1 Tutorial

This chapter is a tour of the basic components of Go. We hope to provide enough information and examples to get you off the ground and doing useful things as quickly as possible. The examples here, and indeed in the whole book, are aimed at tasks that you might have to do in the real world. In this chapter we'll try to give you a taste of the diversity of programs that one might write in Go, ranging from simple file processing and a bit of graphics to concurrent Internet clients and servers. We certainly won't explain everything in the first chapter, but studying such programs in a new language can be an effective way to get started. [1]

When you're learning a new language, there's a natural tendency to write code as you would have written it in a language you already know. Be aware of this bias as you learn Go and try to avoid it. We've tried to illustrate and explain how to write goodYou can edit Typst documents online.[1] Checkout Typst's website.[1] And the online app.[1] Go, so use the code here as a guide when you're writing your own.

1.1 Hello World

We'll start with the now-traditional "hello, world" example, which appears at the beginning of The C Programming Language, published in 1978. C is one of the most direct influences on Go, and "hello, world" illustrates a number of central ideas.

gopl.io/ch1/helloworld

```
package main

import "fmt"

func main() {
   fmt.Println("Hello, BF")
}
```

Go is a compiled language. The Go toolchain converts a source program and the things it depends on into instructions in the native machine language of a computer. These tools are accessed through a single command called go that has a number of subcommands. The sim- plest of these subcommands is run, which compiles the source code from one or more source files whose names end in .go, links it with libraries, then runs the resulting executable file. (We will use$ as the command prompt throughout the book.)

1: https://typst.app/app

`\$ go run helloworld.go` Not surprisingly, this prints Hello, BF Go natively handles Unicode, so it can process text in all the world's languages.

If the program is more than a one-shot experiment, it's likely that you would want to compile

it once and save the compiled result for later use. That is done with go build:$ go build helloworld.go

This creates an executable binary file called helloworld that can be run any time without fur- ther processing:

```
\$ ./helloworld Hello, BF
```

We have labeled each significant example as a reminder that you can obtain the code from the book's source code repository at gopl.io:

gopl.io/ch1/helloworld

If you run go get gopl.io/ch1/helloworld, it will fetch the source code and place it in the corresponding directory. There's more about this topic in Section 2.6 and Section 10.7.

Let's now talk about the program itself. Go code is organized into packages, which are similar to libraries or modules in other languages. A package consists of one or more .go source files in a single directory that define what the package does. Each source file begins with a package declaration, here package main, that states which package the file belongs to, followed by a list of other packages that it imports, and then the declarations of the program that are stored in that file.

The Go standard library has over 100 packages for common tasks like input and output, sorting, and text manipulation. For instance, the fmt package contains functions for printing formatted output and scanning input. Println is one of the basic output functions in fmt; it prints one or more values, separated by spaces, with a newline character at the end so that the values appear as a single line of output.

Package main is special. It defines a standalone executable program, not a library. Within package main the function main is also special—it's where execution of the program begins. Whatever main does is what the program does. Of course, main will normally call upon func- tions in other packages to do much of the work, such as the function fmt.Println.

We must tell the compiler what packages are needed by this source file; that's the role of the import declaration that follows the package declaration. The "hello, world" program uses only one function from one other package, but most programs will import more packages.

You must import exactly the packages you need. A program will not compile if there are missing imports or if there are unnecessary ones. This strict requirement prevents references to unused packages from accumulating as programs evolve.

The import declarations must follow the package declaration. After that, a program consists of the declarations of functions, variables, constants, and types (introduced by the keywords func, var, const, and type); for the most part, the order of declarations does not matter. This program is about as short as possible since it declares only one function, which in turn calls only one other function. To save space, we

will sometimes not show the package and import declarations when presenting examples, but they are in the source file and must be there to compile the code.

A function declaration consists of the keyword func, the name of the function, a parameter list (empty for main), a result list (also empty here), and the body of the function—the state- ments that define what it does—enclosed in braces. We'll take a closer look at functions in Chapter 5.

Go does not require semicolons at the ends of statements or declarations, except where two or more appear on the same line. In effect, newlines following certain tokens are converted into semicolons, so where newlines are placed matters to proper parsing of Go code. For instance, the opening brace { of the function must be on the same line as the end of the func declara- tion, not on a line by itself, and in the expression x + y, a newline is permitted after but not before the + operator.

Go takes a strong stance on code formatting. The gofmt tool rewrites code into the standard format, and the go tool's fmt subcommand applies gofmt to all the files in the specified pack- age, or the ones in the current directory by default. All Go source files in the book have been run through gofmt, and you should get into the habit of doing the same for your own code. Declaring a standard format by fiat eliminates a lot of pointless debate about trivia and, more importantly, enables a variety of automated source code transformations that would be infeasible if arbitrary formatting were allowed.

Many text editors can be configured to run gofmt each time you save a file, so that your source code is always properly formatted. A related tool, goimports, additionally manages the inser- tion and removal of import declarations as needed. It is not part of the standard distribution but you can obtain it with this command:

```
go get golang.org/x/tools/cmd/goimports
```

For most users, the usual way to download and build packages, run their tests, show their doc- umentation, and so on, is with the go tool, which we'll look at in Section 10.7.

1.2 Command-Line Arguments

Most programs process some input to produce some output; that's pretty much the definition of computing. But how does a program get input data on which to operate? Some programs generate their own data, but more often, input comes from an external source: a file, a network connection, the output of another program, a user at a keyboard, command-line arguments, or the like. The next few examples will discuss some of these alternatives, starting with com- mand-line arguments.

The os package provides functions and other values for dealing with the operating system in a platform-independent fashion. Command-line arguments are available to a program in a variable named Args that is part of the os package; thus its name anywhere outside the os package is os.Args.

The variable os.Args is a slice of strings. Slices are a fundamental notion in Go, and we'll talk a lot more about them soon. For now, think of a slice as a dynamically sized sequence s of array elements where individual elements can be accessed as s[i] and a contiguous subse- quence as s[m:n]. The number of elements is given by len(s). As in most other program- ming languages, all indexing in Go uses half-open

intervals that include the first index but exclude the last, because it simplifies logic. For example, the slice s[m:n], where 0 ≤ m ≤ n ≤ len(s), contains n-m elements.

The first element of os.Args, os.Args[0], is the name of the command itself; the other ele- ments are the arguments that were presented to the program when it started execution. A slice expression of the form s[m:n] yields a slice that refers to elements m through n-1, so the elements we need for our next example are those in the slice os.Args[1:len(os.Args)]. If m or n is omitted, it defaults to 0 or len(s) respectively, so we can abbreviate the desired slice as os.Args[1:].

Here's an implementation of the Unix echo command, which prints its command-line argu- ments on a single line. It imports two packages, which are given as a parenthesized list rather than as individual import declarations. Either form is legal, but conventionally the list form is used. The order of imports doesn't matter; the gofmt tool sorts the package names into alphabetical order. (When there are several versions of an example, we will often number them so you can be sure of which one we're talking about.)

gopl.io/ch1/echo1

```
// Echo1 prints its command-line arguments.
package main
import (
    "fmt"
)
```

Under the covers, bufio.Scanner, ioutil.ReadFile, and ioutil.WriteFile use the Read and Write methods of `*os.File`, but it's rare that most programmers need to access those lower-levelroutinesdirectly. Thehigher-levelfunctionslikethosefrombufioandio/ioutil are easier to use.

Exercise 1.4: Modify dup2 to print the names of all files in which each duplicated line occurs. 1.4. Animated GIFs

The next program demonstrates basic usage of Go's standard image packages, which we'll use to create a sequence of bit-mapped images and then encode the sequence as a GIF animation. The images, called Lissajous figures, were a staple visual effect in sci-fi films of the 1960s. They are the parametric curves produced by harmonic oscillation in two dimensions, such as two sine waves fed into the x and y inputs of an oscilloscope. Figure 1.1 shows some examples.

Figure 1.1. Four Lissajous figures.

There are several new constructs in this code, including const declarations, struct types, and composite literals. Unlike most of our examples, this one also involves floating-point com- putations. We'll discuss these topics only briefly here, pushing most details off to later chap- ters, since the primary goal right now is to give you an idea of what Go looks like and the kinds of things that can be done easily with the language and its libraries.

gopl.io/ch1/lissajous

```
    // Lissajous generates GIF animations of random Lissajous figures.
    package main
    import (
        "image"
```

) In case you missed it, there's a recording on EDGARP.

References

[1] R. Astley, and L. Morris, "At-scale impact of the Net Wok: a culinarically holistic investigation of distributed dumplings," Armenian J. Proc., vol. 61, pp. 192–219, 2020.