CPAM

Cross-Platform Application Management

Author:
T.J. Yang

Thanks to:
TWW Inc.

With the help and contributions from:

(in alphabetical order)

Anthony Magro, *Uriel*.

Extension of TWW Inc. toolsets to open source OS (*0.5*)

Software Pacakge creation and management across OS platform made easy

Contents

List of Figures

List of Code Examples

List of Exercises

Preface

> "Is Go an object-oriented language? Yes and no."
>
> *Frequently asked questions*
> GO AUTHORS

Audience

This is an introduction to the Go language from Google. Its aim is to provide a guide to this new and innovative language.

The intended audience of this book is people who are familiar with programming and know some programming languages, be it C[7], C++[22], Perl[8], Java[17], Erlang[6], Scala[1] or Haskell[2]. This is *not* a book which teaches you how to program, this is a book that just teaches you how to use Go.

As with learning new things, probably the best way to do this is to discover it for yourself by creating your own programs. Each chapter therefore includes a number of exercises (and answers) to acquaint you with the language. An exercise is numbered as **Q**n, where n is a number. After the exercise number another number in parentheses displays the difficulty of this particular assignment. This difficulty ranges from 0 to 9, where 0 is easy and 9 is difficult. Then a short name is given, for easier reference. For example:

> **Q1**. (1) A map function ...

introduces a question numbered **Q1** of a level 1 difficulty, concerning a `map()`-function. The answers are included after the exercises on a new page. The numbering and setup of the answers is identical to the exercises, except that an answer starts with **A**n, where the number n corresponds with the number of the exercise. Some exercises don't have an answer, they are marked with an asterisks.

Book layout

Chapter 1: Introduction
: A short introduction and history of Go. It tells how to get the source code of Go itself. It assumes a Unix-like environment, although Go should be fully usable on Windows.

Chapter 2: Basics
: Tells about the basic types, variables and control structures available in the language.

Chapter **??***:* **??**
: In the third chapter we look at functions, the basic building blocks of Go programs.

Chapter **??***:* **??**
: In chapter **??** we see that functions and data can be grouped together in packages. You will also see how to document and test your packages.

Chapter **??***:* **??**
: After that we look at creating your own types in chapter **??**. It also looks at allocation in Go.

Chapter **??***:* **??**
: Go does not support Object Orientation in the traditional sense. In Go the central concept is interfaces.

Chapter **??***:* **??**
: With the `go` keyword functions can be started in separate routines (called goroutines). Communication with those goroutines is done via channels.

Chapter **??***:* **??**
: In the last chapter we show how to interface with the rest of the world from within a Go program. How create files and read and wrote from and to them. We also briefly look into networking.

I hope you will enjoy this book and the language Go.

Settings used in this book

- Go itself is installed in `~/go` ;
- Go source code we want to compile ourself is placed in `~/g/src` and `$GOPATH` is set to `GOPATH=~/g` .

Miek Gieben, 2011, 2012 – `miek@miek.nl`

1 Introduction

> "I am interested in this and hope to do something."
>
> *On adding complex numbers to Go*
> KEN THOMPSON

What is Go? From the website [14]:

> *The Go programming language is an open source project to make programmers more productive. Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multi core and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.*

Go is a young language, where features are still being added or even *removed*. It may be possible that some text is outdated when you read it. Some exercise answers may become incorrect as Go continues to evolve. We will do our best to keep this document up to date with respect to the latest Go release. An effort has been made to create "future proof" code examples.

The following convention is used throughout this book:

- Code is displayed in `DejaVu Mono`;
- Keywords are displayed in **`DejaVu Mono Bold`**;
- Comments are displayed in *`DejaVu Mono Italic`*;
- Extra remarks in the code ← *Are displayed like this*;
- Longer remarks get a number – ❶ – with the explanation following;
- Line numbers are printed on the right side;
- Shell examples use a **%** as prompt;
- User entered text in shell examples **`is in bold`**, system responses are in a `typewriter font`;
- An emphasized paragraph is indented and has a vertical bar on the left.

Official documentation

When searching on the internet use the term "golang" instead of plain "go".

There already is a substantial amount of documentation written about Go. The Go Tutorial [13], and the Effective Go document [10]. The website `http://golang.org/doc/` is a very good starting point for reading up on Go[a]. Reading these documents is certainly not required, but is recommended.

Go comes with its own documentation in the form of a program called `go doc`. You can use it yourself to look in the on-line documentation. For instance, suppose we want to know more about the package *hash*. We would then give the command `go doc hash`. How to create your own package documentation is explained in chapter **??**.

Origins

Go has it origins in Inferno [3] (which in turn was based upon Plan 9 [4]). Inferno included a language called Limbo [5]. Quoting from the Limbo paper:

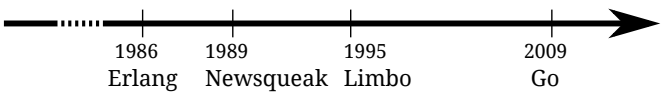
> *Limbo is a programming language intended for applications running distributed systems on small computers. It supports modular programming, strong type checking at compile- and run-time,* inter process communication over typed channels*, automatic* garbage collection*, and simple abstract data types. It is designed for safe execution even on small machines without hardware memory protection.*

A feature Go inherited from Limbo is channels (see chapter **??**). Again from the Limbo documentation.

> *[A channel] is a communication mechanism capable of sending and receiving objects of the specified type to another agent in the system. Channels may be used to communicate between local processes; using library procedures, they may be connected to named destinations. In either case send and receive operations may be directed to them.*

The channels in Go are easier to use than those in Limbo. If we dig even deeper in the history of Go we also find references to "Newsqueak" [20], which pioneered the use of channel communication in a C–like language. Channel communication isn't unique to these languages, a big non–C–like language which also uses them is Erlang [6].

Figure 1.1. Chronology of Go



The whole of idea of using channels to communicate with other processes is called Communicating Sequential Processes (CSP) and was conceived by C. A. R. Hoare [19], who incidentally is the same man that invented QuickSort [18].

[a] `http://golang.org/doc/` itself is served by a Go program called `go doc`.

> Go is the first C–like language that is widely available, runs on many different platforms and makes concurrency easy (or easier).

Getting Go

In this section we tell how to install Go locally on your machine, but you can also compile Go code online at `http://play.golang.org/`. To quickly play with code this is by far the easiest route.

Ubuntu and Debian both have a Go package in their repositories, look for the package "golang". But there are still some minor issues being worked out. For now we will stick to the installation from source.

So we will retrieve the code from the mercurial archive and compile Go yourself. For other Unix like systems the procedure is the same.

- First install Mercurial (to get the `hg` command). In Ubuntu/Debian/Fedora you must install the `mercurial` package;
- For building Go you need the packages: `bison`, `gcc`, `libc6-dev`, `ed`, `gawk` and `make`;
- Set the environment variable `GOROOT` to the root of your Go install:
  ```
  % export GOROOT=/go
  ```
- Then retrieve the Go source code:
  ```
  % hg clone -r release https://go.googlecode.com/hg/ $GOROOT
  ```
- Set your PATH to so that the Shell can find the Go binaries:
  ```
  % export PATH=$GOROOT/bin:$PATH
  ```
- Compile Go
  ```
  % cd $GOROOT/src
  % ./all.bash
  ```

If all goes well, you should see the following at the end:

```
--- cd ../test
0 known bugs; 0 unexpected bugs

ALL TESTS PASSED

---
Installed Go for linux/amd64 in /home/go
Installed commands in /home/go/bin
```

You now have Go installed on your system and you can start playing. Note that currently (March 2012) this install version r60 of, this version is old. If you want Go1, or something close to it, you will have to upgrade to the latest weekly, see the section "Keeping up to date".

Keeping up to date

New releases are announced on the Go Nuts mailing list [16]. To update an existing tree to the latest release, you can run:

```
% cd $GOROOT
% hg pull
% hg update release
% cd src
% ./all.bash
```

To see what you are running right now:

```
% cd $GOROOT
% hg identify
79997f0e5823 release/release.2010-10-20
```

That would be release `2010-10-20`. The release as describe is a "stable" releases, as opposed to the "weekly" releases that are more volatile. If you want to track the weekly releases instead of the stable ones you can use:

```
% hg update weekly
```

In stead of

```
% hg update release
```

Exercises

Q1. (1) Documentation

1. Go's documentation can be read with the `go doc` program, which is included the Go distribution.

   `go doc hash` gives information about the *hash* package. Reading the documentation on *compress* gives the following result:

   ```
   % go doc compress
   SUBDIRECTORIES

           bzip2
           flate
           gzip
           lzw
           testdata
           zlib
   ```

   With which `go doc` command can you read the documentation of *gzip* contained in *compress*?

Answers

A1. (1) Documentation

1. The package *gzip* is in a *subdirectory* of *compress*, so you will only need `go doc compress/gzip`.

   Specific functions inside the "Go manual" can also be accessed. For instance the function `Printf` is described in *fmt*, but to only view the documentation concerning this function use: `go doc fmt Printf`.

   You can even display the source code with: `go doc -src fmt Printf`.

   All the built-in functions are also accesible by using go doc: `go doc builtin`.

2 Basics

> ”In Go, the code does exactly what it says on the page.”
>
> *Go Nuts mailing list*
> ANDREW GERRAND

There are three main components to make a package management system.

Software Build
: This component has tools for us to build software from source code in the form tar ball or source tree on source code management system.

Software Package Creation

Package Distribution
: Communication with these goroutines is done via channels [24, 19];

Erlang [6] also shares some of the features of Go. Notable differences between Erlang and Go is that Erlang borders on being a functional language, where Go is an imperative one. And Erlang runs in a virtual machine, while Go is compiled. Go also has a much more Unix-like feeling to it.

Hello World

In the Go tutorial, Go is presented to the world in the typical manner: letting it print ”Hello World” (Ken Thompson and Dennis Ritchie started this when they presented the C language in the nineteen seventies). We don’t think we can do better, so here it is, ”Hello World” in Go.

Listing 2.1. Hello world

```
package main ❶                                                          1

import "fmt"  // Implements formatted I/O. ❷                           3

/* Print something */ ❸                                                5
func main() { ❹                                                         6
❺                                                                       7
 fmt.Printf("Hello, world; or καλημέρα  κόσμε; or  こんにちは世界\n")      8
}                                                                       9
```

Lets look at the program line by line.

❶ This first line is just required. All Go files start with **package** `<something>`, **package** `main` is required for a standalone executable;

❶ This says we need *"fmt"* in addition to *main*. A package other than *main* is commonly called a library, a familiar concept of many programming languages (see chapter **??**). The line ends with a comment which is started with `//`;

❷ This is also a comment, but this one is enclosed in `/*` and `*/`;

❸ Just as **`package`** `main` was required to be first, **`import`** may come next. In Go, **`package`** is always first, then **`import`**, then everything else. When your Go program is executed, the first function called will be `main.main()`, which mimics the behavior from C. Here we declare that function;

❹ On line 8 we call a function from the package *fmt* to print a string to the screen. The string is enclosed with `"` and may contain non-ASCII characters. Here we use Greek and Japanese.

Compiling and running code

The preferred way to build a Go program, is to use a the `go` tool.

To build `helloworld` we just give:

```
% go build helloworld.go
```

This results in a executable called `helloworld`.

```
% ./helloworld
Hello, world; or καλημέρα  κόσμε; or  こんにちは世界
```

Settings used in this book

- Go itself is installed in `~/go` ;
- Go source code we want to compile ourself is placed in `~/g/src` and `$GOPATH` is set to `GOPATH=~/g` . This variable comes into play when start using packages (chapter **??**).

Variables, types and keywords

In the next sections we will look at variables, basic types, keywords and control structures of our new language. Go has a C-like feel when it comes to its syntax. If you want to put two (or more) statements on one line, they must be separated with a semicolon (';'). Normally you don't need the semicolon.

Go is different from other languages in that the type of a variable is specified *after* the variable name. So not: **`int`** `a`, but `a` **`int`**. When declaring a variable it is assigned the "natural" null value for the type. This means that after **`var`** `a` **`int`**, `a` has a value of 0. With **`var`** `s` **`string`**, `s` is assigned the zero string, which is `""`.

Declaring and assigning in Go is a two step process, but they may be combined. Compare the following pieces of code which have the same effect.

Listing 2.2. Declaration with =

```
var a int
var b bool
a = 15
b = false
```

Listing 2.3. Declaration with :=

```
a := 15
b := false
```

On the left we use the **var** keyword to declare a variable and *then* assign a value to it. The code on the right uses **:=** to do this in one step (this form may only be used *inside* functions). In that case the variable type is *deduced* from the value. A value of 15 indicates an **int**, a value of `false` tells Go that the type should be **bool**. Multiple **var** declarations may also be grouped, **const** and **import** also allow this. Note the use of parentheses:

```
var (
    x int
    b bool
)
```

Multiple variables of the same type can also be declared on a single line: **var** `x, y` **int**, makes `x` and `y` both **int** variables. You can also make use of parallel assignment:

```
a, b := 20, 16
```

Which makes `a` and `b` both integer variables and assigns 20 to `a` and 16 to `b`.

A special name for a variable is _ (underscore) . Any value assigned to it is discarded. In this example we only assign the integer value of 35 to `b` and discard the value 34.

```
_, b := 34, 35
```

Declared, but otherwise unused variables are a compiler error in Go. The following code generates this error: `i declared and not used`

```
package main
func main() {
    var i int
}
```

Boolean types

A boolean type represents the set of boolean truth values denoted by the predeclared constants *true* and *false*. The boolean type is **bool**.

Numerical types

Go has the well known types such as **int**, this type has the appropriate length for your machine. Meaning that on a 32 bits machine they are 32 bits, and on a 64 bits machine they are 64 bits. Note: an **int** is either 32 or 64 bits, no other values are defined. Same goes for **uint**.

If you want to be explicit about the length you can have that too with **int32**, or **uint32**. The full list for (signed and unsigned) integers is **int8**, **int16**, **int32**, **int64** and **byte**, **uint8**,

uint16, **uint32**, **uint64**. With **byte** being an alias for **uint8**. For floating point values there is **float32** and **float64** (there is no `float` type). A 64 bit integer or floating point value is *always* 64 bit, also on 32 bit architectures.

Note however that these types are all distinct and assigning variables which mix these types is a compiler error, like in the following code:

Listing 2.4. Familiar types are still distinct

```
package main                                                              1

func main() {                                                             3
        var a int           ⟵ Generic integer type                        4
        var b int32         ⟵ 32 bits integer type                        5
        a = 15                                                            6
        b = a + a           ⟵ Illegal mixing of these types               7
        b = b + 5           ⟵ 5 is a (typeless) constant, so this is OK   8
}                                                                         9
```

Gives the error on the assignment on line 7:

```
types.go:7: cannot use a + a (type int) as type int32 in assignment
```

The assigned values may be denoted using octal, hexadecimal or the scientific notation: `077`, `0xFF`, `1e3` or `6.022e23` are all valid.

Constants

Constants in Go are just that — constant. They are created at compile time, and can only be numbers, strings or booleans; **const** `x = 42` makes `x` a constant. You can use **iota** [a] to enumerate values.

```
const (
        a = iota
        b = iota
)
```

The first use of **iota** will yield 0, so `a` is equal to 0, whenever **iota** is used again on a new line its value is incremented with 1, so `b` has a value of 1.

You can even do the following, let Go repeat the use of **= iota**:

```
const (
        a = iota
        b               ⟵ Implicitly b = iota
)
```

You may also explicitly type a constant, if you need that:

```
const (
        a = 0               ⟵ Is an int now
        b string = "0"
)
```

[a] The word [iota] is used in a common English phrase, 'not one iota', meaning 'not the slightest difference', in reference to a phrase in the New Testament: "*until heaven and earth pass away, not an iota, not a dot, will pass from the Law*." [25]

Strings

An important other built in type is **`string`**. Assigning a string is as simple as:

```
s := "Hello World!"
```

Strings in Go are a sequence of UTF-8 characters enclosed in double quotes (”). If you use the single quote (’) you mean one character (encoded in UTF-8) — which is *not* a **`string`** in Go.

Once assigned to a variable the string can not be changed anymore: strings in Go are immutable. For people coming from C, the following is not legal in Go:

```
var s string = "hello"
s[0] = 'c'      ⟵ Change first char. to 'c', this is an error
```

To do this in Go you will need the following:

```
s := "hello"
c := []byte(s)       ❶
c[0] = 'c'           ❷
s2 := string(c)      ❸
fmt.Printf("%s\n", s2) ❹
```

❶ Convert `s` to an array of bytes, see chapter **??** section ”**??**” on page **??**;

❷ Change the first element of this array;

❸ Create a *new* string `s2` with the alteration;

❹ print the string with `fmt.Printf`.

Multi-line strings

Due to the insertion of semicolons (see [10] section “Semicolons”), you need to be careful with using multi line strings. If you write:

```
s := "Starting part"
    + "Ending part"
```

This is transformed into:

```
s := "Starting part";
    + "Ending part";
```

Which is not valid syntax, you need to write:

```
s := "Starting part" +
     "Ending part"
```

Then Go will not insert the semicolons in the wrong places. Another way would be to use *raw* string literals by using back quotes: `` ` ``:

```
s := `Starting part
     Ending part`
```

> Be aware that in this last example `s` now also contains the newline. Unlike *interpreted* string literals a raw string literal's value is composed of the *uninterpreted* characters between the quotes.

Runes

`Rune` is an alias for int.

Complex numbers

Go has native support for complex numbers. If you use them you need a variable of the type **`complex128`** (64 bit imaginary part). If you want something smaller there is **`complex64`** – for a 32 bits imaginary part. Complex numbers are written as `re + imi`, where `re` is the real part, `im` is the imaginary part and i is the literal 'i' ($\sqrt{-1}$). An example of using complex numbers:

```
var c complex64 = 5+5i;fmt.Printf("Value is: %v", c)
```
will print: `(5+5i)`

Errors

Any non-trivial program will have the need for error reporting sooner or later. Because of this Go has a builtin type specially for errors, called **`error`**.

Operators and built-in functions

Go supports the normal set of numerical operations, table 2.1 lists the current ones and their relative precedence. They all associate from left to right.

Table 2.1. Operator precedence

Precedence	Operator(s)
Highest	`* / % << >> & &^`
	`+ - | ^`
	`== != < <= > >=`
	`<-`
	`&&`
Lowest	`||`

`+ - * /` and `%` all do what you would expect, `& | ^` and `&^` are bit operators for bitwise and, bitwise or, bitwise xor and bit clear respectively. The `&&` and `||` operators are logical and and logical or. Not listed in the table is the logical not: `!`

Although Go does not support operator overloading (or method overloading for that matter), some of the built-in operators *are* overloaded. For instance `+` can be used for integers, floats, complex numbers and strings (adding strings is concatenating them).

Go keywords

Table 2.2. Keywords in Go

break	**default**	**func**	**interface**	**select**
case	**defer**	**go**	**map**	**struct**
chan	**else**	**goto**	**package**	**switch**
const	**fallthrough**	**if**	**range**	**type**
continue	**for**	**import**	**return**	**var**

Table 2.2 lists all the keywords in Go. In the following paragraphs and chapters we will cover them. Some of these we have seen already.

- For **var** and **const** see section "Variables, types and keywords" on page 7;
- **package** and **import** are briefly touched upon in section "Hello World". In chapter **??** they are documented in more detail.

Others deserve more text and have their own chapter/section:

- **func** is used to declare functions and methods;
- **return** is used to return from functions, for both **func** and **return** see chapter **??** for the details;
- **go** is used for concurrency (chapter **??**);
- **select** used to choose from different types of communication, see chapter **??**;
- **interface** see chapter **??**;
- **struct** is used for abstract data types, see chapter **??**;
- **type** also see chapter **??**.

Control structures

There are only a few control structures in Go [b]. For instance there is no do or while loop, only a **for**. There is a (flexible) **switch** statement and **if** and **switch** accept an optional initialization statement like that of **for**. There also is something called a type switch and a multiway communications multiplexer, **select** (see chapter **??**). The syntax is different (from that in C): parentheses are not required and the body must *always* be brace-delimited.

[b] This section is copied from [10].

If-else

In Go an **if** looks like this:

```
if x > 0 {          ⟵ { is mandatory
    return y
} else {
    return x
}
```

Mandatory braces encourage writing simple **if** statements on multiple lines. It is good style to do so anyway, especially when the body contains a control statement such as a **return** or **break**.

Since **if** and **switch** accept an initialization statement, it's common to see one used to set up a (local) variable.

```
if err := file.Chmod(0664); err != nil {      ⟵ nil is like C's NULL
    log.Stderr(err)     ⟵ Scope of err is limited to if's body
    return err
}
```

You can use the logical operators (see table 2.1) as you would normally do:

```
if true && true  {
    println("true")
}
if ! false {
    println("true")
}
```

In the Go libraries, you will find that when an **if** statement doesn't flow into the next statement – that is, the body ends in **break**, **continue**, **goto**, or **return** – the unnecessary **else** is omitted.

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
doSomething(f)
```

This is a example of a common situation where code must analyze a sequence of error possibilities. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in **return** statements, the resulting code needs no **else** statements.

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    return err
}
doSomething(f, d)
```

Syntax wise the following is illegal in Go:

```
if err != nil
{                       ⟵ Must be on the same line as the if
    return err
}
```

See [10] section “Semicolons” for the deeper reasons behind this.

> Ending with **if-then-else**
>
> Note that if you end a function like this:
>
> ```
> if err != nil {
>     return err
> } else {
>     return nil
> }
> ```
>
> It will not compile. This is a bug in the Go compiler. See [15] for an extended problem description and hopefully a fix.

Goto

Go has a **goto** statement — use it wisely. With **goto** you jump to a label which must be defined within the current function. For instance a loop in disguise:

```
func myfunc() {
        i := 0
Here:           ⟵ First word on a line ending with a colon is a label
        println(i)
        i++
        goto Here     ⟵ Jump
}
```

The name of the label is case sensitive.

For

The Go **for** loop has three forms, only one of which has semicolons.

```
for init; condition; post { }     ⟵ Like a C for

for condition { }                 ⟵ Like a while

for { }                           ⟵  Like a C for(;;) (endless loop)
```

Short declarations make it easy to declare the index variable right in the loop.

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i        ⟵ Short for sum = sum + i
}       ⟵ i ceases to exist after the loop
```

Finally, since Go has no comma operator and ++ and - - are statements not expressions, if you want to run multiple variables in a **for** you should use parallel assignment.

```
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {     ⟵ Parallel assignment
    a[i], a[j] = a[j], a[i]     ⟵ Here too
}
```

Break and continue

With **break** you can quit loops early. By itself, **break** breaks the current loop.

```
for i := 0; i < 10; i++ {
    if i > 5 {
        break     ⟵ Stop this loop, making it only print 0 to 5
    }
    println(i)
}
```

With loops within loops you can specify a label after **break**. Making the label identify *which* loop to stop:

```
J:  for j := 0; j < 5; j++ {
        for i := 0; i < 10; i++ {
            if i > 5 {
                break J     ⟵ Now it breaks the j-loop, not the i one
            }
            println(i)
        }
    }
```

With **continue** you begin the next iteration of the loop, skipping any remaining code. In the same way as **break**, **continue** also accepts a label. The following loop prints 0 to 5.

```
for i := 0; i < 10; i++ {
    if i > 5 {
        continue     ⟵ Skip the rest of the remaining code in the loop
    }
    println(i)
}
```

Range

The keyword **range** can be used for loops. It can loop over slices, arrays, strings, maps and channels (see chapter **??**). **range** is an iterator that, when called, returns a key-value pair from the thing it loops over. Depending on what that is, **range** returns different things.

When looping over a slice or array **range** returns the index in the slice as the key and value belonging to that index. Consider this code:

```
list := []string{"a", "b", "c", "d", "e", "f"}     ❶
for k, v := range list {     ❷
```

```
    // do what you want with k and v ❷
}
```

❶ Create a slice (see ”Arrays, slices and maps” on page 19)) of strings.

❷ Use **range** to loop over them.With each iteration **range** will return the index as **int** and the key as a **string**, starting with 0 and ”a”.

❸ k will have the value 0...5, and v will loop through ”a”...”f”.

You can also use **range** on strings directly. Then it will break out the individual Unicode characters [c] and their start position, by parsing the UTF-8. The loop:

```
for pos, char := range "aΦx" {
    fmt.Printf("character '%c' starts at byte position %d\n", char, pos)
}
```

prints

```
character 'a' starts at byte position 0
character 'Φ' starts at byte position 1
character 'x' starts at byte position 3    ← Φ took 2 bytes
```

Switch

Go’s **switch** is very flexible. The expressions need not be constants or even integers, the cases are evaluated top to bottom until a match is found, and if the **switch** has no expression it switches on **true**. It’s therefore possible – and idiomatic – to write an **if-else-if-else** chain as a **switch**.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

There is no automatic fall through, you can however use **fallthrough** to do just that. Without **fallthrough**:

```
switch i {
    case 0:  // empty case body
    case 1:
        f()  // f is not called when i == 0!
}
```

[c] In the UTF-8 world characters are sometimes called runes. Mostly, when people talk about characters, they mean 8 bit characters. As UTF-8 characters may be up to 32 bits the word rune is used.

And with:

```
switch i {
    case 0:  fallthrough
    case 1:
        f()  // f is called when i == 0!
}
```

With **default** you can specify an action when none of the other cases match.

```
switch i {
    case 0:
    case 1:
        f()
    default:
        g()     // called when i is not 0 or 1
}
```

Cases can be presented in comma-separated lists.

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+':     ⟵ , as "or"
        return true
    }
    return false
}
```

Here's a comparison routine for byte arrays that uses two **switch** statements:

```
// Compare returns an integer comparing the two byte arrays
// lexicographically.
// The result will be 0 if a == b, -1 if a < b, and +1 if a > b
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    // String are equal except for possible tail
    switch {
    case len(a) < len(b):
        return -1
    case len(a) > len(b):
        return 1
    }
    return 0    // Strings are equal
}
```

Built-in functions

A small number of functions are predefined, meaning you *don't* have to include any package to get access to them. Table 2.3 lists them all.[d]

Table 2.3. Pre–defined functions in Go

close	**new**	**panic**	**complex**
delete	**make**	**recover**	**real**
len	**append**	**print**	**imag**
cap	**copy**	**println**	

These built-in functions are documented in the *builtin* pseudo package that is included in recent Go releases.

close is used in channel communication. It closes a channels, see chapter **??** for more on this.

delete is used for deleting entries in maps.

len and *cap* are used on a number of different types, `len` is used for returning the length of strings and the length of slices and arrays. See section "Arrays, slices and maps" for the details of slices and arrays and the function `cap`.

new is used for allocating memory for user defined data types. See section "**??**" on page **??**.

make is used for allocating memory for built-in types (maps, slices and channels). See section "**??**" on page **??**.

copy is used for copying slices. `append` is for concatenating slices. See section "Slices" in this chapter.

panic and *recover* are used for an *exception* mechanism. See the section "**??**" on page **??** for more.

print and *println* are low level printing functions that can be used without reverting to the *fmt* package. These are mainly used for debugging.

complex, *real* and *imag* all deal with complex numbers. Other than the simple example we gave, we will not further explain complex numbers.

[d] You can use the command `go doc builtin` to read the online documentation about the built-in types and functions.

Arrays, slices and maps

Storing multiple values in a list can be done by utilizing arrays, or their more flexible cousin: slices. A dictionary or hash type is also available, it is called a **map** in Go.

Arrays

An array is defined by: `[n]<type>`, where n is the length of the array and `<type>` is the stuff you want to store. Assigning or indexing an element in the array is done with square brackets:

```
var arr [10]int
arr[0] = 42
arr[1] = 13
fmt.Printf("The first element is %d\n", arr[0])
```

Array types like **var** `arr = [10]`**int** have a fixed size. The size is *part* of the type. They can't grow, because then they would have a different type. Also arrays are values: Assigning one array to another *copies* all the elements. In particular, if you pass an array to a function, it will receive a copy of the array, not a pointer to it.

To declare an array you can use the following: **var** `a [3]`**int**, to initialize it to something else than zero, use a composite literal: `a := [3]`**int{**`1, 2, 3`**}** and this can be shortened to `a := [...]`**int{**`1, 2, 3`**}**, where Go counts the elements automatically. Note that all fields must be specified. So if you are using multidimensional arrays you have to do quite some typing:

*A composite literal allows you to assign a value directly to an array, slice or map. See the section "**??**" on page **??** for more.*

```
a := [2][2]int{ [2]int{1,2}, [2]int{3,4} }
```

Which is the same as:

```
a := [2][2]int{ [...]int{1,2}, [...]int{3,4} }
```

When declaring arrays you *always* have to type something in between the square brackets, either a number or three dots (`...`) when using a composite literal. Since release `2010-10-27` this syntax was further simplified. From the release notes:

Go release `2010-10-27` [12].

> *The syntax for arrays, slices, and maps of composite literals has been simplified. Within a composite literal of array, slice, or map type, elements that are themselves composite literals may elide the type if it is identical to the outer literal's element type.*

This means our example can become:

```
a := [2][2]int{ {1,2}, {3,4} }
```

Slices

TODO
Add push/pop to this section as container/vector will be deprecated.

A slice is similar to an array, but it can grow when new elements are added. A slice always refers to an underlying array. What makes slices different from arrays is that a slice is a pointer *to* an array; slices are reference types, which means that if you assign one slice to another, both refer to the same underlying array. For instance, if a function takes a

*Reference types are created with **make**.*

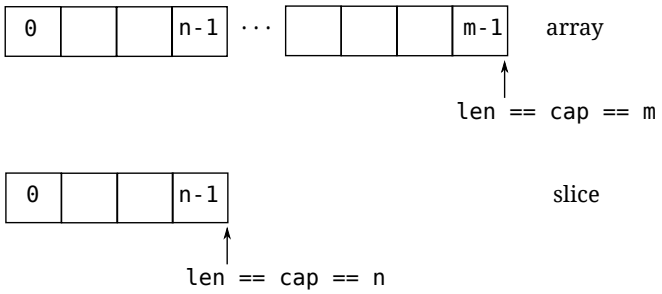
slice argument, changes it makes to the elements of the slice will be visible to the caller, analogous to passing a pointer to the underlying array. With:

```
sl := make([]int, 10)
```

you create a slice which can hold ten elements. Note that the underlying array isn't specified. A slice is always coupled to an array that has a fixed size. For slices we define a capacity and a length. Figure 2.1 depicts the following Go code. First we create an array of m elements of the type **int**: `var array[m]int`
Next, we create a slice from this array: `slice := array[0:n]`
And now we have:

- `len(slice)== n == cap(slice)== n ;`
- `len(array)== cap(array)== m .`

Figure 2.1. Array versus slice



Given an array, or another slice, a new slice is created via `a[I:J]`. This creates a new slice which refers to the variable `a`, starts at index `I`, and ends before index `J`. It has length `J - I`.

```
// array[n:m], create a slice from array with elements n to m-1
a := [...]int{1, 2, 3, 4, 5} ❶
s1 := a[2:4] ❶
s2 := a[1:5] ❷
s3 := a[:]  ❸
s4 := a[:4] ❹
s5 := s2[:] ❺
```

❶ Define an array with 5 elements, from index 0 to 4;

❶ Create a slice with the elements from index 2 to 3, this contains: `3, 4`;

❷ Create a slice with the elements from index 1 to 4, contains: `2, 3, 4, 5`;

❸ Create a slice with all the elements of the array in it. This is a shorthand for: `a[0:len(a)]`;

❹ Create a slice with the elements from index 0 to 3, this is thus short for: `a[0:4]`, and yields: `1, 2, 3, 4`;

❺ Create a slice from the slice `s2`, note that `s5` still refers to the array `a`.

In the code listed in 2.5 we dare to do the impossible on line 8 and try to allocate something beyond the capacity (maximum length of the underlying array) and we are greeted with a *runtime* error.

Listing 2.5. Arrays and slices

```
package main                                                              1

func main() {                                                             3
        var array [100]int     // Create array, index from 0 to 99        4
        slice := array[0:99]   // Create slice, index from 0 to 98        5

        slice[98] = 'a'        // OK                                      7
        slice[99] = 'a'        // Error: "throw: index out of range"      8
}                                                                         9
```

If you want to extend a slice, there are a couple of built-in functions that make life easier: **`append`** and **`copy`**. From [11]:

> *The function **`append`** appends zero or more values `x` to a slice `s` and returns the resulting slice, with the same type as `s`. If the capacity of `s` is not large enough to fit the additional values, **`append`** allocates a new, sufficiently large slice that fits both the existing slice elements and the additional values. Thus, the returned slice may refer to a different underlying array.*

```
s0 := []int{0, 0}
s1 := append(s0, 2)        ❶
s2 := append(s1, 3, 5, 7)  ❷
s3 := append(s2, s0...)    ❸
```

❶ append a single element, `s1 == []int{0, 0, 2}`;

❷ append multiple elements, `s2 == []int{0, 0, 2, 3, 5, 7}`;

❸ append a slice, `s3 == []int{0, 0, 2, 3, 5, 7, 0, 0}`. Note the three dots!

And

> *The function **`copy`** copies slice elements from a source `src` to a destination `dst` and returns the number of elements copied. Source and destination may overlap. The number of arguments copied is the minimum of **`len`**`(src)` and **`len`**`(dst)`.*

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
n1 := copy(s, a[0:])        ⟵ n1 == 6, s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])        ⟵ n2 == 4, s == []int{2, 3, 4, 5, 4, 5}
```

Maps

Many other languages have a similar type built-in, Perl has hashes, Python has its dictionaries and C++ also has maps (as part of the libraries) for instance. In Go we have the **map** type. A **map** can be thought of as an array indexed by strings (in its most simple form). In the following listing we define a **map** which converts from a **string** (month abbreviation) to an **int** – the number of days in that month. The generic way to define a map is with: `map[<from type>]<to type>`

```
monthdays := map[string]int{
        "Jan": 31, "Feb": 28, "Mar": 31,
        "Apr": 30, "May": 31, "Jun": 30,
        "Jul": 31, "Aug": 31, "Sep": 30,
        "Oct": 31, "Nov": 30, "Dec": 31,     ⟵ The comma here is required
}
```

Note to use **make** when only declaring a **map**: `monthdays := make(map[string]int)`

For indexing (searching) in the map, we use square brackets. For example, suppose we want to print the number of days in December: `fmt.Printf("%d\n", monthdays["Dec"])` If you are looping over an array, slice, string, or map a **range** clause will help you again, which returns the key and corresponding value with each invocation.

```
year := 0
for _, days := range monthdays {        ⟵ Key is not used, hence _, days
    year += days
}
fmt.Printf("Numbers of days in a year: %d\n", year)
```

Adding elements to the **map** would be done as:

```
monthdays["Undecim"] = 30           ⟵ Add a month
monthdays["Feb"]     = 29           ⟵ Overwrite entry - for leap years
```

To test for existence , you would use the following[21]:

```
var value int
var present bool

value, present = monthdays["Jan"]     ⟵ If exist, present has the value true
                                      ⟵ Or better and more Go like
v, ok := monthdays["Jan"]             ⟵ Hence, the "comma ok" form
```

And finally you can remove elements from the **map**:

```
delete(monthdays, "Mar")               ⟵ Deletes "Mar", always rainy anyway
```

In general the syntax `delete(m, x)` will delete the map entry retrieved by the expression `m[x]`.

Exercises

Q2. (1) For-loop

1. Create a simple loop with the **`for`** construct. Make it loop 10 times and print out the loop counter with the *fmt* package.
2. Rewrite the loop from 1. to use **`goto`**. The keyword **`for`** may not be used.
3. Rewrite the loop again so that it fills an array and then prints that array to the screen.

Q3. (1) FizzBuzz

1. Solve this problem, called the Fizz-Buzz [23] problem:

   Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

Q4. (1) Strings

1. Create a Go program that prints the following (up to 100 characters):

   ```
   A
   AA
   AAA
   AAAA
   AAAAA
   AAAAAA
   AAAAAAA
   ...
   ```

2. Create a program that counts the numbers of characters in this string:

   ```
   asSASA ddd dsjkdsjs dk
   ```

   Make it also output the number of bytes in that string. *Hint.* Check out the *utf8* package.
3. Extend the program from the previous question to replace the three runes at position 4 with 'abc'.
4. Write a Go program that reverses a string, so "foobar" is printed as "raboof". *Hint.* Unfortunately you need to know about conversion, skip ahead to section "**??**" on page **??**.

Q5. (4) Average

1. Give the code that calculates the average of a **`float64`** slice. In a later exercise (Q**??** you will make it into a function.

Answers

A2. (1) For-loop

1. There are a multitude of possibilities, one of the solutions could be:

Listing 2.6. Simple for loop

```
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {     ⟵ See section For on page 14
                fmt.Printf("%d\n", i)
        }
}
```

Let's compile this and look at the output.

```
% 6g for.go && 6l -o for for.6
% ./for
0
1
.
.
.
9
```

2. Rewriting the loop results in code that should look something like this (only showing the `main`-function):

```
func main() {
        i := 0              ⟵ Define our loop variable
I:                          ⟵ Define a label
        fmt.Printf("%d\n", i)
        i++
        if i < 10 {
                goto I      ⟵ Jump back to the label
        }
}
```

3. The following is one possible solution:

Listing 2.7. For loop with an array

```
func main() {
        var arr [10]int     ⟵ Create an array with 10 elements
        for i := 0; i < 10; i++ {
                arr[i] = i     ⟵ Fill it one by one
        }
```

```
        fmt.Printf("%v", arr)     ⟵ With %v Go prints the type
}
```

You could even do this in one fell swoop by using a composite literal:

```
a := [...]int{0,1,2,3,4,5,6,7,8,9}     ⟵ With [...] you let Go count
fmt.Printf("%v\n", a)
```

A3. (1) FizzBuzz

1. A possible solution to this simple problem is the following program.

Listing 2.8. Fizz-Buzz

```
package main

import "fmt"

func main() {
        const (
                FIZZ = 3 ❶
                BUZZ = 5
        )
        var p bool      ❶
        for i := 1; i < 100; i++ { ❷;
                p = false
                if i%FIZZ == 0 { ❸
                        fmt.Printf("Fizz")
                        p = true
                }
                if i%BUZZ == 0 { ❹
                        fmt.Printf("Buzz")
                        p = true
                }
                if !p { ❺
                        fmt.Printf("%v", i)
                }
                fmt.Println() ❻
        }
}
```

❶ Define two constants to make the code more readable. See section ”Constants”;

❶ Holds if we already printed someting;

❷ for-loop, see section ”For”

❸ If divisible by FIZZ, print ”Fizz”;

❹ And if divisble by BUZZ, print ”Buzz”. Note that we have also taken care of the FizzBuzz case;

❺ If neither FIZZ nor BUZZ printed, print the value;

❻ Format each output on a new line.

A4. (1) Strings

1. This program is a solution:

Listing 2.9. Strings

```
package main

import "fmt"

func main() {
        str := "A"
        for i := 0; i < 100; i++ {
                fmt.Printf("%s\n", str)
                str = str + "A"     ⟵ String concatenation
        }
}
```

2. To answer this question we need some help of the *unicode/utf8* package. First we check the documentation with `go doc unicode/utf8 | less`. When we read the documentation we notice `func RuneCount(p []byte)int`. Secondly we can convert *string* to a `byte` slice with

```
str := "hello"
b   := []byte(str)     ⟵ Conversion, see page ??
```

Putting this together leads to the following program.

Listing 2.10. Runes in strings

```
package main

import (
        "fmt"
        "unicode/utf8"
)

func main() {
        str := "dsjkdshdjsdh....js"
        fmt.Printf("String %s\nLenght: %d, Runes: %d\n", str,
                len([]byte(str)), utf8.RuneCount([]byte(str)))
}
```

3. Reversing a string can be done as follows. We startfrom the left (`i`) and the right (`j`) and swap the characters as we see them:

Listing 2.11. Reverse a string

```
import "fmt"

func main() {
        s := "foobar"
        a := []byte(s)     ⟵ Again a conversion
        // Reverse a
        for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
                a[i], a[j] = a[j], a[i]     ⟵ Parallel assignment
        }
        fmt.Printf("%s\n", string(a))     ⟵ Convert it back
}
```

A5. (4) Average

1. The following code calculates the average.

```
sum := 0.0
switch len(xs) {
case 0:                 ❶
        avg = 0
default:                ❷
        for _, v := range xs {
                sum += v
        }
        avg = sum / float64(len(xs)) ❸
}
```

❶ If the length is zero, we return 0;

❷ Otherwise we calculate the average;

❸ We have to convert the value to a **`float64`** to make the division work.

A Colophon

This work was created with LaTeX. The main text is set in the Google Droid fonts. All typewriter text is typeset in DejaVu Mono.

Contributors

The following people have helped to make this book what it is today.

- Miek Gieben <miek@miek.nl>;
- JC van Winkel.

Help with proof reading, checking exercises and text improvements (no particular order and either real name or an alias): *Anthony Magro*, *Uriel*.

T.J. Yang

Miek Gieben has a master's degree in Computer Science from the Radboud University Nijmegen (Netherlands). He is involved in the development and now the deployment of the DNSSEC protocol – the successor of the DNS and as such co-authored [9].

After playing with the language Erlang, Go was the first concurrent language that actually stuck with him.

He fills his spare time with coding in, and writing of Go. He is the maintainer of the Go DNS library: `https://github.com/miekg/godns`. He maintains a personal blog on `http://www.miek.nl` and tweets under the name `@miekg`. The postings and tweets may sometimes actually have to do something with Go.

License and copyright

B Index

C Bibliography

[1] LAMP Group at EPFL. Scala. http://www.scala-lang.org/, 2003.

[2] Haskell Authors. Haskell. http://www.haskell.org/, 1990.

[3] Inferno Authors. Inferno. http://www.vitanuova.com/inferno/, 1995.

[4] Plan 9 Authors. Plan 9. http://plan9.bell-labs.com/plan9/index.html, 1992.

[5] Plan 9 Authors. Limbo. http://www.vitanuova.com/inferno/papers/limbo.html, 1995.

[6] Ericsson Cooperation. Erlang. http://www.erlang.se/, 1896.

[7] Brian Kernighan Dennis Ritchie. The C programming language, 1975.

[8] Larry Wall et al. Perl. http://perl.org/, 1987.

[9] Kolkman & Gieben. Dnssec operational practices. http://www.ietf.org/rfc/rfc4641.txt, 2006.

[10] Go Authors. Effective Go. http://golang.org/doc/effective_go.html, 2010.

[11] Go Authors. Go language specification. http://golang.org/doc/go_spec.html, 2010.

[12] Go Authors. Go release history. http://golang.org/doc/devel/release.html, 2010.

[13] Go Authors. Go tutorial. http://golang.org/doc/go_tutorial.html, 2010.

[14] Go Authors. Go website. http://golang.org/, 2010.

[15] Go Community. Go issue 65: Compiler can't spot guaranteed return in if statement. http://code.google.com/p/go/issues/detail?id=65, 2010.

[16] Go Community. Go nuts mailing list. http://groups.google.com/group/golang-nuts, 2010.

[17] James Gosling et al. Java. http://oracle.com/java/, 1995.

[18] C. A. R. Hoare. Quicksort. http://en.wikipedia.org/wiki/Quicksort, 1960.

[19] C. A. R. Hoare. Communicating sequential processes (csp). http://www.usingcsp.com/cspbook.pdf, 1985.

[20] Rob Pike. Newsqueak: A language for communicating with mice. http://swtch.com/~rsc/thread/newsqueak.pdf, 1989.

[21] Rob Pike. The Go programming language, day 2. http://golang.org/doc/{G}oCourseDay2.pdf, 2010.

[22] Bjarne Stroustrup. The C++ programming language, 1983.

[23] Imran On Tech. Using fizzbuzz to find developers... http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/, 2010.

[24] Wikipedia. Communicating sequential processes. http://en.wikipedia.org/wiki/Communicating_sequential_processes, 2010.

[25] Wikipedia. Iota. http://en.wikipedia.org/wiki/Iota, 2010.

This page is intentionally left blank.