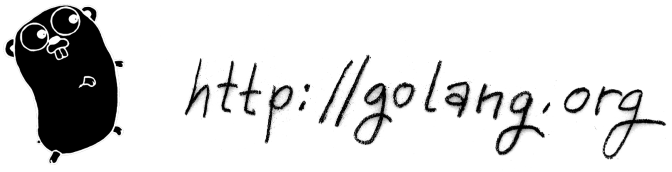
Learning Go



Authors:
Miek Gieben

Thanks to:
Go authors
Google
Go Nuts mailing list

THIS IS A WORK IN PROGRESS

Version: 0.1 (August 16, 2010)

Google™

Go is fun!

Table of Contents

List of Figures

List of Tables

List of Code Examples

List of Exercises

1 Introduction

"I am interested in this and hope to do something."

On adding complex numbers to Go
Ken Thompson

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

Its intended audience is people who are familiar with programming and know different languages, be it C[12], C^{++}[19], Java [13], Erlang[11], Scala[1] or Haskell[6]. 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, the best way to do this is to discover it for yourself by creating your own programs. Each chapter 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 50[1], where 0 is easy and 50 is extremely difficult (such a problem is a unresolved research problem). Then a short name is given, for easier reference. For example

Q1. (4) A map function ...

gives a question numbered **Q1** of a level 4 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.

> The Go language is a young language and features are still added or even *removed*. It may be possible that some text is outdated when you read it. It is also possible that some answers to the exercises are too old. 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*;
- Line numbers are printed on the right side;

[1]This is the same scale Donald Knuth uses.

- Longer remarks get a number - ❶ - with the explanation following;
- Shell examples use a % as prompt.

Official documentation

There already is a substantial amount of documentation written about Go. The Go Tutorial [5], and the Effective Go document [2]. The website `http://golang.org/doc/` is a very good starting point for reading up on Go[2]. This book does not require that you have read those documents, but it will help, if you did.

Getting Go

There are currently (2010) no packages for Go in any Linux distribution. The route to install Go is thus slightly longer than it could be. When Go stabilizes this situation will change probably. For now, you need to retrieve the code from the mercurial archive and compile Go yourself.

- First install Mercurial (to get the `hg` command). In Ubuntu/Debian/Fedora you must install the `mercurial` package;
- For building Go you also need the packages: `bison`, `gcc`, `libc6-dev`, `ed`, `gawk` and `make`;
- Then retrieve the Go source code:
```
% export GOROOT=~/go # set the GOROOT environment variable
% hg clone -r release https://go.googlecode.com/hg/ $GOROOT
```
- Set some more environment variables, namely:

  GOOS
  : Must be set to your operation system, valid values are `linux`, `darwin` (for Mac OS X) and `freebsd`.

  GOARCH
  : Must be set the either `386` for 32 bit systems or to `amd64` for 64 bit.

  GOBIN
  : Can be optionally set to point to a directory where the Go binaries should be placed.

  In this book we use the following settings:
```
% export GOOS=linux
% export GOARCH=amd64
% export GOBIN=~/bin
```
- Compile Go
```
% cd $GOROOT/src
% ./all.bash
```

If all goes well, you should now have Go installed on your system and you can start playing.

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

Online documentation

Go comes with its own documentation in the form of a Go program called `godoc`. You can use it yourself to look in the online documentation. For instance, suppose we want to know more about the package *hash*. We would then give the command `godoc hash`. How to read this and how you can create your own package documentation is explained in chapter 4.

Origins

Go has it origins in Plan 9 [9]. Plan 9 is (or was supposed to be) the successor of Unix. As you know one of the core ideas of Unix is ”everything is a file”, so the `read()` and `write()` calls work as well on normal files as they do on I/O devices. However for some devices this has never happened, notably network and video devices. In Plan 9 this ”everything is a file”–idea is taken to the next level and truly everything is presented to the user as a file, including networking and video devices. This has nothing to do with Go per se, but Plan 9 included a language called Limbo [8]. 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.*

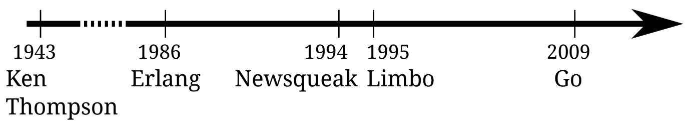
That sounds a lot like Go, and one could say Go is a reimplementation of Limbo on Unix–like systems like Linux. One feature of Limbo that is also included in Go is the excellent support for cross compiling.

Another feature Go inherited from Limbo is channels (see chapter 6). Again from the 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” [18], 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 [11].

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 [14], who incidentally is the same man that invented QuickSort [15].

> Go is the first C–like language that is widely available, runs on many different platforms and makes concurrency easy (or easier). Becoming proficient in a new language means doing your exercises, reading about it helps, but getting down to the little details when programming your self is even better.

Exercises

Q1. (1) Documentation

1. Go's documentation can be read with the `godoc`.
   `godoc hash` gives information about the *hash* package. Reading the documentation on *container* gives the following result:

   ```
   SUBDIRECTORIES

   heap
   list
   ring
   vector
   ```

   With which `godoc` command can you read the documentation of *vector* contained in *container*?

Answers

A1. (1) Documentation

1. The package *vector* is in a *subdirectory* of *container*, so you'll only need `godoc container/vector`.
   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: `godoc fmt Printf`.

2 Basics

”There and back again.”

Lord of the Rings
Bilbo Baggins

There are a few things that make Go different from most other languages out there.

Clean and Simple
: Go strives to keep things small and beautiful, you should be able to do a lot in only a few lines of code;

Concurrent
: Go makes it easy to ”fire off” functions to be run as *very* lightweight threads. These threads are called goroutines[1] in Go;

Channels
: Communication with these goroutines is done via channels [24][14];

Fast
: Compilation is fast and execution is fast. The aim is to be as fast as C. Compilation time is measured in seconds;

Safe
: Go has garbage collection, no more `free()` in Go, the language takes care of this;

Standard format
: A Go program can be formatted in (almost) any way the programmers want, *but* an official format exist. The rule is very simple: The output of the filter `gofmt` *is the official* endorsed format.

Postfix types
: Types are given *after* the variable name, thus `var a int`, instead of `int a;` as one would in C;

UTF-8
: UTF-8 is all over the place, in strings *and* in the program code. Finally you can use `Φ = Φ + 1` in your source code;

Open Source
: The Go license is completely open source, see the file LICENSE in the Go source code distribution;

Fun
: Programming with Go should be fun again!

[1] Yes, that sounds a lot like *co*routines, but goroutines are slightly different as we will see in chapter 6.

Erlang [11] 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                                                        1

import ”fmt”  // Implements formatted I/O.                          3

func main() {                                                       5
 fmt.Printf(”Hello, world; or καλημέρα κόσμϵ; or こんにちは世界\n”)    6
}                                                                   7
```

Go is 100% UTF-8 capable, you can put any Unicode *glyph* in your source — assuming you can type it on your keyboard. Go supports both the `/* */` and `//` types of comments. There is no need to put a semicolon at the end of each line (as you would in C).

Lets look at the program line by line. Line 1 is just required. All Go files start with **`package`** `<something>`. **`package`** `main` is required for a standalone executable. Line 3 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 4). 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.

Compiling and running code

As Go is a compiled language we first need to compile our program. Compiling in Go is *fast*, often measured in mere seconds, even for large programs.

```
% 6g helloworld.go                ← compiles to helloworld.6 (for 64 bit)
% 6l -o helloworld helloworld.6     ← linking stage
```

For 32 bit you should use `8g` and `8l`, this will then generate `helloworld.8`.[2] After compilation we can run it:

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

[2]This is the only thing you will need to change when you are cross compiling. Just use a different compiler and you are done.

Building a program

Another, less laborious, way to build a Go program, is to use a `Makefile`. The following one can be used to build `helloworld`:

Listing 2.2. `Makefile` for a program

```
# Copyright 2009 The Go Authors. All rights reserved.         1
# Use of this source code is governed by a BSD-style          2
# license that can be found in the LICENSE file.              3

include $(GOROOT)/src/Make.$(GOARCH)                          5

TARG=helloworld                                               7
GOFILES=\                                                     8
        helloworld.go\                                        9

include $(GOROOT)/src/Make.cmd                                11
```

A line 7 you specify the name for your compiled program and on line 9 you enumerate the source files. Now only an invocation of `make` is enough to get your program compiled.

Variables, types and keywords

In the next sections we will look at the variables, basic types, keywords and control structures of our new language. 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 2.3. Declaration with `=`

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

Listing 2.4. 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:[3] `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`** and **`float`**. These types have 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.

If you want to be explicit about the length you can have that too with **`int32`**, or **`uint32`**. The full (explicit) list for (signed and unsigned) integers is **`int8/16/32/64`**. With **`byte`** being an alias for **`uint8`**. For floating point values there only is **`float32`** and **`float64`**.

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

Listing 2.5. 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` When assigning values octal, hexadecimal and the scientific notation may be used: `077`, `0xFF`, `1e3` or `6.022 e23` are all valid.

[3]Sometimes this can be annoying.

Constants

Constants in Go are just that — constant. They are created at compile time, and can only be numbers, strings or boo leans; **const** x = 42 makes x a constant. You can use **iota**[4] to enumerate values.

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

You can even do the following. 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

```
const (
        a = iota
        b           // implicitly 'b = iota'
)
```

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 the first character 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("%v\n", s2)
```

❶ Convert s to an array of **byte**s, see chapter 5 section "Conversions";

❷ Change the first element of this array;

❸ Create a *new* string s2 with the alteration.

[4]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.*"[26]

Complex numbers

Go has native support for complex numbers. If you use them you need a variable of the type **`complex`**. If you need to specify the number of bits you have **`complex32`** and **`complex64`** for 32 and 64 bits. Complex numbers are written as `re + im`i, 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 complex = 5+5i;fmt.Printf("Value is: %v", c")
```

will print

```
(5+5i)
```

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 bit-wise and, or, xor and bit clear respectively. 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.

A small number of functions in Go are predefined, meaning they already exists and you *don't* have to include any package to get access to them. Table 2.2 lists them all.

Table 2.2. Pre–defined functions in Go

`close`	**`new`**	**`panic`**	**`cmplx`**
`closed`	**`make`**	**`panicnl`**	**`real`**
`len`	**`copy`**	**`print`**	**`imag`**
`cap`		**`printnl`**	

`cmplx`, *`real`* and *`imag`* all deal with complex numbers.

`close` and *`closed`* are used in channel communication and the closing of those channels, see chapter 6 for more on this.

`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.

make is used for allocating memory for built-in types (maps, slices and channels).

copy is used for copying slices. `new`, `make` and `copy` make their appearance in chapter 5.

panic and *panicln* used for an *exception* mechanism. See the section "Panic and recovering" on page 25 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.

Go keywords

Table 2.3. 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 2.3 lists all the keywords in Go. In the following paragraphs and chapters we will go over them.

Control structures

There are only a few control structure in Go[5]. 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 6). The syntax is different: parentheses are not required and the bodies must *always* be brace-delimited.

If

In Go a simple **if** looks like this:

```
if x > 0 {          ← { is mandatory
    return y
}                   ←
```

Mandatory braces encourage writing simple **if** statements on multiple lines. It's 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.

[5]This section is copied from [2].

```
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
}
```

In the Go libraries, you'll 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)
```

Goto

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

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

For

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

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

// Like a while
for condition { }
```

```
// Like a C for(;;)
for { }
```

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

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

Finally, since Go has no comma operator and ++ and - - are statements not expressions (see appendix B), 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

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

```
for i := 0; i < 10; i++ {
    println(i)
    if i > 5 {
        break     ← Breaks this loop, making it print 0 to 6
    }
}
```

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++ {
            println(i)
            if i > 5 {
                break J     ← Now it breaks the j-loop, not the i one
            }
        }
    }
```

Range

For strings, you can use the **range** clause to manage the loop for you. It breaks out the individual Unicode characters by parsing the UTF-8 (erroneous encodings consume one byte and produce the replacement rune[6] U+FFFD). The loop:

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

[6] 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.

prints

```
character 'a' starts at byte position 0
character 'Φ' starts at byte position 1
character 'x' starts at byte position 3
```

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, but 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
    }
```

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. 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 if *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.

```
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 (...).

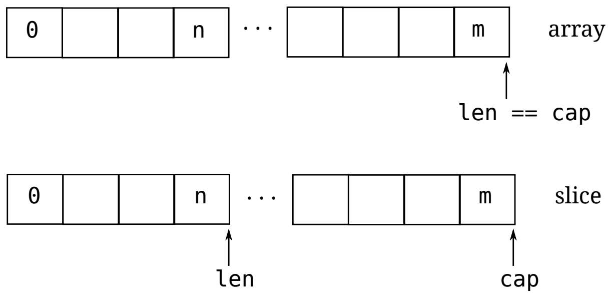
Slices

A slice refers to an under laying 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 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. 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+1]int`
Next, we create a slice from this array: `slice := array[0:n+1]`
And now we have:

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

Figure 2.1. Array versus slice



The following code defines an array and slice:

Listing 2.6. 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
```

On line 8 we dare to do to impossible and try to allocate something beyond the capacity (maximum length of the under laying array) and we are greeted with a *runtime* error.

Maps

Many other languages have a similar type built-in, Perl has hashes Python has its dictionaries and C++ also has maps (in *lib*) 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 OK
}
```

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're looping over an array, slice, string, or map a **range** clause help you again, which returns the key and corresponding value with each invocation.

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

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

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

To test for existence, you would use the following[17]:

```
var value int
var present bool

value, present = monthday["Jan"]        // Does it exist?
// Or better and more Go like
v, ok := monthday["Jan"]                // Hence, the "comma ok" form
```

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

```
monthday["Mar"] = 0, false    // Deletes "Mar", always rains anyway
```

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. Put the body of the loop in a separate function.
3. Rewrite the loop from 1. to use **goto**. The keyword **for** may not be used.

Q3. (1) FizzBuzz

1. Solve this problem, called the Fizz-Buzz [21] 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[7]. 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”.

[7]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.

Answers

A2. (1) For-loop

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

Listing 2.7. Simple for-loop

```
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {
                fmt.Printf("%d\n", i)
        }
}
```

Lets compile this on an Intel 386 Linux machine and look at the output.

```
% 8g for.go && 8l -o for for.8
% ./for
0
1
.
.
.
9
```

2. Next we put the body of the loop - the **`fmt.Printf`** - in a separate function.

Listing 2.8. Loop calls function

```
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {
                show(i)
        }
}

func show(j int) {
        fmt.Printf("%d\n", j)
}
```

The presented program should be self explanatory. Note however the "`j int`" instead of the more usual "`int j`" in the function definition.

A3. (1) FizzBuzz

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

Listing 2.9. Fizz-Buzz

```
package main

import ”fmt”

func main() {
	const (
		FIZZ = 3
		BUZZ = 5
	)
	for i := 1; i < 100; i++ {
		p := false
		if i%FIZZ == 0 {
			fmt.Printf(”Fizz”)
			p = true
		}
		if i%BUZZ == 0 {
			fmt.Printf(”Buzz”)
			p = true
		}
		if !p {
			fmt.Printf(”%v”, i)
		}
		fmt.Println()
	}
}
```

A4. (1) Strings

1. This program is a solution:

Listing 2.10. Strings

```
package main

import ”fmt”

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

2. To answer this question we need some help of the *utf8* package. First we check the documentation with `godoc 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)      ← A conversion, see section 5 on page 45
```

Putting this together leads to the following program.

Listing 2.11. Runes in strings

```
package main

import (
        "fmt"
        "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:

```
package main

import "fmt"

func main() {
    a := "my string"
    b := ""
    for _,k := range a {
        b = k + b
    }
    fmt.Printf("%s\n", b)
}
```

3 Functions

I'm always delighted by the light touch and stillness of early programming languages. Not much text; a lot gets done. Old programs read like quiet conversations between a well-spoken research worker and a well- studied mechanical colleague, not as a debate with a compiler. Who'd have guessed sophistication bought such noise?

Richard P. Gabriel

Functions are the basic building blocks in Go programs; all interesting stuff happens in them. A function is declared as follows:

Listing 3.1. A function declaration

```
type mytype int        ← Define a new type, see section 5 in chapter 5
```

$$\underbrace{\texttt{func}}_{❶}\ \underbrace{\texttt{(p mytype)}}_{❷}\ \underbrace{\texttt{funcname(}}_{❸}\underbrace{\texttt{q int)}}_{❹}\ \underbrace{\texttt{(r,s int)}}_{❺}\ \underbrace{\texttt{\{ return 0,0 \}}}_{❻}$$

❶ The keyword `func` is used to declare a function;

❷ A function can be defined to work on a specific type, a more common name for such a function is method. This part is called a *receiver* and it is optional;

❸ *funcname* is the name of your function;

❹ The variable `q` of type `int` is the parameter;

❺ The variables `r` and `s` are the named return parameters for this function. Note that functions in Go can have multiple return values. See section "Multiple return values" for more information. If you want the return parameters not to be named you only give the types: `(int,int)`. If you have only one value to return you may omit the parentheses. If your function is a subroutine and does not have anything to return you may omit this entirely;

❻ This is the function's body, note that `return` is a statement so the braces around the parameters are optional.

Here are a two examples, the first is a function without a return value, the second is a simple function that returns its input.

```
func subroutine(in int) {
    return
}
```

```
func identity(in int) int {
    return in
}
```

Go disallows nested functions. You can however work around this by using anonymous functions (see section ”Functions as values” on page 24 in this chapter).

Scope

Variables declared outside any functions are global in Go, those defined in functions are local to those functions. If names overlap — a local variable is declared with the same name as a global one — the local variable hides the global one when the current function is executed.

Listing 3.2. Local scope

```
package main

var a = 6

func main() {
    p()
    q()
    p()
}

func p() {
    println(a)
}

func q() {
    a := 5          ← Definition
    println(a)
}
```

Listing 3.3. Global scope

```
package main

var a = 6

func main() {
    p()
    q()
    p()
}

func p() {
    println(a)
}

func q() {
    a = 5    ← Assignment
    println(a)
}
```

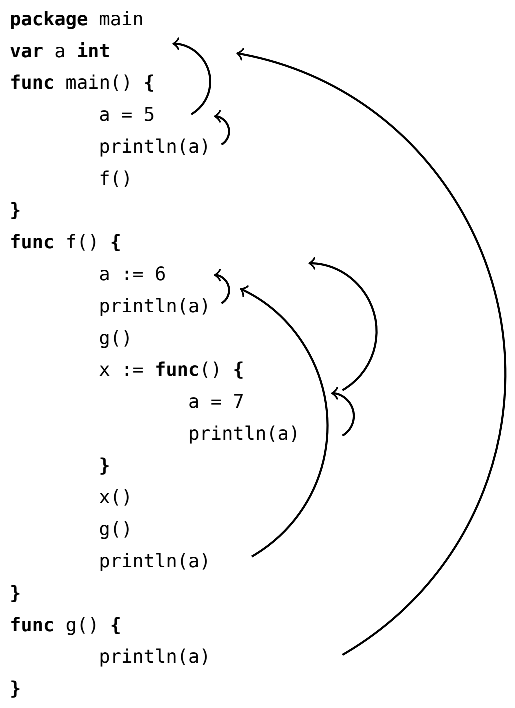
In listing 3.2 we introduce a local variable `a` in the function `q()`. This local `a` is only visible in `q()`. That is why the code will print: `656`. In listing 3.3 no new variables are introduced, there is only a global `a`. Assigning a new value to it is globally visible. This code will print: `655`

In the following example we call `g()` from `f()`:

Listing 3.4. Scope when calling functions from functions

```
package main

var a int

func main() {
        a = 5
        println(a)
        f()
}

func f() {
        a := 6
        println(a)
        g()
}

func g() {
        println(a)
}
```

The printout will be: `565`. A *local* variable is only valid when we are executing the function in which it is defined. Finally, one can create a ”function literal” in which you essentially define a function inside another function, i.e. a nested function. The following figure should clarify why it prints: `565757`. *Hint*: Just follow the arrows.


Listing 3.5. Scope and function literals

```
package main
var a int
func main() {
        a = 5
        println(a)
        f()
}
func f() {
        a := 6
        println(a)
        g()
        x := func() {
                a = 7
                println(a)
        }
        x()
        g()
        println(a)
}
func g() {
        println(a)
}
```

Multiple return values

One of Go’s unusual features is that functions and methods can return multiple values (Python can do this too). This can be used to improve on a couple of clumsy idioms in C programs: in-band error returns (such as -1 for `EOF`) and modifying an argument.

In C, a write error is signaled by a negative count with the error code secreted away in a volatile location. In Go, `Write` returns a count and an error: ”Yes, you wrote some bytes but not all of them because you filled the device”. The signature of `*File.Write` in package *os* is:

```
func (file *File) Write(b []byte) (n int, err Error)
```

and as the documentation says, it returns the number of bytes written and a non-`nil` `Error` when `n != len(b)`. This is a common style in Go.

A similar approach obviates the need to pass a pointer to a return value to simulate a reference parameter. Here’s a simple-minded function to grab a number from a position in a byte array, returning the number and the next position.

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i])-’0’
    }
    return x, i
}
```

You could use it to scan the numbers in an input array a like this:

```
    for i := 0; i < len(a); {
        x, i = nextInt(a, i)
        fmt.Printf(”%d”, x)
    }
```

Without having tuples as a native type, multiple return values is the next best thing to have. You can return precisely what you want without overloading the domain space with special values to signal errors.

Named result parameters

The return or result ”parameters” of a Go function can be given names and used as regular variables, just like the incoming parameters. When named, they are initialized to the zero values for their types when the function begins; if the function executes a **`return`** statement with no arguments, the current values of the result parameters are used as the returned values. Using this features enables you (again) to do more with less code[1].

The names are not mandatory but they can make code shorter and clearer: *they are documentation*. If we name the results of `nextInt` it becomes obvious which returned **`int`** is which.

```
func nextInt(b []byte, pos int) (value, nextPos int) {
```

Because named results are initialized and tied to an unadorned **`return`**, they can simplify as well as clarify. Here’s a version of `io.ReadFull` that uses them well:

```
func ReadFull(r Reader, buf []byte) (n int, err os.Error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
```

[1]This is sort of a motto of Go; ”Do more, with less code”.

```
            n += nr
            buf = buf[nr:len(buf)]
        }
        return
    }
```

Some text in this chapter comes from [10].

In the following example we declare a simple function which calculates the factorial value of a value x.

```
func Factorial(x int) int {      ← func Factorial(x int) (int) is also OK
   if x == 0 {
      return 1;
   } else {
      return x * Factorial(x - 1);
   }
}
```

So you could also write factorial as:

```
func Factorial(x int) (result int) {
  if x == 0 {
    result = 1;
  } else {
    result = x * Factorial(x - 1);
  }
  return;
}
```

When we use named result values, the code is shorter and easier to read. You can also write a function with multiple return values:

```
func fib(n) (val int, pos int) {
   if n == 0 {
      val = 1;
      pos = 0;
   } else if n == 1 {
      val = 1;
      pos = 1;
   } else {
      v1, _ := fib(n-1);
      v2,_ := fib(n-2);
      val = v1 + v2;
      pos = n;
   }
   return;
}
```

Deferred code

Suppose you have a function in which you open a file and perform various writes and reads on it. In such a function there are often spots where you want to return early. If you do that, you will need to close the file descriptor you are working on. This often leads to the following code:

Listing 3.6. Without `defer`

```
function ReadWrite() bool {
    file.Open("file")
    // Do you thing
    if failureX {
        file.Close("file")
        return false
    }

    if failureY {
        file.Close("file")
        return false
    }
    file.Close("file")
    return true
}
```

Here a lot of code is repeated, but with a good reason. To overcome this — and do more in less code — Go has the `defer` statement. After `defer` you specify a function which is called just *before* a return from the function is executed.

The code above could be rewritten as follows. This makes the function more readable, shorter and puts the `Close` right next to the `Open`.

Listing 3.7. With `defer`

```
function ReadWrite() bool {
    file.Open("file")
    defer file.Close("file")     ← file.Close() is the function
    // Do you thing
    if failureX {
        return false             ← Close() is now done automatically
    }
    if failureY {
        return false             ← And here too
    }
    return true
}
```

You can put multiple functions on the "deferred list", like this example from [2]:

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

Deferred functions are executed in LIFO order, so the above code prints: `4 3 2 1 0`.

With `defer` you can even change return values, provided that you are using named result parameters and a function literal, i.e:

Listing 3.8. Function literal

```
defer func() {
        ...
}()                 ← () is needed here
```

In that (unnamed) function you can access any named return parameter:

Listing 3.9. Access return values within `defer`

```
func f() (ret int) {        ← ret is initialized with zero
        defer func() {
                ret++       ← Increment ret with 1
        }()
        return 0            ← 1 not 0 will be returned
}
```

Variadic parameters

Functions that take variadic parameters are functions that have a variable number of parameters. To do this, you first need to declare your function to take variadic arguments:

Listing 3.10. Variadac parameters

```
func myfunc(arg ... int) {}
```

The `arg ... int` instructs Go to see this as a function that takes a variable number of arguments. Note that these arguments all have the type **`int`**. Inside your function's body the variable `arg` is a slice of **`int`**s:

```
for _, n := range arg {
    fmt.Printf("And the number is: %d\n", n)
}
```

If you don't specify the type of the variadic argument it defaults to the empty interface `interface{}` (see "Interfaces" in chapter 5).

Functions as values

As with almost everything in Go, functions are also *just* values. They can be assigned to variables as follows:

Listing 3.11. Anonymous function

```
package main

func main() {
        a := func() {     // Define a nameless function and assign to a
          println("Hello")
        }                 // No () here
        a()               // Call the function
}
```

If we use `fmt.Printf("%T\n", a)` to print the type of `a`, it prints `func()`.

Functions–as–values may also be used in other places, like in maps. Here we convert from integers to functions.

Listing 3.12. Function as values in maps

```
var xs = map[int]func() int{
    1: func() int { return 10 },
    2: func() int { return 20 },
3: func() int { return 30 },
...
```

Or you can write a function that takes a function as its parameter, for example a `Map` function that works on **`int`** slices. This is left as an exercise for the reader, see exercise Q11 on page 26.

Panic and recovering

todo

Exercises

Q5. (3) Integer ordering

1. Write a function that returns it parameters in the right order:
   `f(7,2)` → 2,7
   `f(2,7)` → 2,7

Q6. (4) Scope

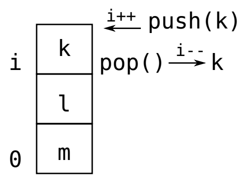
1. What is wrong with the following program?

```
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {
                fmt.Printf("%v\n", i)
        }
        fmt.Printf("%v\n", i)
}
```

Q7. (5) Stack

1. Create a simple stack which can hold a fixed amount of **`int`**s. It does not have to grow beyond this limit. Define both a `push` – put something on the stack – and a `pop` – retrieve something fro the stack – function. The stack should be a LIFO (last in, first out) stack.

Figure 3.1. A simple LIFO stack



2. Bonus. Write a `String` method which converts the stack to a string representation. This way you can print the stack using: `fmt.Printf("My stack %v\n", stack)`
   The stack in the figure could be represented as: `[0:m] [1:l] [2:k]`

Q8. (2) Stack as package

1. Create a proper package for your stack implementation, `Push`, `Pop` and the **`Stack`** type need to be exported.
2. Which official Go package could also be used for a (FIFO) stack?

Q9. (5) Var args

1. Write a function that takes a variable numbers of **`int`**s and prints each integer on a separate line

todo BETTER

Q10. (5) Fibonaci

1. The Fibonaci sequence starts as follows: $1, 1, 2, 3, 5, 8, 13, \ldots$ Or in mathematical terms: $x_1 = 1; x_2 = 1; x_n = x_{n-1} + x_{n-2} \quad \forall n > 2$.
   Write a function that takes an **`int`** value and gives that many terms of the Fibonaci sequence.

Q11. (4) Map function A `map()`-function is a function that takes a function and a list. The function is applied to each member in the list and a new list containing these calculated values is returned. Thus:

$$map(f(), (a_1, a_2, \ldots, a_{n-1}, a_n)) = (f(a_1), f(a_2), \ldots, f(a_{n-1}), f(a_n))$$

1. Write a simple `map()`-function in Go. It is sufficient for this function only to work for **`int`**s.
2. Expand your code to also work on a list of **`strings`**.

Answers

A5. (3) Integer ordering

1.

A6. (4) Scope

1. The program does not even compile, because `i` on line 9 is not defined: `i` is only defined within the **for**-loop. To fix this the function `main()` should become:

   ```
   func main() {
           var i int
           for i = 0; i < 10; i++ {
                   fmt.Printf(”%v\n”, i)
           }
           fmt.Printf(”%v\n”, i)
   }
   ```

   Now `i` is defined outside the **for**-loop and still visible after wards. This code will print the numbers 0 through 10.

A7. (5) Stack

1. First we define a new type that represents a stack; we need an array (to hold the keys) and an index, which points to the last element. Our small stack can only hold 10 elements.

   ```
   type stack struct {     ← stack is not exported
       i    int
       data [10]int
   }
   ```

   Next we need the `push` and `pop` functions to actually use the thing. *First we show the* wrong *solution!* In Go data passed to functions is *passed-by-value* meaning a copy is created and given to the function. The first stab for the function `push` could be:

   ```
   func (s stack) push(k int) {     ← Works on copy of argument
           if s.i+1 > 9 {
                   return
           }
           s.data[s.i] = k
           s.i++
   }
   ```

   The function works on the `s` which is of the type **`stack`**. To use this we just call `s.push(50)`, to push the integer 50 on the stack. But the push function gets a copy of `s`, so it is *not* working the *real* thing. Nothing gets pushed to our stack this way, for example the following code:

   ```
   var s stack     ← make s a simple stack variable
   s.push(25)
   fmt.Printf(”stack %v\n”, s);
   s.push(14)
   ```

```
fmt.Printf(”stack %v\n”, s);
```

prints:

```
stack [0:0]
stack [0:0]
```

To solve this we need to give the function `push` a pointer to the stack. This means we need to change `push` from
`func (s stack)push(k int)` $\longrightarrow$ `func (s *stack)push(k int)`
We should now use `new()` (see ”Allocation with new” in chapter 5) to create a *pointer* to a newly allocated **`stack`**, so line 1 from the example above needs to be
`s := new(stack)`
And our two functions become:

Listing 3.13. The push and pop functions

```
func (s *stack) push(k int) {
        if s.i+1 > 9 {
                return
        }
        s.data[s.i] = k
        s.i++
}

func (s *stack) pop() (ret int) {
        if s.i-1 < 0 {
                return 0
        }
        ret = s.data[s.i]
        s.i--
        return ret
}
```

Which we then use as follows

Listing 3.14. Stack usage

```
func main() {
        s := new(stack) // returns pointer!
        s.push(25)
        s.push(14)
        fmt.Printf(”stack %v\n”, s)
}
```

2. While this was a bonus question, having the ability to print the stack was very valuable when writing the code for this exercise. According to the Go documentation `fmt.Printf(”%v”)` can print any value (%v) that satisfies the `Stringer` interface. For this to work we only need to define a `String()` function for our type:

Listing 3.15. stack.String()

```
func (s *stack) String() string {
        var str string
        for i := 0; i < s.i; i++ {
                str = str + "[" +
                        strconv.Itoa(i) + ":" + strconv.Itoa(s.data[i
                             ]) + "]"
        }
        return str
}
```

A8. (2) Stack as package

1. There are a few details that should be changed to make a proper package for our stack. First, the exported function should begin with a capital letter and so should **Stack**. So the full package (including the `String()` function becomes

Listing 3.16. Stack in a package

```
package stack

import (
        "strconv"
)

type Stack struct {
        i    int
        data [10]int
}

func (s *Stack) Push(k int) {
        if s.i+1 > 9 {
                return
        }
        s.data[s.i] = k
        s.i++
}

func (s *Stack) Pop() (ret int) {
        if s.i-1 < 0 {
                return 0
        }
        ret = s.data[s.i]
        s.i--
        return ret
}

func (s *Stack) String() string {
        var str string
        for i := 0; i < s.i; i++ {
                str = str + "[" + strconv.Itoa(i) + ":"
```

```
                        + strconv.Itoa(s.data[i]) + "]"
        }
        return str
}
```

A9. (5) Var args

1. For this we need the ...-syntax to signal we define a function that takes an arbitrary number of arguments.

Listing 3.17. A function with variable number of arguments

```
package main

import "fmt"

func main() {
        printthem(1, 4, 5, 7, 4)
        printthem(1, 2, 4)
}

func printthem(numbers ... int) {    ← numbers is now a slice of ints
        for _, d := range numbers {
                fmt.Printf("%d\n", d)
        }
}
```

A10. (5) Fibonaci

1. The following program calculates the Fibonaci numbers.

Listing 3.18. A Fibonaci function in Go

```
package main

import "fmt"

func fibonacci(value int) []int {
        x := make([]int, value) ❶
        x[0], x[1] = 1, 1 ❷
        for n := 2; n < value; n++ {
                x[n] = x[n-1] + x[n-2] ❸
        }
        return x ❹
}

func main() {
        for _, term := range fibonacci(10) { ❺
                fmt.Printf("%v ", term)
        }
}
```

❶ We create an **array** to hold the integers up to the value given in the function call;

❷ Starting point of the Fibonicai calculation;

❸ $x_n = x_{n-1} + x_{n-2}$;

❹ Return the *entire* array;

❺ Using the **range** keyword we ”walk” the numbers returned by the fibonacci funcion. Here up to 10. And we print them.

A11. (4) Map function

Listing 3.19. A `Map` function

1.
```go
func Map(f func(int) int, l []int) []int {
        j := make([]int, len(l))
        for k, v := range l {
                j[k] = f(v)
        }
        return j
}

func main() {
        m := []int{1, 3, 4}
        f := func(i int) int {
                return i * i
        }
        fmt.Printf(”%\v”, (Map(f, m)))
}
```

2. Answer to question but now with strings

4 Packages

^

Answer to whether there is a bitwise negation operator.
Ken Thompson

Packages are a collection of functions and data, they are like the Perl packages[7]. You declare a package with the **package** keyword. Note that (at least one of) the filename(s) should match the package name. Go packages may consist out of multiple files, but they share the **package** <name> line. Lets define our package *even* in the file even.go.

Listing 4.1. A small package

```
package even                ← Start our own namespace

func Even(i int) bool {     ← Exported
        if i % 2 == 0 {
                return true
        }
        return false
}

func odd(i int) bool {      ← Private
        if i % 2 != 0 {
                return false
        }
        return true
}
```

Names that start with a capital letter are exported and may be used outside your package, more on that later. We can now use the package as follows in our own program myeven.go:

Listing 4.2. Use of the even package

```
package main

import (                    ← Import the following packages
        "./even"            ← The local package even
        "fmt"               ← The official fmt package
)

func main() {
        i := 5
        fmt.Printf("Is %d even? %v\n", i, even.Even(i))
}
```

Now we just need to compile and link, first the package, then myeven.go and then link it:

```
% 6g even.go        # package
% 6g myeven.go       # our program
% 6l -o myeven myeven.8
```

And test it:

```
% ./myeven
Is 5 even? false
```

In Go, a function from a package is exported (visible outside the package, i.e. public) when the first letter of the function name is a capital, hence the function name `Even`. If we change our `myeven.go` on line 7 to using to unexported function `even.odd`:

```
fmt.Printf("Is %d even? %v\n", i, even.odd(i))
```

We get an error when compiling, because we are trying to use a *private* function:

```
myeven.go:7: cannot refer to unexported name even.odd
myeven.go:7: undefined: even.odd
```

To summarize:

- Public functions have a name *starting* with a capital letter;
- Private function have a name *starting* with a lowercase letter.

Building a package

The create a package that other people can use (by just using **import** "even") we first need to create a directory to put the package files in.

```
% mkdir even
% cp even.go even
```

Next we can use the following `Makefile`, which is adapted for our *even* package.

Listing 4.3. `Makefile` for a package

```
# Copyright 2009 The Go Authors. All rights reserved.            1
# Use of this source code is governed by a BSD-style             2
# license that can be found in the LICENSE file.                 3

include $(GOROOT)/src/Make.$(GOARCH)                             5

TARG=even                                                        7
GOFILES=\                                                        8
        even.go\                                                 9

include $(GOROOT)/src/Make.pkg                                   11
```

Note that on line 7 we define our *even* package and on the lines 8 and 9 we enter the files which make up the package. Also note that this is the *same* `Makefile` setup as we used in section "**??**" in chapter 2.

If we now issue `make`, a file named "`_go_.6`", a directory named "`_obj/`" and a file inside "`_obj/`" called "`even.a`" is created. The file `even.a` is a static library which holds the compiled Go code. With `make install` the package (well, only the `even.a`) is installed in the *official* package directory:

```
% make install
cp _obj/even.a $GOROOT/pkg/linux_amd64/even.a
```

After installing we can change our **`import`** from `"./even"` to `"even"`.

But what if you do not want to install packages in the official Go directory, or maybe you do not have the permission to do so? When using `6/8g` you can use the *I*-flag to specify an alternate package directory. With this flag you can leave your import statement as-is (**`import`** `"even"`) and continue development in your own directory. So the following commands should build (and link) our `myeven` program with our package.

```
% 6g -I even/_obj myeven.go # building
% 6l -L even/_obj myeven.6 # linking
```

Identifiers

Names are as important in Go as in any other language. In some cases they even have semantic effect: for instance, the visibility of a name outside a package is determined by whether its first character is upper case. It's therefore worth spending a little time talking about naming conventions in Go programs.

The convention we adopted was to leave well-known legacy not-quite-words alone rather than try to figure out where the capital letters go. `Atoi`, `Getwd`, `Chmod`. Camelcasing works best when you have whole words to work with: `ReadFile, NewWriter, MakeSlice`

Package names

When a package is imported (with **`import`**), the package name becomes an accessor for the contents. After

```
import "bytes"
```

the importing package can talk about `bytes.Buffer`. It's helpful if everyone using the package can use the same name to refer to its contents, which implies that the package name should be good: short, concise, evocative. By convention, packages are given lower case, single-word names; there should be no need for underscores or mixedCaps. Err on the side of brevity, since everyone using your package will be typing that name. And don't worry about collisions a priori. The package name is only the default name for imports. With the above import you can use `bytes.Buffer`. With

```
import bar "bytes"
```

it becomes `bar.Buffer`. So it does need not be unique across all source code, and in the rare case of a collision the importing package can choose a different name to use locally. In any case, confusion is rare because the file name in the import determines just which package is being used.

Another convention is that the package name is the base name of its source directory; the package in *src/pkg/container/vector* is imported as `container/vector` but has name *vector*, not *container_vector* and not *containerVector*.

The importer of a package will use the name to refer to its contents, so exported names in the package can use that fact to avoid stutter. For instance, the buffered reader type in the *bufio* package is called Reader, not BufReader, because users see it as bufio.Reader, which is a clear, concise name. Moreover, because imported entities are always addressed with their package name, bufio.Reader does not conflict with io.Reader. Similarly, the function to make new instances of ring.Ring—which is the definition of a constructor in Go—would normally be called NewRing, but since **Ring** is the only type exported by the package, and since the package is called *ring*, it's called just New. Clients of the package see that as ring.New. Use the package structure to help you choose good names.

Another short example is once.Do; once.Do(setup) reads well and would not be improved by writing once.DoOrWaitUntilDone(setup). Long names don't automatically make things more readable. If the name represents something intricate or subtle, it's usually better to write a helpful doc comment than to attempt to put all the information into the name.

Finally, the convention in Go is to use MixedCaps or mixedCaps rather than underscores to write multiword names.

Initialization

Every source file in a package can define an init() function. This function is called after the variables in the package have gotten their value. The init() function can be used to setup state before the execution begins.

todo
Variable section

Documenting packages

From [2] Every package should have a *package comment*, a block comment preceding the **package** clause. For multi-file packages, the package comment only needs to be present in one file, and any one will do. The package comment should introduce the package and provide information relevant to the package as a whole. It will appear first on the godoc page and should set up the detailed documentation that follows. An example is:

```
/*
    The regexp package implements a simple library for
    regular expressions.

    The syntax of the regular expressions accepted is:

    regexp:
        concatenation  '|' concatenation
*/
package regexp
```

Each defined (and exported) function should has a small line of text documenting the behavior of the function, from the *fmt* package:

```
// Printf formats according to a format specifier and writes to standard output.
func Printf(format string, a ...interface) (n int, errno os.Error) {
```

Testing packages

In Go it is customary to write (unit) tests for your package. Writing tests involves the *testing* package and the program gotest. Both have excellent documentation. When you include tests with your package keep in mind that has to be build using the standard `Makefile` (see section "Building a package").

The testing itself is carried out with `gotest`. The `gotest` program run all the test functions. Without any defined tests for our *even* package a `make test`, yields:

```
% make test
no test files found (*_test.go)
make: *** [test] Error 2
```

Let us fix this by defining a test in a test file. Test files reside in the package directory and are named `*_test.go`. Those test files are just like other Go program, but `gotest` will only execute the test functions. Each test function has the same signature and its name should start with `Test`:

```
func TestXxx(t *testing.T)     ← Test<Capital>restOftheName
```

When writing test you will need to tell `gotest` that a test has failed or was successful. A successful test function just returns. When test fails you can signal this with the following functions [4]. These are the most important ones (see `godoc testing` for more):

```
func (t *T) Fail()
```

`Fail` marks the Test function as having failed but continues execution.

```
func (t *T) FailNow()
```

`FailNow` marks the Test function as having failed and stops its execution. Execution will continue at the next Test. So any other test in *this* file are skipped too.

```
func (t *T) Log(args ...interface{})
```

`Log` formats its arguments using default formatting, analogous to `Print()`, and records the text in the error log.

```
func (t *T) Fatal(args ...interface{})
```

`Fatal` is equivalent to Log() followed by FailNow().

Putting all this together we can write our very own test file. First we pick a name: `even_test.go`. Then we add the following contents:

Listing 4.4. Test file for even package

```
package even                                  1

import "testing"                              3

func TestEven(t *testing.T) {                 5
        if true != Even(2) {                  6
                t.Log("2 should be even!")    7
                t.Fail()                      8
        }                                     9
}                                             10
```

Note that we use `package even` on line 1, the test fall in the same namespace as the package we are testing. This not only convenient, but also allows tests of unexported function and structures. We then import the *testing* package and on line 5 we define the only test function in this file. The displayed Go code should not hold any surprises: we check if the `Even` function works OK. Now, the moment we've been waiting for, executing the test:

```
% make test
6g -o _gotest_.6 even.go  even_test.go
rm -f _test/even.a
gopack grc _test/even.a _gotest_.6
PASS
```

Our test ran and reported PASS. Success! To show how a failed test look we redefine our test function:

```
// Entering the twilight zone
func TestEven(t *testing.T) {
        if false != Even(2) {
                t.Log("2 should be odd!")
                t.Fail()
        }
}
```

We now get:

```
--- FAIL: even.TestEven
      2 should be odd!
FAIL
make: *** [test] Error 1
```

And you can act accordingly (by fixing the test for instance).

> Writing new packages should go hand in hand with writing (some) documentation and test functions. It will make your code better and it shows that your really put in the effort.

Advanced importing

todo
import from the web?

Useful packages

The standard Go repository includes a huge number of packages and it is even possible to install more along side your current Go installation. We cannot comment on each and everyone of the package, but the following a worth a mention:

fmt
 …

io
 …

bufio
 …

sort
 …

strconv
 …

os
: …

flag
: …

json
: …

template
: …

http
: …

unsafe
: …

reflect
: …

exec
: …

Exercises

Q12. (7) Calculator

1. Create a reverse polish calculator. Use your stack package.
2. Bonus. Rewrite your calculator to use the the package you found for question 2.

Answers

A12. (7) Calculator

1. This is one answer

Listing 4.5. A (rpn) calculator

```
package main

import (
        ”bufio”
        ”os”
        ”strconv”
        ”fmt”
)

var reader *bufio.Reader = bufio.NewReader(os.Stdin)
var st = new(Stack)

type Stack struct {
        i    int
        data [10]int
}

func (s *Stack) push(k int) {
        if s.i+1 > 9 {
                return
        }
        s.data[s.i] = k
        s.i++
}

func (s *Stack) pop() (ret int) {
        s.i--
        if s.i < 0 {
                s.i = 0
                return 0
        }
        ret = s.data[s.i]
        return ret
}

func (s *Stack) String() string {
        var str string
        for i := 0; i < s.i; i++ {
                str = str + ”[” +
                        strconv.Itoa(i) + ”:” + strconv.Itoa(s.data[i
                            ]) + ”]”
        }
        return str
}

func main() {
        for {
                s, err := reader.ReadString('\n')
                var token string
                if err != nil {
                        return
                }
                for _, c := range s {
                        switch {
                        case c >= '0' && c <= '9':
                                token = token + string(c)
                        case c == ' ':
                                r, _ := strconv.Atoi(token)
                                st.push(r)
                                token = ""
                        case c == '+':
                                fmt.Printf("%d\n", st.pop()+st.pop())
                        case c == '*':
                                fmt.Printf("%d\n", st.pop()*st.pop())
                        case c == '-':
                                p := st.pop()
                                q := st.pop()
                                fmt.Printf("%d\n", q-p)
                        case c == 'q':
                                return
                        default:
                                //error
                        }
                }
        }
}
```

2. The *container/vector* package would be a could candidate. It even comes with `Push` and `Pop` functions *predefined*. The changes needed to our program are *minimal* to say the least, the following unified diff shows the differences:

```
--- calc.go     2010-05-16 10:19:13.886855818 +0200
+++ calcvec.go  2010-05-16 10:13:35.000000000 +0200
@@ -5,11 +5,11 @@
        ”os”
        ”strconv”
        ”fmt”
-       ”./stack”
+       ”container/vector”
 )

 var reader *bufio.Reader = bufio.NewReader(os.Stdin)
-var st = new(Stack)
+var st = new(vector.IntVector)

 func main() {
        for {
```

Only two lines need to be changed. *Nice*.

5 Beyond the basics

`fmt.Printf("%p", i)`

Printing the address of a pointer in Go.

You may have wished otherwise, but Go does have pointers. There is however no pointer arithmetic, so they act more like references than pointers that you may know from C. Pointers as they are still called in Go are useful. Remember that when you call a function in Go, the variables are *pass-by-value*. So, for efficiency and the possibility to modify a passed value *in* functions we have pointers.

Just like in C you declare a pointer by prefixing the type with an '*': **`var`** `p *`**`int`**. Now `p` is a pointer to an integer value. All newly declared variables are assigned their zero value and pointer are no difference. A newly declared, or just a pointer that points to nothing has a nil-value. In other languages this is often called a NULL pointer in Go it is just `nil`. To make a pointer point to something you can use the address-of operator (&), which we do on line 5:

Listing 5.1. Make use of a pointer

```
var p *int                                                            1
fmt.Printf("%v", p)     ← Prints nil                                  2

var i int               ← Declare integer variable i                  4
p = &i                  ← Make p point to i                           5

fmt.Printf("%v", p)     ← Prints something like 0x7ff96b81c000a       7
```

More general: `*X` is a pointer to an **`X`**; `[3]X` is an array of three **`X`**'s. The types are therefore really easy to read just read out the names of the type modifiers: **`[]`** declares something called an array slice; '*' declares a pointer; **`[size]`** declares an array. So **`[]*[3]*int`** is an array slice of pointers to arrays of three pointers to **`int`**s.

todo
Make nice info graphic

Dereferencing a pointer is done by prefixing the pointer variable with '*':

Listing 5.2. Dereferencing a pointer

```
p = &i
*p = 8                    ← Change the value of i
fmt.Printf("%v\n", *p)    ← Prints 8
fmt.Printf("%v\n", i)     ← Idem
```

As said, there is no pointer arithmetic, so if you write: `*p++`, it is interpreted as `(*p)++`: first deference and then increment the value.

Allocation

Go has garbage collection, meaning that you don't have to worry about memory allocation and deallocation. Of course almost every language since 1980 has this, but it is nice to see garbage collection in a C-like language which in not C++. The following sections show how to handle allocation in Go. There is a somewhat an artificial distinction between **new** and **make**.

Allocation with new

Go has two allocation primitives, **new** and **make**. They do different things and apply to different types, which can be confusing, but the rules are simple. **new** is a built-in function essentially the same as its namesakes in other languages: new(T) allocates zeroed storage for a new item of type **T** and returns its address, a value of type ***T**. In Go terminology, it returns a pointer to a newly allocated zero value of type **T**.

This means a user of the data structure can create one with **new** and get right to work. For example, the documentation for bytes.Buffer states that ”the zero value for Buffer is an empty buffer ready to use.” Similarly, sync.Mutex does not have an explicit constructor or Init method. Instead, the zero value for a sync.Mutex is defined to be an unlocked mutex.

The zero-value-is-useful property works transitively. Consider this type declaration.

```
type SyncedBuffer struct {
    lock    sync.Mutex
    buffer  bytes.Buffer
}
```

Values of type **SyncedBuffer** are also ready to use immediately upon allocation or just declaration. In this snippet, both p and v will work correctly without further arrangement.

```
p := new(SyncedBuffer)  // type *SyncedBuffer
var v SyncedBuffer      // type SyncedBuffer
```

Allocation with make

Back to allocation. The built-in function make(T, args) serves a purpose different from new(T). It creates slices, maps, and channels only, and it returns an initialized (not zero) value of type **T**, not ***T**. The reason for the distinction is that these three types are, under the covers, references to data structures that must be initialized before use. A slice, for example, is a three-item descriptor containing a pointer to the data (inside an array), the length, and the capacity; until those items are initialized, the slice is **nil**. For slices, maps, and channels, **make** initializes the internal data structure and prepares the value for use. For instance, **make**([]**int**, 10, 100) allocates an array of 100 **int**s and then creates a slice structure with length 10 and a capacity of 100 pointing at the first 10 elements of the array. In contrast, **new**([]**int**) returns a pointer to a newly allocated, zeroed slice structure, that is, a pointer to a **nil** slice value.

These examples illustrate the difference between **new()** and **make()**.

```
var p *[]int = new([]int)       // allocates slice structure; *p == nil
                                // rarely useful
var v  []int = make([]int, 100) // v refers to a new array of 100 ints

// Unnecessarily complex:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Idiomatic:
```

```
v := make([]int, 100)
```

Remember that **make()** applies only to maps, slices and channels and does not return a pointer. To obtain an explicit pointer allocate with **new()**.

Constructors and composite literals

Sometimes the zero value isn't good enough and an initializing constructor is necessary, as in this example taken from the package *os*.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

There's a lot of boiler plate in there. We can simplify it using a composite literal, which is an expression that creates a new instance each time it is evaluated.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}
    return &f
}
```

Note that it's perfectly OK to return the address of a local variable; the storage associated with the variable survives after the function returns. In fact, taking the address of a composite literal allocates a fresh instance each time it is evaluated, so we can combine these last two lines.

```
return &File{fd, name, nil, 0}
```

The fields of a composite literal are laid out in order and must all be present. However, by labeling the elements explicitly as field:value pairs, the initializers can appear in any order, with the missing ones left as their respective zero values. Thus we could say

```
return &File{fd: fd, name: name}
```

As a limiting case, if a composite literal contains no fields at all, it creates a zero value for the type. The expressions `new(File)` and `&File{}` are equivalent.

Composite literals can also be created for arrays, slices, and maps, with the field labels being indices or map keys as appropriate. In these examples, the initializations work regardless of the values of `Enone`, `Eio`, and `Einval`, as long as they are distinct.

```
ar := [...]string   {Enone: "no error", Eio: "Eio", Einval: "invalid
     argument"}
sl := []string      {Enone: "no error", Eio: "Eio", Einval: "invalid
     argument"}
```

```
ma := map[int]string{Enone: ”no error”, Eio: ”Eio”, Einval: ”invalid
     argument”}
```

Conversions

Sometimes you want to convert a type to another type. In C this is known as casting a value to another type. This is also possible in Go, but there are some rules. For starters, converting from one value to another is done by functions and not all conversion are allowed.

- From a **string** to a slice of **byte**s.

  ```
  mystring := ”hello this is string”
  byteslice := []byte(mystring)
  ```

- From a slice of **byte**s to a **string**.

  ```
  b := []byte{’h’,’e’,’l’,’l’,’o’}     ← Composite literal
  s := string(b)
  ```

For numeric values the following conversion are defined:

- Convert to a integer with a specific (bit) length: **uint8**(**int**);
- From floating point to an integer value: **int**(**float**). This discards the fraction part from the floating point value;
- The other way around: **float**(**int**);

Defining your own

Of course Go allows you to define new types, it does this in (almost) the same way as in C, with the **struct** keyword.

An empty struct is created with **var** empty **struct {}**. A more real-life example would be when we want record somebody’s name and age in a single structure:

Listing 5.3. Structures

```
package main

import ”fmt”

func main() {
        var a struct {
                name string
                age  int
        }

        a.name = ”Pete”
        a.age = 42

        fmt.Printf(”%v\n”, a)
}
```

Apropos, the output of `fmt.Printf("\%v\n", a)` is

```
Pete, 42
```

That is nice! Go knows how to print your structure. If you only want to print one, or a few, fields of the structure you'll need to use `.<field name>`. For example to only print the name:

```
fmt.Printf("%s", a.name)     ← %s formats a string
```

Interfaces

The following text is from [20]. Written by Ian Lance Taylor — one of the authors of Go.

In Go, the word interface is overloaded to mean several different things. Every type has an interface, which is the set of methods defined for that type. This bit of code defines a struct type **S** with one field, and defines two methods for **S**.

Listing 5.4. Defining a struct and methods on it

```
type S struct { i int }
func (p *S) Get() int { return p.i }
func (p *S) Put(v int) { p.i = v }
```

You can also define an interface type, which is simply a set of methods. This defines an interface **I** with two methods:

```
type I interface {
  Get() int;
  Put(int);
}
```

S is a valid implementation for **I**, because it defines the two methods which **I** requires. Note that this is true even though there is no explicit declaration that **S** implements **I**. A Go program can use this fact via yet another meaning of interface, which is an interface value:

```
func f(p I) { fmt.Println(p.Get()); p.Put(1) }
```

Here the variable `p` holds a value of interface type. Because **S** implements **I**, we can call `f` passing in a pointer to a value of type **S**:

```
var s S; f(&s)
```

The reason we need to take the address of **s**, rather than a value of type **S**, is because we defined the methods on **s** to operate on pointers, see the code above in listing 5.4. This is not a requirement — we could have defined the methods to take values — but then the `Put` method would not work as expected.

The fact that you do not need to declare whether or not a type implements an interface means that Go implements a form of duck typing[25]. This is not pure duck typing, because when possible the Go compiler will statically check whether the type implements the interface. However, Go does have a purely dynamic aspect, in that you can convert from one interface type to another. In the general case, that conversion is checked at runtime. If the conversion is invalid — if the type of the value stored in the existing interface value does not satisfy the interface to which it is being converted — the program will fail with a runtime error.

Interfaces in Go are similar to ideas in several other programming languages: pure abstract virtual base classes in C++, typeclasses in Haskell or duck typing in Python. However there is no other language which combines interface values, static type checking, dynamic runtime conversion, and no requirement for explicitly declaring that a type satisfies an interface. The result in Go is powerful, flexible, efficient, and easy to write.

Empty interface

For example, since every type satisfies the empty interface: `interface {}`. We can create a generic function which has an empty interface as its argument:

Listing 5.5. A function with a empty interface argument

```
func g(any interface{}) int { return any.(I).Get() }
```

The `return any.(I).Get()` is the tricky bit in this function. The value `any` has type **`interface`**, meaning no guarantee of any methods at all: it could contain any type. The `.(I)` is a type switch which converts `any` to an interface of type **`I`**. If we have that type we can invoke the `Get()` function. So if we create a new variable of the type **`*S`**, we can just call `g()`, because **`*S`** also implements the empty interface.

```
s = new(S)
fmt.Println(g(s));
```

The call to `g` will work fine and will print 0. If we however invoke `g()` with a value that does not implement **`I`** we have a problem:

Listing 5.6. Failing to implement an interface

```
i := 5          // make i a "lousy" int
fmt.Println(g(i))
```

This compiles OK, but when we run this we get slammed with:

```
panic: interface conversion: int is not main.I: missing method Get
```

Which is completely true, the built-in type **`int`** does not have a `Get()` function.

Methods

You can also define functions (or better called methods from now on) on interfaces. In fact, any value, and type can have methods. You can make an integer type with its own methods. For example:

```
type Foo int;

func (self Foo) Emit() {
  fmt.Printf("\%v", self);
}

type Emitter interface {
  Emit();
}
```

Interface names

By convention, one-method interfaces are named by the method name plus the *-er* suffix: Read*er*, Writ*er*, Formatt*er* etc.

There are a number of such names and it's productive to honor them and the function names they capture. `Read`, `Write`, `Close`, `Flush`, `String` and so on have canonical signatures and meanings. To avoid confusion, don't give your method one of those names unless it has the same signature and meaning. Conversely, if your type implements a method with the same meaning as a method on a well-known type, give it the same name and signature; call your string-converter method `String` not `ToString`.

Text copied from [2].

Introspection

type switch

In a program, you can discover the dynamic type of an interface variable by using a **`switch`**. Such a type switch uses the syntax of a type assertion with the keyword type inside the parentheses. If the switch declares a variable in the expression, the variable will have the corresponding type in each clause.

Listing 5.7. Dynamically find out the type

```
package main
type PersonAge struct { ❶
        name string
        age  int
}

type PersonShoe struct { ❷
        name     string
        shoesize int
}

func main() {
        p1 := new(PersonAge)
        p2 := new(PersonShoe)
        WhichOne(p1)
        WhichOne(p2)
}

func WhichOne(x interface{}) { ❸
        switch t := x.(type) { ❹
        case *PersonAge:         ❺
                println("Age person")
        case *PersonShoe:
                println("Shoe person")
        }
}
```

❶ First we define two structures as a new type, `PersonAge`;

❷ And `PersonShoe`;

❸ This function must accept *both* types as valid input, so we use the empty Interface, which every type implements;

❹ The type switch: `(type)`;

❺ When allocated with `new` it's a pointer. So we check for **`*PersonAge`**. If `WhichOne()` was called with a non pointer value, we should check for **`PersonAge`**.

The following is another example of performing a type switch, but this time checking for more (built-in) types:

Listing 5.8. A more generic type switch

```
switch t := interfaceValue.(type) {     ← The type switch
case bool:
    fmt.Printf("boolean %t\n", t)
case int:
    fmt.Printf("integer %d\n", t)
case *bool:
    fmt.Printf("pointer to boolean %t\n", *t)
case *int:
    fmt.Printf("pointer to integer %d\n", *t)
default:
    fmt.Printf("unexpected type %T", t)  // %T prints type
}
```

Introspection and reflection

In the following example we want to look at the "tag" (here named "namestr") defined in the type definition of **`Person`**. To do this we need the *reflect* package (there is no other way in Go). Keep in mind that looking at a tag means going back the *type* definition. So we use the *reflect*package to figure out the type of the variable and *then* access the tag.

Listing 5.9. Introspection using reflection

```
type Person {
    name string "namestr"
    age  int
}

p1 := new(Person)       ← new returns a pointer to Person
ShowTag(p1)             ← ShowTag() is now called with this pointer

func ShowTag(i interface{}) {
 switch t := reflect.NewValue(i).(type) {     ← Type assert on reflect value
 case *reflect.PtrValue:              ← Hence the case for *reflect.PtrValue
```

$$\texttt{tag := }\underbrace{\texttt{t.Elem()}}_{❶}\texttt{.}\underbrace{\texttt{Type()}}_{❷}\texttt{.}\underbrace{\texttt{(*reflect.StructType)}}_{❸}\texttt{.}\underbrace{\texttt{Field(0)}}_{❹}\texttt{.}\underbrace{\texttt{Tag}}_{❺}$$

❶ We are dealing with a **`PtrValue`** and according to the documentation[1]:

[1] `godoc reflect`

> *func (v *PtrValue) Elem() Value*
> *Elem returns the value that v points to. If v is a nil pointer, Elem returns a nil Value.*

we can use `Elem()` to get the type the pointer points to. In this case **`*reflect.StructValue`**. We have also used **`reflect.NewValue(i)`** for the type assertion, so that we get back types in the **`*reflect`** namespace;

❷ On a `Value` we can use the function `Type()` which returns **`reflect.Type`**;

❸ Again according to the documentation, we have:

> *...which returns an object with interface type **`Type`**. That contains a pointer to a struct of type **`*StructType`**, **`*IntType`**, etc. representing the details of the underlying type. A type switch or type assertion can reveal which.*

So we can access your specific type as a member of this struct. Which we do with **`(*reflect.StructType)`**;

❹ A **`StructType`** has a number of methods, one of which is `Field(`n`)` which returns the n^{th} field of a structure. The type of this return is a **`StructField`**;

❺ The struct **`StructField`** has a `Tag` member which returns the tag-name as a string. So on the 0^{th} field we can unleash `.Tag` to access this name: `Field(0).Tag`. This *finally* gives us `namestr`.

To make the difference between looking a types and values more clear, that a look at the following code:

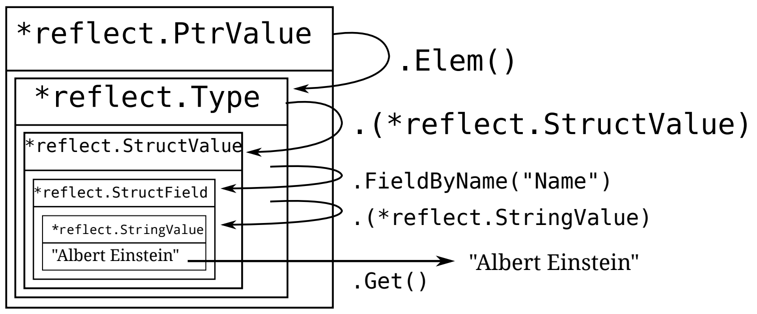
Listing 5.10. Reflection and the type and value

```
func show(i interface{}) {
    switch t := i.(type) {
      case *Person:
        r := reflect.NewValue(i)     ←− Enter the world of reflection
        tag := ❶
          r.(*reflect.PtrValue).Elem().Type().(*reflect.StructType).Field
               (0).tag
        nam := ❷
          r.(*reflect.PtrValue).Elem().(*reflect.StructValue).Field(0)\
               newline.(*reflect.StringValue).Get()
```

❶ Here we want to get the "tag", which means going for the type. Thus we need `Elem().Type().`*`(*reflect.StructType)`* to get to it;

❷ Now we want to get access to the *value* of one of the members and we employ `Elem().`*`(*reflect.StructValue)`* to get to it. Now we have arrived at the structure. Then we go the the first field `Field(0)`, tell *reflect* is a `*reflect.StringValue` and invoke the `Get()` method on it.

Figure 5.1. Pealing away the layers using reflection. Going from a ***Person** *via* ***reflect.PtrValue** *using the methods described in* `godoc reflect` *to get the actual* **string** *contained deep within.*



Reflection works by pealing off layers once you have got your hands on a **Value** in the reflection world.

Setting a value works similarly as getting a value, but only works on *exported* members. Again some code:

Listing 5.11. Reflect with private member

```
type Person struct {
 name string ”namestr”
 age  int
}

func Set(i interface{}) {
 switch t := i.(type) {
 case *Person:
  r := reflect.NewValue(i)
  r.(*reflect.PtrValue).Elem().
    (*reflect.StructValue).
    FieldByName(”name”).
    (*reflect.StringValue).
    Set(”Albert Einstein”)
  }
}
```

Listing 5.12. Reflect with public member

```
type Person struct {
 Name string ”namestr”     ←
 age  int
}

func Set(i interface{}) {
 switch t := i.(type) {
 case *Person:
  r := reflect.NewValue(i)
  r.(*reflect.PtrValue).Elem().
   (*reflect.StructValue).
   FieldByName(”Name”).     ←
   (*reflect.StringValue).
   Set(”Albert Einstein”)
  }
}
```

The code on the left compiles and runs, but when you run it, you are greeted with a stack trace and a *runtime* error:

```
panic: cannot set value obtained via unexported struct field
```

The code on the right works OK and sets the member `Name` to ”Albert Einstein”. Of course this only works when you call `Set()` with a pointer argument.

Profiling

todo 6prof

Exercises

Q13. (6) Interfaces and compilation

1. The code in listing 5.6 compiles OK — as stated in the text. But when you run it you'll get a runtime error, so something *is* wrong. Why does the code compile cleanly then?

Q14. (6) Map function with interfaces Use the answer from exercise Q11, but now make it generic using interfaces.

Q15. (6) Pointers

1. Suppose we have defined the following structure:

```
type Person struct {
    name string
    age  int
}
```

What is the difference between the following two lines?

```
var p1 Person
p2 := new(Person)
```

2. What is the difference between the following two allocations?

```
func Set(t *T) {
    x = t
}
```

and

```
func Set(t T) {
    x= &t
}
```

Q16. (5) Pointers and reflection

1. One of the last paragraphs in section "Introspection and reflection" on page 49, has the following words:

> “
> *The code on the right works OK and sets the member `Name` to "Albert Einstein". Of course this only works when you call `Set()` with a pointer argument.*

Why is this the case?

Q17. (6) Linked List

1. Make use of the package *container/list* to create a (double) linked list. Push the values 1, 2 and 4 to the list and then print it.
2. Create your own linked list implementation. And perform the same actions is in question 1

Q18. (6) Cat

1. Write a program which mimics Unix program `cat`. For those who don't know this program, the following invocation displays the contents of the file `blah`:

```
% cat blah
```

2. Make it support the n flag, where each line is numbered.

Q19. (8) Method calls

1. Suppose we have the following program:

   ```
   package main

   import ”container/vector”

   func main() {
           k1 := vector.IntVector{}
           k2 := &vector.IntVector{}
           k3 := new(vector.IntVector)
           k1.Push(2)
           k2.Push(3)
           k3.Push(4)
   }
   ```

   What are the types of `k1`, `k2` and `k3`?

2. Now, this program compiles and runs OK. All the `Push` operations work even though the variables are of a different type. The documentation for `Push` says:

   > *func (p *IntVector) Push(x int) Push appends x to the end of the vector.*

   So the receiver has to be of type **`*IntVector`**, why does the code above work then?

Answers

A13. (6) Interfaces and compilation

1. The code compiles because an integer type implements the empty interface and that is the check that happens at compile time.
   A proper way to fix this, is to test if such an empty interface can be converted and if so, call the appropriate method. The Go code that defines the function `g` in listing 5.5 – repeated here:

   ```
   func g(any interface{}) int { return any.(I).Get() }
   ```

   Should be changed to become:

   ```
   func g(any interface{}) int {
       if v, ok := any.(I); ok {   // Check if any can be converted
           return v.Get()          // If so invoke Get()
       }
       return -1                   // Just so we return anything
   }
   ```

   If `g()` is called now there are no run-time errors anymore. The idiom used is called ”comma ok” in Go.

A14. (6) Map function with interfaces

Listing 5.13. A generic map function in Go

```
package main
import ”fmt”

/* define the empty interface as a type */
type e interface{}

func mult2(f e) e {
    switch f.(type) {
    case int:
        return f.(int) * 2
    case string:
        return f.(string) + f.(string) + f.(string) + f.(string)
    }
    return f
}

func Map(n []e, f func(e) e) []e {
    m := make([]e, len(n))
    for k, v := range n {
        m[k] = f(v)
    }
    return m
}

func main() {
```

```
    m := []e{1, 2, 3, 4}
    s := []e{"a", "b", "c", "d"}
    mf := Map(m, mult2)
    sf := Map(s, mult2)
    fmt.Printf("%v\n", mf)
    fmt.Printf("%v\n", sf)
}
```

A15. (6) Pointers

1. In first line: `var p1 Person` allocates a `Person`-*value* to `p1`. The type of `p1` is **`Person`**.
   The second line: `p2 := new(Person)` allocates memory and assigns a *pointer* to `p2`. The type of `p2` is **`*Person`**.
2. In the second function, `x` points to a new (heap-allocated) variable `t` which contains a copy of whatever the actual argument value is.
   In the first function, `x` points to the same thing that `t` does, which is the same thing that the actual argument points to.
   So in the second function, we have an "extra" variable containing a copy of the interesting value.

A16. (5) Pointers and reflection

1. When called with a non-pointer argument the variable is a copy (call-by-value). So you are doing the reflection voodoo on a copy. And thus you are *not* changing the original value, but only this copy.

A17. (6) Linked List

1.
2.

A18. (6) Cat

1. A solution might be:

Listing 5.14. A cat program

```
package main

import (
        "os"
        "fmt"
        "bufio"
        "flag"
)

var numberFlag = flag.Bool("n", false, "number each line")

func cat(r *bufio.Reader) {
        i := 1
        for {
                buf, e := r.ReadBytes('\n')
                if e == os.EOF {
                        break
                }
                if *numberFlag {
                        fmt.Fprintf(os.Stdout, "%5d  %s", i, buf)
                        i++
                } else {
                        fmt.Fprintf(os.Stdout, "%s", buf)
                }
        }
        return
}

func main() {
        flag.Parse()
        if flag.NArg() == 0 {
                cat(bufio.NewReader(os.Stdin))
        }
        for i := 0; i < flag.NArg(); i++ {
                f, e := os.Open(flag.Arg(i), os.O_RDONLY, 0)
                if e != nil {
                        fmt.Fprintf(os.Stderr, "%s: error reading from
                             %s: %s\n",
                                os.Args[0], flag.Arg(i), e.String())
                        continue
                }
                cat(bufio.NewReader(f))
        }
}
```

A19. (8) Method calls

1. The type of `k1` is **`vector.IntVector`**. Why? We use a composite literal (the `{}`), so we get a value of that type back. The variable `k2` is of **`*vector.IntVector`**, because we take the address (&) of the composite literal. And finally `k3` has also the type **`*vector.IntVector`**, because `new` returns a pointer to the type.
2. The answer is given in [3] in the section "Calls", where among other things it says:

   > *A method call `x.m()` is valid if the method set of (the type of) `x` contains `m` and the argument list can be assigned to the parameter list of `m`. If `x` is addressable and `&x`'s method set contains `m`, `x.m()` is shorthand for `(&x).m()`.*

   In other words because `k1` is addressable and **`*vector.IntVector`** *does* have the `Push` method, the call `k1.Push(2)` is translated by Go into `(&k1).Push(2)` which makes the type system happy again (and you too — now you know this).

6 Concurrency

- Parallelism is about performance;
- Concurrency is about program design.

Google IO 2010
Robe Pike

In this chapter we will show off Go's ability for concurrent programming using channels and goroutines. Goroutines are the central entity in Go's ability for concurrency. But what *is* a goroutines? From [2]:

> "
> *They're called goroutines because the existing terms — threads, coroutines, processes, and so on — convey inaccurate connotations. A goroutine has a simple model:* it is a function executing in parallel with other goroutines in the same address space. *It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.*

The following idea for a program was taken from [16]. We run a function as two goroutines, the goroutines wait for an amount of time and them print something to the screen. The `main` function waits long enough, so that both goroutines will have printed their text. Right now we wait for 5 seconds (`time.Sleep()` counts in ns) on line 18, but in fact we have no idea how long we should wait until all goroutines have exited. If we did not wait for the goroutines (i.e. remove line 18) the program would be terminated immediately and any running goroutines would die with it.

Listing 6.1. Go routines in action

```
func ready(w string, sec int) {                       1
        time.Sleep(int64(sec) * 1e9)                  2
        fmt.Println(w, "is ready!")                   3
}                                                     4

func main() {                                         6
        go ready("Tee", 2)                            7
        go ready("Coffee", 1)                         8
        fmt.Println("I'm waiting")                    9
        time.Sleep(5 * 1e9)                           10
}                                                     11
```

What we need here is some kind of mechanism which allows us to communicate with the goroutines. This mechanism is available to us in the form of channels. A channel can be compared to a two-way pipe in Unix shells. You can send to and receive values from it. Those values can only be of a specific type: the type of the channel. If we define a channel, we must also define the type of the values we can send on the channel:

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface{})
```

Makes `ci` a channel on which we can send and receive integers, makes `cs` a channel for strings and `cf` a channel for types that satisfy the empty interface. Sending on a channel and receiving from it, is done with the same operator: **`<-`**. Depending on the operands it figures out what to do:

```
ci <- 1         ⟵ Send the integer 1 to the channel ci
<-ci            ⟵ Receive an integer from the channel ci
i := <-ci       ⟵ Receive from the channel ci and storing it in i
```

Lets put this to use.

Listing 6.2. Go routines and a channel

```
var c chan int ❶                                             1

func ready(w string, sec int) {                              3
        time.Sleep(int64(sec) * 1e9)                         4
        fmt.Println(w, "is ready!")                          5
        c <- 1  ❷                                            6
}                                                            7

func main() {i                                               9
        c = make(chan int) ❸                                 10
        go ready("Tee", 2) ❹                                 11
        go ready("Coffee", 1)                                12
        fmt.Println("I'm waiting, but not too long")         13
        <-c ❺                                                14
        <-c ❻                                                15
}                                                            16
```

❶ Declare `c` to be a variable that is a channel of **`int`**s. That is: this channel can move integers. Note that this variable is global so that the goroutines have access to it;

❷ Send the integer 1 on the channel `c`;

❸ Initialize `c`, note that we must use `make` here;

❹ Start the goroutines as usual;

❺ Wait until we receive a value from the channel. Note that the value we receive is discarded;

❻ Two goroutines, two values to receive.

There is still some remaining ugliness; we have to read twice from the channel (lines 14 and 15). This is OK in this case, but what if we don't know how many goroutines we started? This is where another Go built-in comes in: **`select`**. With **`select`** you can (among other things) listing for incoming data on a channel.

Using **`select`** in our program does not really make it shorter, because we run too few goroutines. We remove the lines 14 and 15 and replace them with the following:

Listing 6.3. Using ***select***

```
L: for {
        select {
        case <-c:
                i++
                if i > 1 {
                        break L
                }
        }
}
```

todo
block 'n stuff, close 'n closed

Make it run in parallel

While our goroutines were running concurrent, they were not running in parallel. When you do not tell Go anything there can only be one goroutine running at a time. With `runtime.GOMAXPROCS(n)` you can set the number of goroutines that can run in parallel. From the documentation:

> *GOMAXPROCS sets the maximum number of CPUs that can be executing simultaneously and returns the previous setting. If n < 1, it does not change the current setting.* This call will go away when the scheduler improves.

If you do not want to change any source code you can also set an environment variable `GOMAXPROCS` to the desired value.

Exercises

Q20. (4) Channels

1. Modify the program you created in exercise Q2 to use channels, in other words, the function called in the body should now be a goroutine and communication should happen via channels. You should not worry yourself on how the goroutine terminates.
2. There are a few annoying issues left if you resolve question 1. One of the problems is that the goroutine isn't neatly cleaned up when `main.main()` exits. And worse, due to a race condition between the exit of `main.main()` and `main.shower()` not all numbers are printed. It should print up until 9, but sometimes it prints only to 8. Adding a second quit-channel you can remedy both issues. Do this.[1]

Q21. (7) Fibonaci II

1. This is the same exercise as the one given page 26 in exercise 10. For completeness the complete question:

   > *The Fibonaci sequence starts as follows:* $1, 1, 2, 3, 5, 8, 13, \ldots$ *Or in mathematical terms:* $x_1 = 1; x_2 = 1; x_n = x_{n-1} + x_{n-2} \quad \forall n > 2$. *Write a function that takes an* ***int*** *value and gives that many terms of the Fibonaci sequence.*

   But now the twist: You must use channels.

[1]You will need the `select` statement.

Answers

A20. (4) Channels

1. A possible program is:

   Listing 6.4. Channels in Go

   ```
   package main                                          1

   import "fmt"                                          3

   func main() {                                         5
           ch := make(chan int)                          6
           go shower(ch)                                 7
           for i := 0; i < 10; i++ {                     8
                   ch <- i                               9
           }                                             10
   }                                                     11

   func shower(c chan int) {                             13
           for {                                         14
                   j := <-c                              15
                   fmt.Printf("%d\n", j)                 16
           }                                             17
   }                                                     18
   ```

   We start of in the usual way, then at line 6 we create a new channel of **`int`**s. In the next line we fire off the function `shower` with the `ch` variable as it argument, so that we may communicate with it. Next we start our for-loop (lines 8-10) and in the loop we send (with **`<-`**) our number to the function (now a goroutine) `shower`. In the function `shower` we wait (as this blocks) until we receive a number (line 15). Any received number is printed (line 16) and then continue the endless loop started on line 14.

2. An answer is

   Listing 6.5. Adding an extra quit channel

   ```
   package main

   import "fmt"

   func main() {
           ch := make(chan int)
           quit := make(chan bool)
           go shower(ch, quit)
           for i := 0; i < 10; i++ {
                   ch <- i
           }
           quit <- false   // or true, does not matter
   ```

```
}                                                          13

func shower(c chan int, quit chan bool) {                  15
        for {                                              16
                select {                                   17
                case j := <-c:                             18
                        fmt.Printf("%d\n", j)              19
                case <-quit:                               20
                        break                              21
                }                                          22
        }                                                  23
}                                                          24
```

On line 20 we read from the quit channel and we discard the value we read. We could have used `q := <-quit`, but then we would have used the variable only once — which is illegal in Go. Another trick you might have pulled out of your hat may be: `_ = <-quit`. This is valid in Go, but the Go idiom favors the one given on line 20.

A21. (7) Fibonaci II

1. The following program calculates the Fibonaci numbers using channels.

Listing 6.6. A Fibonaci function in Go

```
package main
import "fmt"

func dup3(in <-chan int) (<-chan int, <-chan int, <-chan int) {
        a, b, c := make(chan int, 2), make(chan int, 2), make(chan int
             , 2)
        go func() {
                for {
                        x := <-in
                        a <- x
                        b <- x
                        c <- x
                }
        }()
        return a, b, c
}

func fib() <-chan int {
        x := make(chan int, 2)
        a, b, out := dup3(x)
        go func() {
                x <- 0
                x <- 1
                <-a
                for {
                        x <- <-a+<-b
                }
        }()
        return out
}

func main() {
        x := fib()
        for i := 0; i < 10; i++ {
                fmt.Println(<-x)
        }
}

// See sdh33b.blogspot.com/2009/12/fibonacci-in-go.html
```

7 Communication

Communication with the outside world:

- Files
- Input/Output, Stdin, Stdout
- Networking
- Starting other programs
- Forking
- Databases
- Web

Files

Reading from (and writing to) files is easy in Go. This program only uses the *os* package to read data from the file `/etc/passwd`.

Listing 7.1. Reading from a file (unbufferd)

```
package main

import ”os”

func main() {
        buf := make([]byte, 1024)
        f, _ := os.Open(”/etc/passwd”, os.O_RDONLY, 0666)
        defer f.Close()
        for {
                n, _ := f.Read(buf)
                if n == 0 {
                        break
                }
                os.Stdout.Write(buf[0:n])
        }
}
```

If you want to use buffered IO there is the *bufio* package:

Listing 7.2. Reading from a file (bufferd)

```
package main

import (
        ”os”
        "bufio"                                          5
)                                                        6

func main() {                                            8
        buf := make([]byte, 1024)                        9
        f, _ := os.Open("/etc/passwd", os.O_RDONLY, 0666) 10
        defer f.Close()                                  11
        r := bufio.NewReader(f)                          12
        w := bufio.NewWriter(os.Stdout)                  13
        defer w.Flush()                                  14
        for {                                            15
                n, _ := r.Read(buf)                      16
                if n == 0 {                              17
                        break                            18
                }                                        19
                w.Write(buf[0:n])                        20
        }                                                21
}                                                        22
```

On line 12 we create a **`bufio.Reader`** from `f` which is of type **`*File`**. `NewReader` expects an **`io.Reader`**, so you might think this will fail. But it doesn't. An **`io.Reader`** is defined as:

```
type Reader interface {
    Read(p []byte) (n int, err os.Error)
}
```

So *anything* that has such a `Read()` function implements this interface. And from listing 7.1 (line 10) we can see that **`*File`** indeed does so.

Executing commands

The *exec* package has function to run external commands, and it the premier way to execute commands from within a Go program. We start commands with the `Run` function:

```
func Run(argv0 string, argv, envv []string, dir string, stdin, stdout,
     stderr int) (p *Cmd, err os.Error)
```

> "
> *Run starts the binary prog running with arguments `argv` and environment `envv`. It returns a pointer to a new **`Cmd`** representing the command or an error.*

Lets execute `ls -l`:

```
import "exec"

cmd, err := exec.Run("/bin/ls", []string{"ls", "-l"}, nil, "", exec.
     DevNull, exec.DevNull, exec.DevNull})
```

In the *os* package we find the `ForkExec` function. This is another way (but more low level) to start executables.[1] The prototype for `ForkExec` is:

[1]There is talk on the Gonuts mailing list about separating `Fork` and `Exec`.

```
func ForkExec(argv0 string, argv []string, envv []string, dir string, fd
    []*File) (pid int, err Error)
```

With the following documentation:

> “
>
> *`ForkExec` forks the current process and invokes `Exec` with the file, arguments, and environment specified by `argv0`, `argv`, and `envv`. It returns the process id of the forked process and an **`Error`**, if any. The `fd` array specifies the file descriptors to be set up in the new process: `fd[0]` will be Unix file descriptor 0 (standard input), `fd[1]` descriptor 1, and so on. A `nil` entry will cause the child to have no open file descriptor with that index. If `dir` is not empty, the child chdirs into the directory before execing the program.*

Suppose we want to execute `ls -l` again:

```
import "os"

pid, err := os.ForkExec("/bin/ls", []string{"ls", "-l"}, nil, "", []*os.
    File{ os.Stdin, os.Stdout, os.Stderr})
defer os.Wait(pid, os.WNOHANG)     ← Otherwise you create a zombie
```

Exercises

Q22. (8) Processes

1. Write a program that takes a list of all running processes and prints how many child processes each parent has spawned. The output should look like:

   ```
   Pid 0 has 2 children: [1 2]
   Pid 490 has 2 children: [1199 26524]
   Pid 1824 has 1 child: [7293]
   ```

   - For acquiring the process list, you'll need to capture the output of `ps -e -opid,ppid,comm`. This output looks like:

     ```
       PID  PPID COMMAND
      9024  9023 zsh
     19560  9024 ps
     ```

   - If a parent has one child you must print `child`, is there are more than one print `children`;
   - The process list must be numerically sorted, so you start with pid 0 and work your way up.

   Here is a Perl version to help you on your way (or to create complete and utter confusion).

   Listing 7.3. Processes in Perl

   ```
   #!/usr/bin/perl -l
   use strict;
   use warnings;
   ```

```
my (%child, $pid, $parent);
my @ps=‘ps -e -opid,ppid,comm‘;     # Capture the ouput from ‘ps‘
foreach (@ps[1..$#ps]) {            # Discard the header line
    ($pid, $parent, undef) = split; # Split the line, discard ’comm’
    push @{$child{$parent}}, $pid;  # Save the child PIDs on a list
}
# Walk through the sorted PPIDs
foreach (sort { $a <=> $b } keys %child) {
    print ”Pid ”, $_, ” has ”, @{$child{$_}}+0, ” child”,  # Print them
        @{$child{$_}} == 1 ? ”: ” : ”ren: ”, ”[@{$child{$_}}]”;
}
```

Answers

A22. (8) Processes

1. There is lots of stuff to do here. We can divide our program up in the following sections:
    1. Starting `ps` and capturing the output;
    2. Parsing the output and saving the child PIDs for each PPID;
    3. Sorting the PPID list;
    4. Printing the sorted list to the screen

   In the solution presented below, we've opted to use *container/vector* to hold the PIDs. This "list" grows automatically.
   The function `atoi` (lines 19 through 22) is defined to get ride of the multiple return values of the original `strconv.Atoi`, so that it can be used inside function calls that only accept one argument, as we do on lines 45, 47 and 50.
   A possible program is:

Listing 7.4. Processes in Go

```
package main                                                      1

import (                                                          3
        "os"                                                      4
        "fmt"                                                     5
        "sort"                                                    6
        "bufio"                                                   7
        "strings"                                                 8
        "strconv"                                                 9
        "container/vector"                                        10
)                                                                 11

const (                                                           13
        PID = iota                                                14
        PPID                                                      15
)                                                                 16

func atoi(s string) (x int) {                                     18
        x, _ = strconv.Atoi(s)                                    19
        return                                                    20
}                                                                 21

func main() {                                                     23
        pr, pw, _ := os.Pipe()                                    24
        defer pr.Close()                                          25
        r := bufio.NewReader(pr)                                  26
        w := bufio.NewWriter(os.Stdout)                           27
        defer w.Flush()                                           28
        pid, _ := os.ForkExec("/bin/ps", []string{"ps", "-e", "-opid,   29
             ppid,comm"}, nil, "", []*os.File{nil, pw, nil})
        defer os.Wait(pid, os.WNOHANG)
        pw.Close()

        child := make(map[int]*vector.IntVector)
        s, ok := r.ReadString('\n') // Discard the header line
        s, ok = r.ReadString('\n')
        for ok == nil {
                f := strings.Fields(s)
                if _, present := child[atoi(f[PPID])]; !present {
                        v := new(vector.IntVector)
                        child[atoi(f[PPID])] = v
                }
                // Save the child PIDs on a vector
                child[atoi(f[PPID])].Push(atoi(f[PID]))
                s, ok = r.ReadString('\n')
        }

        // Sort the PPIDs
        schild := make([]int, len(child))
        i := 0
        for k, _ := range child {
                schild[i] = k
                i++
        }
        sort.SortInts(schild)
        // Walk throught the sorted list
        for _, ppid := range schild {
                fmt.Printf("Pid %d has %d child", ppid, child[ppid].
                    Len())
                if child[ppid].Len() == 1 {
                        fmt.Printf(": %v\n", []int(*child[ppid]))
                } else {
                        fmt.Printf("ren: %v\n", []int(*child[ppid]))
                }
        }
}
```

A Exercices

These exercises need to be put in ”their” chapter. They are included here so the at least validate as LaTeX.

Exercises

Q23. (3) Minimum and maximum

1. Write a function that calculates the maximum value in an `int` slice (`[]int`).
2. Write a function that calculates the minimum value in a `int` slice (`[]int`).

Q24. (5) Bubble sort

1. Write a function that performs Bubble sort on slice of `int`s. From [23]:

   “

   > *It works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements ”bubble” to the top of the list.*

Q25. (5) Word and letter count

1. Write a small program that reads text from standard input and performs the following actions:

   1. Count the number of characters (including spaces);
   2. Count the number of words;
   3. Count the numbers of lines.

   In other words implement `wc(1)` (check you local manual page), however you only have to read from standard input.

Q26. (4) Average

1. Write a function that calculates the average of a `float` slice.

Answers

A23. (3) Minimum and maximum

1. This the function for calculating a maximum:

```
func max(l []int) (max int) {
        max = l[0]
        for _, v := range l {
                if v > max {
                        max = v
                }
        }
        return
}
```

2. This the function for calculating a minimum:

```
func min(l []int) (min int) {
        min = l[0]
        for _, v := range l {
                if v < min {
                        min = v
                }
        }
        return
}
```

A24. (5) Bubble sort

1.

A25. (5) Word and letter count

1. The following program is an implementation of `wc(1)`.

Listing A.1. wc(1) in Go

```
package main

import (
        ”os”
        ”fmt”
        ”bufio”
        ”strings”
)

func main() {
        var chars, words, lines int
        r := bufio.NewReader(os.Stdin)
        for {
                switch s, ok := r.ReadString(’\n’); true {
                        case ok != nil:
                                fmt.Printf(”%d %d %d\n”, chars, words, lines);
                                return
                        default:
                                chars += len(s)
                                words += len(strings.Fields(s))
                                lines++
                        }
                }
        }
```

A26. (4) Average

1. The following function calculates the average.

Listing A.2. Average function in Go

```
func average(xs []float) (ave float) {
        sum := 0
        switch len(xs) {
        case 0:
                ave = 0
        default:
                for _, v := range xs {
                        sum += v
                }
                ave = sum / len(xs)
        }
        return
}

func main() {
        ...
}
```

B Articles

Expression versus statement

This section comes from [22].

In this book we talk about expressions and statements, but what *is* the difference between the two? In short:

Expression
: Something which evaluates to a value, like: `1+2/x` ;

Statement
: A line of code which does something, like `goto Error` .

The distinction was crystal-clear in the earliest general-purpose programming languages, like FORTRAN. In FORTRAN, a statement was one unit of execution: ”a thing that you did”. An expression on its own couldn't do anything. You had to assign it to a variable.

```
1 + 2 / X
```

is an error in FORTRAN, because it doesn't do anything. You had to do something with that expression:

```
X = 1 + 2 / X
```

The earliest popular language to blur the lines was C. The designers of C realized that no harm was done if you were allowed to evaluate an expression and throw away the result. In C, every expression could be a statement:

```
1 + 2 / x
```

is a totally legit statement even though absolutely nothing will happen. Why? Because in C, expressions could have side-effects — they could change something:

```
1 + 2 / callfunc(12)
```

Because `callfunc()` might just do something useful. Once you allow any expression to be a statement, you might as well allow the assignment operator (=) inside expressions. That's why C lets you do things like: `callfunc(x = 2)`. This evaluates the expression `x = 2` (assigning the value of 2 to x) and then passes that (the 2) to the function `callfunc()`.

This blurring of expressions and statements occurs in all the C-derivatives (C, C++, C#, Java and of course Go), which still have some statements (like **`while`**) but which allow almost any expression to be used as a statement. Functional languages like Lisp don't have statements. All they have is expressions.

C 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. In no particular order:

Miek Gieben `<miek@miek.nl>`
: Main text, exercises and answers;

JC van Winkel
: Proof reading and changes to the main text.

Help with proof reading, checking exercises and little amounts of text: *Mayuresh Kathe*, *Sonia Keys*, *Makoto Inoue*, and *Russel Winder*.

Further reading

In the process of creating this book we have read huge amounts of text, mostly on the Internet. They are all referenced in the bibliography, but here follows a more detailed list of the interesting ones.

Detailed explanation about `defer` and the panic and recover mechanism in Go
: See `http://blog.golang.org/2010/08/defer-panic-and-recover.html`.

D Index

E Bibliography

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

[2] Go Authors. Effective go. http://www.golang.org/doc/effective_go.html.

[3] Go Authors. Go language specification. http://golang.org/doc/go_spec.html.

[4] Go Authors. Go package documentation. http://golang/doc/pkg/.

[5] Go Authors. Go tutorial. http://www.golang.org/doc/go_tutorial.html.

[6] Haskell Authors. Haskell. http://www.haskell.org/.

[7] Perl Package Authors. Comprehensive perl archive network. http://cpan.org.

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

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

[10] Mark C. Chu-Carroll. Google's new language: Go. http://scienceblogs.com/goodmath/2009/11/googles_new_language_go.php.

[11] Ericsson Cooperation. Erlang. http://www.erlang.se/.

[12] Brian Kernighan Dennis Ritchie. The c programming language. ...

[13] James Gosling et al. Java. ...

[14] C. A. R. Hoare. Communicating sequential processes (csp). .....

[15] C. A. R. Hoare. Quicksort. http://en.wikipedia.org/wiki/Quicksort.

[16] Roberto Costumero Moreno. Curso de go.

[17] Rob Pike. The go programming language, day 2. http://golang.org/doc/GoCourseDay2.pdf.

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

[19] Bjarne Stroustrup. The c++ programming language. ...

[20] Ian Lance Taylor. Go interfaces. http://www.airs.com/blog/archives/277.

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

[22] Stackoverflow user ΤΖΩΤΖΙΟΥ. Expression vs statement. http://stackoverflow.com/questions/19132/expression-versus-statement.

[23] Wikipedia. Bubble sort. http://en.wikipedia.org/wiki/Bubble_sort.

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

[25] Wikipedia. Duck typing. `http://en.wikipedia.org/wiki/Duck_typing`.

[26] Wikipedia. Iota. `http://en.wikipedia.org/wiki/Iota`.

This page is intentionally left blank.