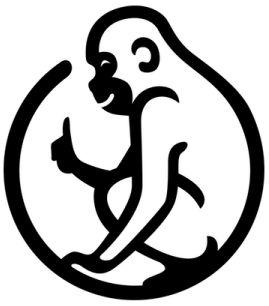
THORSTEN BALL

WRITING A COMPILER IN GO



WRITING A COMPILER IN GO

THORSTEN BALL

CHAPTER 1

COMPILERS & VIRTUAL MACHINES

This first chapter is not included in the sample. Instead, the sample contains the second chapter, which is the first one in which we write code.

CHAPTER 2

HELLO BYTECODE!

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

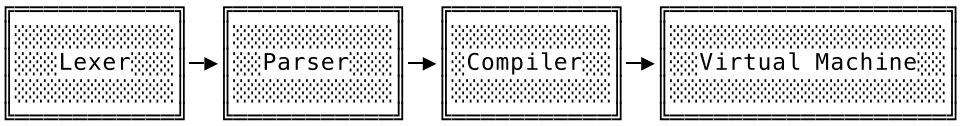
```
1 + 2
```

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.

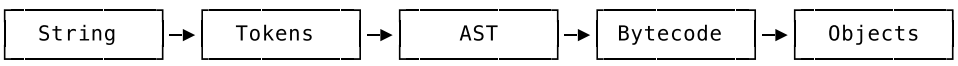
At the end of this chapter we want to be able to:

- 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.

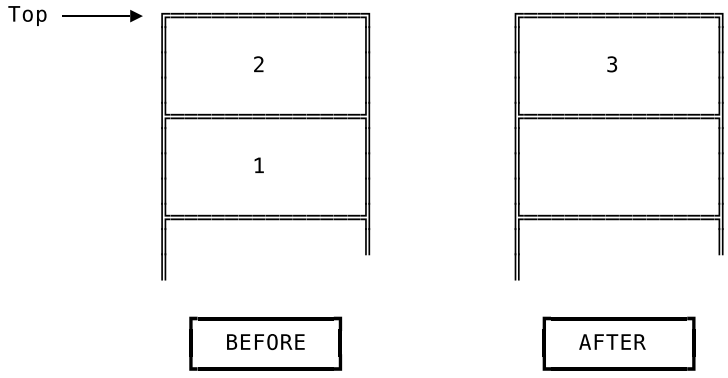
FIRST INSTRUCTIONS

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.

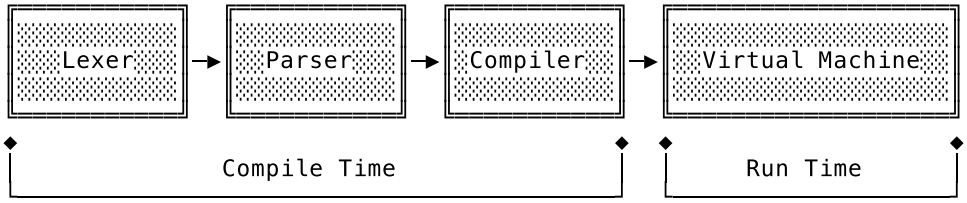
The first one is `Instruction` – singular. Why didn't we define it here as `[]byte`? Because it's far more handy to pass around and work with a `[]byte` and treat it *implicitly* as an instruction than to encode this definition in Go's type system. You'll see soon enough how often we're going to use `[]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 literal (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 instructions 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 previously 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 `0` 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.Errorf("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 `Opcode`s 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 `0xFF 0xFE` and `65535` would be encoded as `0xFF 0xFF` – 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[0] = 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 0.007s
```

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, 0),
                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.Fatalf("compiler error: %s", err)
        }

        bytecode := compiler.Bytecode()

        err = testInstructions(tt.expectedInstructions, bytecode.Instructions)
        if err != nil {
            t.Fatalf("testInstructions failed: %s", err)
        }

        err = testConstants(t, tt.expectedConstants, bytecode.Constants)
        if err != nil {
            t.Fatalf("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 `compilerTestCase`s 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 (0.00s)
 compiler_test.go:31: testInstructions failed: wrong instructions length.
  want="\x00\x00\x00\x00\x00\x01"
  got =""
FAIL
FAIL    monkey/compiler 0.008s
```

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="\x00\x00\x00\x00\x00\x01"
```

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 := `0000 OpConstant 1
0003 OpConstant 2
0006 OpConstant 65535
`

    concatted := Instructions{}
    for _, ins := range instructions {
        concatted = append(concatted, ins...)
    }
```

```
    if concatted.String() != expected {
        t.Errorf("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 `\x00\x00\x00\x00\x00\x01`, 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 (0.00s)
 code_test.go:49: instructions wrongly formatted.
  want="0000 OpConstant 1\n0003 OpConstant 2\n0006 OpConstant 65535\n"
  got=""
FAIL
FAIL    monkey/code 0.008s
```

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.Fatalf("definition not found: %q\n", err)
        }

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

```
        if n != tt.bytesRead {
            t.Fatalf("n wrong. want=%d, got=%d", tt.bytesRead, n)
        }

        for i, want := range tt.operands {
            if operandsRead[i] != want {
                t.Errorf("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 := 0

    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 (0.00s)
 code_test.go:49: instructions wrongly formatted.
  want="0000 OpConstant 1\n0003 OpConstant 2\n0006 OpConstant 65535\n"
  got=""
FAIL
FAIL    monkey/code 0.008s
```

Now that we have `ReadOperands`, we can get rid of the blank string and properly print instructions:

```
// code/code.go

import (
    "bytes"
    // [...]
)

func (ins Instructions) String() string {
    var out bytes.Buffer

    i := 0
    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, "%04d %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[0])
    }
```

```
        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 0.008s
```

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 (0.00s)
 compiler_test.go:31: testInstructions failed: wrong instructions length.
  want="0000 OpConstant 0\n0003 OpConstant 1\n"
  got =""
FAIL
FAIL    monkey/compiler 0.008s
```

Isn't that beautiful? Alright, granted, beautiful may be a tad too much, but it sure isn't the eyesore that was `want="\x00\x00\x00\x00\x00\x01"`.

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".

END OF SAMPLE

You've reached the end of the sample. I hope you enjoyed it.

You can buy the full version of the book (including the complete codebase) or a bundle that includes *Writing An Interpreter In Go* here:

https://compilerbook.com