
* Stack Overflow - Implementation of closures in Lua?: [https://stackoverflow.com/](https://stackoverflow.com/questions/7781432/implementation-of-closures-in-lua) [questions/7781432/implementation-of-closures-in-lua](https://stackoverflow.com/questions/7781432/implementation-of-closures-in-lua)
* The Cliffs of Inanity - Emacs JIT Calling Convention: <http://tromey.com/blog/?p=999>
* Vyacheslav Egorov - Explaining JavaScript VMs in JavaScript - Inline Caches: [https:](https://mrale.ph/blog/2012/06/03/explaining-js-vms-in-js-inline-caches.html)

[//mrale.ph/blog/2012/06/03/explaining-js-vms-in-js-inline-caches.html](https://mrale.ph/blog/2012/06/03/explaining-js-vms-in-js-inline-caches.html)

* Peter Michaux - Scheme from Scratch - Introduction: [http://peter.michaux.ca/articles/](http://peter.michaux.ca/articles/scheme-from-scratch-introduction) [scheme-from-scratch-introduction](http://peter.michaux.ca/articles/scheme-from-scratch-introduction)

SOURCE CODE

* 8cc - A Small C Compiler - <https://github.com/rui314/8cc>
* GNU Guile 2.2 - <https://www.gnu.org/software/guile/download/>
* MoarVM - A modern VM built for Rakudo Perl 6 - [https://github.com/MoarVM/](https://github.com/MoarVM/MoarVM) [MoarVM](https://github.com/MoarVM/MoarVM)
* The Lua Programming Language (1.1, 3.1, 5.3.2) - <https://www.lua.org/versions.html>
* The Ruby Programming Language - <https://github.com/ruby/ruby>
* The Wren Programming Language - <https://github.com/munificent/wren>
* c4 - C in four functions - <https://github.com/rswier/c4>
* tcc - Tiny C Compiler - <https://github.com/LuaDist/tcc>

# FEEDBACK

If you spot a typo, find something wrong with the code, have a suggestion to make or just a question, feel free to send me an email:

###### [me@thorstenball.com](mailto:me@thorstenball.com)

# CHANGELOG

7 MAy 2019 - 1.2

* Code
  + Added go.mod files to the code folder so that users with Go >= 1.13 can easily run it without having to set $GOPATH
* Introduction and following chapters
  + Changed REPL code from fmt.Pfintf to correctly use fmt.Fpfintf to make use of

out parameter

* Chapter 1 - Real Machines
  + Fixed a typo where it said “We’re going the” instead of “We’re going to”
* Chapter 2 - Starting With Bytes
  + Removed a comma that disrupted flow
* Chapter 3 - Booleans
  + Fixed a typo where it said “There are also has” instead of “There are also”
* Chapter 3 - Comparison Operators
  + Changed the conditional in executeComp6fison that checks that both operands are integers from a || to the correct &&
* Chapter 4
  + Fixed a typo where it said “We need put” instead of “We need to put”
* Chapter 7
  + Fixed the wrong function names in the code snippet: minsOne and minsTwo to minusOne

and minusTwo

* Chapter 9 - Recursive Closures
  + Removed duplicate word “fail”

20 MARCH 2019 - 1.1

Besides fixes for various typos and spelling errors:

* Chapter 3 - Infix Expressions:
  + Remove duplicated test case
* Chapter 6 - Array:
  + Fix broken link to Wikipedia entry for “Composite Data Type”
* Chapter 9 - Recursive Closures:
  + Add a completely new section that addresses a bug in the previous implementation that prevented recursive closures defined in other closures from working correctly
* Chapter 10 - Taking Time:
  + Chapter shortened after reworking Chapter 9 to include the proper for recursive closures

31 JULy 2018 - 1.0

Initial Release