Go course

Petr Shevtsov

2019-05-27

Contents

Preface

The course is targeted at a beginner level student new to Go but might be familiar with 1-2 other languages (e.g. Python, HTML).

Module 1

Basic concepts

1.1 What is Go?

1.1.1 Welcome to Go

Go is a general purpose programming language.

It was designed as a "C for the 21st century" with scalability and concurrency in mind. It belongs to the C-family, like C++, Java and C#. It also has characteristics of a dynamic language, so Ruby or Python programmers would also find it comfortable to work with.

Go is a compiled, statically typed language with garbage collection.

What does it mean?

Compilation

During the compilation the source code you wrote is translated into the low-level language natural to the computer to execute. Compiled languages are tend to **run faster** because they operate closer to the "bare metal", but sometimes it is unpleasant to work with compiled languages because the compilation can be slow. Compilation speed is on of the Go's benefits, it was designed to be **quick to compile**.

Static typing

Being statically typed means that variables **must be of a specific type** (text string, number, boolean, list, etc). Using a type system, the compiler is able to

detect problems earlier, before the program is actually used.

Garbage collection

Languages with garbage collectors are able to keep track of variables and free them when they **are no longer used**.

Go is used to create computer programs. Anything from graphics and mobile application to machine learning and networked servers can be written in Go.

The following notable software pieces are written (or have some parts) written in Go:

- Docker
- Kubernetes
- Dropbox
- OpenShift
- YouTube
- Google Chrome

1.1.2 Quiz

Go is a:

- Client-side scripting language
- **General purpose programming language**
- Machine learning program

1.1.3 Quiz

Which of the following is true?

- Go is an interpreted language
- **Go has garbage collection**
- Go is a dynamically typed language
- Go compiles into a virtual machine byte code

1.2 Hello, world!

1.2.1 Your first Go program

A "Hello, world!" program is traditionally used to introduce programmers to a programming language. Below is a Go code that outputs "Hello, world!":

```
package main

import "fmt"

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

```
## Hello, world!
```

Let's break down the code to understand each line:

```
package main
```

Every Go file belongs to one (and only one) *package*. The package to which the code file belongs must be indicated before any other code lines. A standalone executable belongs to package `main`.

```
import "fmt"
```

This line tells Go that this program needs some elements (in out case the function `Println`) from the package `fmt`, which implements formatted I/O analogous to `printf` and `scanf` in C. The package names are enclosed within double quotes (`""`).

```
function main() { }
```

When the program executes, the first function called will be `main()` (like in C). The code in functions (or the body) is enclosed between braces: `{ }`

> The first `{` must be on the same line as the function declaration!

```
    fmt.Println("Hello, world!")
```

This line calls the function `Println` from the package `fmt`, which prints the string parameter to the console, followed by a newline character `\n`.

> The same result can be obtained with `fmt.Print("Hello, world!\n")`

1.2.2 Quiz

Fill in the blanks to import the `fmt` package:

```
import "fmt"
```

1.2.3 Quiz

What is the starting point for a computer program written in Go?

- **Main function**
- Fist line
- `package main`

1.2.4 Quiz

Rearrange the code blocks to form a valid Go program:

```
package main

import "fmt"

func main() {
    fmt.Println("Go is awesome!")
}
```

1.3 The Go CLI

1.3.1 Getting the tools

TODO:

Installation instructions

1.4 Value types

1.4.1 Boolean types

A boolean type contains either `true` or `false`. The boolean type is `bool`

1.4.2 Numerical types

Go has the well known types such as `int`. The length of this type depends on the machine, so on 32-bit machine it is 32 bits while on 64-bit machine it is 64 bits. `uint` type is like `int` but it stores unsigned values. This type also has the appropriate length for the machine.

However, If you want to be explicit about the length you can use either `int32` or `uint32`.

The full list of integers (signed and unsigned) is the following: `int8`, `int16`, `int64`, `byte`, `uint8`, `uint16`, `uint32` and `uint64`.

> `byte` is an alias for `uint8`.

For floating point values there is `float32` and `float64`. There is *no* machine dependent `float` type.

> A 64 bit integer or floating point value is *always* 64 bit, even on 32-bit architectures.

1.4.3 Strings

Another important built-in type is `string`.

1.4.4 Runes

`rune` is an alias for `int32`. It is an UTF-8 encoded code point.

1.4.5 Complex numbers

Go has native support for complex numbers. Complex numbers types are `complex64` and `complex128`.

1.5 Variables

1.5.1 Variable names

A variable is a name for an area in memory. Creating a variable reserves a memory location, or a space in memory for storing values.

The name of a variable (also called the identifier) in Go is a sequence of letters, digits and underscore character. The first character in a variable name must be a letter or an underscore.

For example:

```
x
_z5
```

Letters in Go variable names are not limited to the letters of the Latin alphabet, so the following variable names are also valid:

```
å č
```

The naming of identifiers for variables follows the `camelCase` rules (start with a small letter, every new part of the word starts with a capital letter): `startDate`, `wordCount`.

1.5.2 Declaration and assignment

Variables can be declared using the `var` keyword. Go is different from other languages (e.g. C or C++) in that the type of a variable is specified *after* the variable name.

In C:

```
int a;
```

In Go:

```
var a int
```

> Multiple variables of the same type can be also declared on a single line: `var x, y int`.

Multiple `var` declarations may also be grouped:

```
var (
    n int
    s string
)
```

When a variable is declared it contains the default zero or null value for its type: `0` for `int`, `false` for `bool`, empty string (`""`) for `string`, etc.

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

```
var a int
var b bool
a = 42
b = true
```

```
var a int = 42
var b bool = true
```

Go can infer the type of the declared and assigned variable:

```
var a = 42   // inferred type: int
var b = true // inferred type: bool
```

1.5.3 Short declaration syntax

With the type omitted, the keyword `var` is pretty superfluous, so we may write as the following:

```
a := 42
b := true
```

> The types of `a` and `b` (`int` and `bool`) are inferred by the compiler.

You can also make use of parallel assignment:

```
a, b := 42, true
```

1.5.4 Discard assignments

A special name for a variable is `_` (underscore). Any value assigned to it is discarded:

```
// we only assign the integer value of 36 to b and discard the value 25.
_, b := 25, 36
```

1.5.5 Declaration and usage

A variable which is used, but not declared, gives a compiler error:

```
package main

import "fmt"

func main() {
    fmt.Println(a)
}
```

```
undefined: a
```

Declared but otherwise unused variables are a compiler error in Go:

```
package main

func main() {
```

```
    var x int
}
```

```
x declared and not used
```

1.6 Operators

TODO:

1.7 Constants

TODO:

1.8 Comments

Comments are explanatory information that you can include in the Go code to explain what the code is doing. The compiler ignores comments, so they have no affect on a program.

Go uses C++-style comments: `//` for single-line comments that finish at the end of the line and `/* … */` for comments than can span multiple lines.

For example:

```
package main

import "fmt"

func main() {
    // Output a string "Hello, world!"
    fmt.Println("Hello, world!")
}
```

```
## Hello, world!
```

> Comments clarify the program's intent to the reader. Later on you'll learn how to use comments to produce documentation for the Go code.

1.8.1 Quiz

Which of the following indicates a single-line comment?

- `-- single-line comment`
- `// single-line comment`
- `# single-line comment`

1.9 Packages and imports

TODO:

Module project

```go
// This is a comment.
// Every Go file must be a part of some package.
// This file is a part of package main.
package main

// We import package "fmt" from the standard Go library.
import "fmt"

// `who` is a constant.
const who = "world"

// Function main() is the main entry point of any application written in Go.
func main() {
    // We declare `greeting` as a variable of type string and assign the value.
    var greeting string = "Hello"
    // We declare `message` variable using the shorthand syntax. The type of
    // the variable is determined by the assigned value. In our case it is
    // string type.
    message := greeting + ", " + who
    // Let's print the value of the variable `message` using the function from
    // the package "fmt".
    fmt.Println(message)
}
```

```
## Hello, world
```

Module 2

Conditionals and Loops

2.1 The `if` statement

2.2 The `else` statement

2.3 `if`/`else` chains

2.4 The `if` statement with expression

2.5 The `switch` statement

2.6 The `switch` without condition

2.7 The `for` statement

2.8 The `defer` statement

Module project

```
package main

import "fmt"
```

```
func main() {
    // Here we loop from 0 to 99 and only output some sentences for particular
    // cases.
    for age := 0; age < 99; age++ {
        switch age {
        case 16:
            fmt.Println("When you're", age, "you can drive a car!")
        case 18:
            fmt.Println("When you're", age, "you can buy a lottery ticket!")
        case 21:
            fmt.Println("When you're", age, "you can buy some beer!")
            break
        default:
            continue
        }
    }
}
```

```
## When you're 16 you can drive a car!
## When you're 18 you can buy a lottery ticket!
## When you're 21 you can buy some beer!
```

Module 3

Composite Data Types

```
package main

import (
    "fmt"
    "strings"
)

// Shakespeare contains the text of one of the Shakespeare's sonets. We'll use
// this text to count words in it.
const Shakespeare = `
From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thy self thy foe, to thy sweet self too cruel:
Thou that art now the world's fresh ornament,
And only herald to the gaudy spring,
Within thine own bud buriest thy content,
And tender churl mak'st waste in niggarding:
  Pity the world, or else this glutton be,
  To eat the world's due, by the grave and thee.
`

// getWord returns a word in lower case and with trimmed punctuation.
func getWord(s string) string {
    return strings.ToLower(strings.Trim(s, ",:."))
}

func main() {
    wordCount := make(map[string]int)
    // Here we loop through the slice of words produced using strings.Fields
    // function. That function splits text into a slice of strings using
    // space-characters: whitespaces, tabs, new line symbols, etc.
    for _, word := range strings.Fields(Shakespeare) {
        // Trim punctuation and make it lower case.
        word = getWord(word)
        // Increase the count. If the there was no such word in the map it uses
        // zero as its count.
        wordCount[word]++
    }
```

```
    // Loop through the map and print its keys and values.
    for word, count := range wordCount {
        fmt.Println(word, count)
    }
}
```

```
## art 1
## herald 1
## churl 1
## else 1
## thee 1
## too 1
## mak'st 1
## waste 1
## this 1
## from 1
## creatures 1
## as 1
## thou 2
## contracted 1
## and 3
## due 1
## foe 1
## bud 1
## eat 1
## we 1
## that 2
## rose 1
## his 2
## a 1
## buriest 1
## world 1
## tender 2
## thine 2
## feed'st 1
## with 1
## sweet 1
## thereby 1
## heir 1
## where 1
## cruel 1
## might 2
## spring 1
## desire 1
## bright 1
```

```
## ornament 1
## gaudy 1
## within 1
## to 4
## flame 1
## by 2
## thy 5
## fuel 1
## self 2
## world's 2
## die 1
## abundance 1
## only 1
## glutton 1
## should 1
## eyes 1
## light's 1
## content 1
## time 1
## decease 1
## memory 1
## self-substantial 1
## lies 1
## making 1
## famine 1
## now 1
## increase 1
## never 1
## but 2
## the 6
## own 2
## fresh 1
## or 1
## niggarding 1
## pity 1
## be 1
## fairest 1
## beauty's 1
## riper 1
## bear 1
## in 1
## grave 1
```

Module 4

Functions and pointers

```go
package main

import "fmt"

// fibonacci returns the nth Fibonacci number.
func fibonacci(n int) int {
    if n < 2 {
        return n
    }
    return fibonacci(n-1) + fibonacci(n-2)
}

// fibonacciSequence changes the int slice to make it contain the Fibonacci
// numbers according to its keys. This function operates on the actual slice,
// that's why it does not return anything.
func fibonacciSequence(slice []int) {
    for n := range slice {
        slice[n] = fibonacci(n)
    }
}

func main() {
    // Create an empty int slice of length 10
    sequence := make([]int, 10)
    // Fill the slice with Fibonacci numbers secuence
    fibonacciSequence(sequence)

    fmt.Println(sequence)
}
```

```
## [0 1 1 2 3 5 8 13 21 34]
```

Module 5

Methods

5.1 Method declarations

5.2 Methods with a pointer receiver

5.3 Composing types with structs

5.4 Working with struct methods

5.5 Method values

5.6 Method expressions

5.7 Encapsulation

Module project

```
package main

import (
    "fmt"
)
```

```go
// printer is a struct with no fields. It only has a method.
type printer struct{}

// receipt is a variadic function. It can be called with any number of
// arguments, just like fmt.Println()
func (p printer) receipt(a ...interface{}) {
    fmt.Println(a...)
}

// Account represents a bank account data structure, it has one field and an
// embedded struct.
type Account struct {
    balance int
    printer
}

// NewAccount created a new Account setting the initial balance.
func NewAccount(balance int) *Account {
    return &Account{
        balance: balance,
    }
}

// Deposit increases the account balance by the specified amount.
// It prints the information about the operation using the method of the
// embedded printer struct.
func (a *Account) Deposit(amount int) {
    a.receipt("--> trying to deposit", amount)
    a.balance = a.balance + amount
}

// Withdraw checks if the account balance is not lesser than the amount to
// withdraw and decreses the balance by the specified amount.
// It prints the information about the operation using the method of the
// embedded printer struct.
func (a *Account) Withdraw(amount int) {
    a.receipt("<-- trying to withdraw", amount)
    if amount > a.balance {
        a.receipt("Withdraw error: not enough funds to withdraw", amount)
        return
    }
    a.balance = a.balance - amount
}

// Balance outputs the account balance using the Method of the embedded printer
```

```
// struct.
func (a Account) Balance() {
    a.receipt("Account balance:", a.balance)
}

func main() {
    account := NewAccount(100)
    account.Balance()

    account.Withdraw(25)
    account.Balance()

    account.Deposit(50)
    account.Balance()

    account.Withdraw(1000)
    account.Balance()
}
```

```
## Account balance: 100
## <-- trying to withdraw 25
## Account balance: 75
## --> trying to deposit 50
## Account balance: 125
## <-- trying to withdraw 1000
## Withdraw error: not enough funds to withdraw 1000
## Account balance: 125
```

Module 6

Interfaces

6.1 Introduction

6.2 Interface types

6.3 Satisfaction

6.4 `flag.Value`

6.5 Interface values

6.6 Sorting with `sort.Interface`

6.7 The `error` interface

6.8 Type assertions

6.9 Type switches

Module project

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "sort"
    "strings"
)

// Shakespeare contains the text of one of the Shakespeare's sonets. We'll use
// this text to count words in it.
const Shakespeare = `
From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thy self thy foe, to thy sweet self too cruel:
Thou that art now the world's fresh ornament,
And only herald to the gaudy spring,
Within thine own bud buriest thy content,
And tender churl mak'st waste in niggarding:
  Pity the world, or else this glutton be,
  To eat the world's due, by the grave and thee.
`

// WordCountPair is a record of word and its count.
type WordCountPair struct {
    Word  string
    Count int
}

// getWord returns a word in lower case and with trimmed punctuation.
func getWord(s string) string {
    return strings.ToLower(strings.Trim(s, ",:."))
}

// Functions can be types too!
type lessFunc func(p1, p2 *WordCountPair) bool

// multiSorter implements the Sort interface, sorting the word-count pairs.
```

```
type multiSorter struct {
    wordCountPairs []WordCountPair
    less           []lessFunc
}

// Sort sorts the argument slice according to the less functions passed to
// OrderedBy.
func (ms *multiSorter) Sort(wordCountPairs []WordCountPair) {
    ms.wordCountPairs = wordCountPairs
    sort.Sort(ms)
}

// OrderedBy returns a Sorter that sorts using the less functions, in order.
// Call its Sort method to sort the data.
func OrderedBy(less ...lessFunc) *multiSorter {
    return &multiSorter{
        less: less,
    }
}

// Len is part of sort.Interface.
func (ms *multiSorter) Len() int {
    return len(ms.wordCountPairs)
}

// Swap is part of sort.Interface.
func (ms *multiSorter) Swap(i, j int) {
    ms.wordCountPairs[i], ms.wordCountPairs[j] =
        ms.wordCountPairs[j], ms.wordCountPairs[i]
}

// Less is part of sort.Interface. It is implemented by looping along the less
// functions until it finds a comparison the discriminates between the two items
// (one is less than the other).
func (ms *multiSorter) Less(i, j int) bool {
    p, q := &ms.wordCountPairs[i], &ms.wordCountPairs[j]
    // Try all but the last comparison.
    var k int
    for k = 0; k < len(ms.less)-1; k++ {
        less := ms.less[k]
        switch {
        case less(p, q):
            // p < q, so we have a decision.
            return true
        case less(q, p):
```

```
            // p > q, so we have a decision.
            return false
        }
        // p == q; try the next comparison.
    }
    // All comparisons to here said "equal", so just return whatever the final
    // comparison reports.
    return ms.less[k](p, q)
}

// WordCount counts words read from input (io.Reader interface) and returns the
// word-count pairs.
func WordCount(input io.Reader) []WordCountPair {
    m := make(map[string]int)
    scanner := bufio.NewScanner(input)
    scanner.Split(bufio.ScanWords)
    for scanner.Scan() {
        // Read a word using word scanner, trim punctuation and make it lower case.
        word := getWord(scanner.Text())
        // Increase the count. If the there was no such word in the map it uses
        // zero as its count.
        m[word]++
    }

    // Create a slice the same length as the word-count map.
    pairs := make([]WordCountPair, len(m))

    // Fill the silce with data from the map.
    i := 0
    for word, count := range m {
        pairs[i] = WordCountPair{Word: word, Count: count}
        i++
    }

    return pairs
}

func main() {
    input := strings.NewReader(Shakespeare)
    pairs := WordCount(input)

    // Closures that order the WordCountPair structure.
    word := func(p1, p2 *WordCountPair) bool {
        return p1.Word < p2.Word
    }
```

```
    count := func(p1, p2 *WordCountPair) bool {
        return p1.Count > p2.Count // Note: > orders downward.
    }

    OrderedBy(count, word).Sort(pairs)

    for _, pair := range pairs {
        fmt.Println(pair.Word, pair.Count)
    }
}
```

```
## the 6
## thy 5
## to 4
## and 3
## but 2
## by 2
## his 2
## might 2
## own 2
## self 2
## tender 2
## that 2
## thine 2
## thou 2
## world's 2
## a 1
## abundance 1
## art 1
## as 1
## be 1
## bear 1
## beauty's 1
## bright 1
## bud 1
## buriest 1
## churl 1
## content 1
## contracted 1
## creatures 1
## cruel 1
## decease 1
## desire 1
## die 1
## due 1
```

```
## eat 1
## else 1
## eyes 1
## fairest 1
## famine 1
## feed'st 1
## flame 1
## foe 1
## fresh 1
## from 1
## fuel 1
## gaudy 1
## glutton 1
## grave 1
## heir 1
## herald 1
## in 1
## increase 1
## lies 1
## light's 1
## mak'st 1
## making 1
## memory 1
## never 1
## niggarding 1
## now 1
## only 1
## or 1
## ornament 1
## pity 1
## riper 1
## rose 1
## self-substantial 1
## should 1
## spring 1
## sweet 1
## thee 1
## thereby 1
## this 1
## time 1
## too 1
## waste 1
## we 1
## where 1
## with 1
## within 1
```

```
## world 1
```

Module 7

Goroutines and channels

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

const (
    NumberOfRacers   = 10
    NumberOfLaps     = 3
    MaxSleepDuration = 3 // seconds
)

func init() {
    rand.Seed(time.Now().UnixNano())
}

func race(racer int, start chan struct{}, finish chan int, status chan []int, wg *sync.WaitGroup)
    defer wg.Done()
    <-start
    for lap := 1; lap <= NumberOfLaps; lap++ {
        sleep := time.Duration(rand.Intn(MaxSleepDuration))
        time.Sleep(sleep * time.Second)
        go func(racer, lap int) {
            status <- []int{racer, lap}
        }(racer, lap)
    }
    finish <- racer
}

func main() {
    start := make(chan struct{})
    finish := make(chan int)
    status := make(chan []int)
    done := make(chan struct{})
    var wg sync.WaitGroup
    wg.Add(NumberOfRacers)
    for racer := 1; racer <= NumberOfRacers; racer++ {
        go race(racer, start, finish, status, &wg)
    }

    go func() {
```

```
        wg.Wait()
        close(done)
    }()

    startTime := time.Now()
    close(start)

    ticker := time.NewTicker(time.Second)
    defer ticker.Stop()

    var winners []int

OuterLoop:
    for {
        select {
        case t := <-ticker.C:
            fmt.Println("Race time:", t.Sub(startTime))
        case s := <-status:
            fmt.Printf("Racer #%d is on lap %d\n", s[0], s[1])
        case finished := <-finish:
            fmt.Printf("Racer #%d finished!\n", finished)
            if len(winners) < 3 {
                winners = append(winners, finished)
            }
        case <-done:
            break OuterLoop
        }
    }

    close(finish)
    close(status)

    fmt.Println("\nWinners:")
    place := 1
    for _, racer := range winners {
        fmt.Printf("%d place: Racer #%d\n", place, racer)
        place++
    }
}
```

```
## Racer #5 is on lap 1
## Racer #6 is on lap 1
## Racer #7 is on lap 2
## Racer #7 is on lap 1
## Race time: 1.000354387s
```

```
## Racer #3 is on lap 2
## Racer #5 finished!
## Racer #9 is on lap 1
## Racer #3 is on lap 1
## Racer #5 is on lap 2
## Racer #5 is on lap 3
## Race time: 2.000360272s
## Racer #2 is on lap 1
## Racer #4 is on lap 1
## Racer #10 is on lap 2
## Racer #6 is on lap 2
## Racer #7 finished!
## Racer #8 finished!
## Racer #1 is on lap 2
## Racer #10 is on lap 1
## Racer #7 is on lap 3
## Racer #8 is on lap 1
## Racer #8 is on lap 2
## Racer #8 is on lap 3
## Racer #1 is on lap 1
## Racer #3 finished!
## Racer #3 is on lap 3
## Race time: 3.000341974s
## Racer #9 is on lap 2
## Racer #2 is on lap 2
## Racer #6 finished!
## Racer #6 is on lap 3
## Race time: 4.000240107s
## Racer #1 finished!
## Racer #4 finished!
## Racer #10 finished!
## Racer #1 is on lap 3
## Racer #4 is on lap 2
## Racer #4 is on lap 3
## Racer #10 is on lap 3
## Race time: 5.000290642s
## Racer #2 finished!
## Racer #9 finished!
##
## Winners:
## 1 place: Racer #5
## 2 place: Racer #7
## 3 place: Racer #8
```

Course project

```
package main

import (
    "encoding/csv"
    "fmt"
    "io"
    "log"
    "os"
    "strconv"
    "strings"
)

func main() {
    var input io.Reader

    switch len(os.Args) {
    case 1:
        input = os.Stdin
    case 2:
        f, err := os.Open(os.Args[1])
        if err != nil {
            log.Fatal(err)
        }
        defer f.Close()
        input = f
    default:
        log.Fatal("This program expects either 0 or 1 arguments.")
    }

    r := csv.NewReader(input)
    r.FieldsPerRecord = -1

    records, err := r.ReadAll()
```

```
    if err != nil {
        log.Fatal(err)
    }

    // Remove the very first "record" (i.e 'Category: All categories') if exists
    if len(records[0]) == 1 {
        records = append(records[:0], records[1:]...)
    }

    // Save names to a slice
    names := records[0][1:] // Skip 'weeks' column
    commonSuffix := longestCommonSuffix(names)
    if commonSuffix != "" {
        for i, name := range names {
            names[i] = strings.TrimSuffix(name, commonSuffix)
        }
    }
    records = append(records[:0], records[1:]...)

    avg := make([]int, len(names))

    for _, record := range records {
        record = record[1:] // Skip 'weeks' column
        for i, s := range record {
            n, err := strconv.Atoi(s)
            if err != nil {
                log.Fatal(err)
            }
            avg[i] += n
        }
    }

    for i := 0; i < len(avg); i++ {
        avg[i] = avg[i] / len(records)
    }

    for i, n := range avg {
        n = n/10 + 1
        fmt.Printf("%s %s (%d)\n", strings.Repeat(" ", n), names[i], avg[i])
    }
}

func longestCommonSuffix(a []string) string {
    if len(a) == 0 {
        return ""
```

```
    }

    suffix := a[0]
    if len(a) == 1 {
        return suffix
    }

    for _, s := range a[1:] {
        suffixLength := len(suffix)
        sLength := len(s)

        if suffixLength == 0 || sLength == 0 {
            return ""
        }

        maxLength := suffixLength
        if sLength < maxLength {
            maxLength = sLength
        }

        for i := 0; i < maxLength; i++ {
            j := suffixLength - i - 1
            k := sLength - i - 1
            if suffix[j] != s[k] {
                suffix = suffix[j+1:]
                break
            }
        }
    }
    return suffix
}
```