Программирование на языке Go

для любопытных

Содержание

Краткое введение

Имеется много литературы по программированию на Go, в том числе и на русском языке. Однако, для всех этих текстов, как и для книг по программированию вообще, характерны одни и те же недостатки. Для начинающих программистов эти книги слишком тяжеловесны и перегружены деталями, бесполезными на этапе первого знакомства с языком. С другой стороны для людей, умеющих программировать хотя бы на одном современном языке, не требуется изложение материала, начиная «с Адама и Евы». Например, совсем не обязательно подробно описывать систему базовых типов. Для всех языков она примерно одинакова, а те нюансы, что имеются в данном конкретном языке легко узнать из документации, а также понять особенности на рассматриваемых примерах. Кроме того, все книги по программированию грешат ненужным многословием, вызывающим одну лишь скуку и «томление души». Кроме того, я противник длинных идентификаторов, которые бывают полезны в больших реальных программах и то только для обозначения долгоживущих объектов. В коротких примерах, на которых обычно и знакомятся с языком, и где переменные и функции существуют лишь на нескольких строчках кода, все обозначения должны быть максимально краткими. Поэтому я буду всегда функции обозначать буквой ***f***, переменные - ***x*** и так далее. По моему мнению в книге по программированию надо также использовать только самые простейшие примеры; более сложными программами обучающийся должен овладевать самостоятельно. Попробуем говорить о программировании на Go просто и «своими словами», не утомляя читателя заумной терминологией.

Теперь несколько общих характеристик языка Go. Язык компилируемый с хорошей скоростью компиляции. Имеет строгую статическую типизацию. При этом допускает инициализацию переменных при их объявлении и умеет сам выводить тип. Одновременно допускается и динамическая типизация в особых случаях. В языке имеется эффективный механизм поддержки

параллелизма. В своей основе язык процедурный (императивный), но допускает и программирование в объектно-ориентированном стиле с некоторыми особенностями. Был период всеобщего увлечения ООП, но, кажется, что теперь ситуация меняется и снова появился интерес к «старому доброму» процедурному программированию. И это, несомненно, правильно: использование классов оправдано только там, где они действительно необходимы. Перспективным, на мой взгляд, является также функциональный стиль программирования, применяющий только чистые, без побочных эффектов, функции, благодаря чему естественным образом достигается даже более строгая инкапсуляция, чем в ООП.

В целом можно считать, что Go – это примерно тоже, что и язык Си, улучшенный и приведённый к уровню современных языков. Поэтому он не имеет того множества недостатков, которыми обладают те многочисленные языки, что реализованы на виртуальной машине Java. Разработчики Go даже считают, что это язык будущего. На самом деле по сравнению с такими языками, как Ruby, Scala, Clojure, Groovy и так далее, язык Go — это возврат обратно к старым традициям программирования. Впрочем, может быть для того, чтобы сделать качественный рывок вперёд, действительно стоит сначала вернуться назад?

Файлы с программами на Go должны иметь расширение ***.go***. Для компиляции и выполнения программы из командной строки надо перейти в директорию, содержащую файл программы, и использовать команду ***go*** с дополнительными подкомандами. Так, для компиляции и выполнения программы достаточно написать:

go run prog.go

где ***prog.go*** – исходный файл. Если надо сохранить скомпилированный и собранный файл, применяется команда:

go build prog.go

Очень удобно пользоваться редактором Geany, из которого можно руководить компиляцией и выполнением программ. Имеется и собственный IDE для Go – Gogland.

1. Базовые структуры

1.1 Hello, World!

Не нарушая традиции начнём с программы Hello на Go:

```
package main
import "fmt"
func main()  {
      fmt.Println("Привет, мир!") → Привет, мир!
}
```

Уже на этом простейшем примере можно узнать довольно много о языке.

Все программы на Go оформляются в виде пакетов. Поэтому в первой строке всегда будет слово ***package*** и название пакета. В виде пакетов оформляются и модули (библиотеки, подпрограммы, процедуры), названия которых могут быть произвольными. Но головной пакет всегда называется ***main*** и в нём обязательно присутствие функции ***main***, с которой и начинается выполнение программы. Кстати, большинство современных языков давно уже избавились от этих main, программа выполняется просто строка за строкой, сверху вниз. В сочетании с интерактивным режимом (Repl) это даёт огромные преимущества при отладке программ.

Кроме функции main в Go имеется ещё одна особая функция с идентификатором ***init***, объявление которой имеет вид:

```
func init() { . . . }
```

Она может быть с аргументами или без аргументов, а в её теле могут программироваться любые действия. Любой файл может содержать любое количество таких функций. Функцию ***init*** нельзя вызвать или обратиться к ней, но во всех прочих отношениях она не отличается от нормальных функций. В каждом файле функции ***init*** выполняются автоматически при запуске программы в том порядке, в котором они объявлены. Следовательно, с помощью этой функции можно запрограммировать какие-то действия, которые должны быть обязательно выполнены при каждом запуске программы.

После названия пакета в Go должна идти директива импорта других пакетов, необходимых для работы программы. У нас импортируется один библиотечный пакет ***fmt***, содержащий функцию вывода ***Println***, которой мы далее и воспользовались. Пакет ***fmt*** содержит и другие функции ввода-вывода, а также разные вспомогательные средства для них, благодаря которым Go обладает большими возможностями обмена информацией с внешним миром.

(Вообще-то, я думаю, импорт пакета ***fmt***, а также и некоторых других, было бы целесообразно обеспечить по умолчанию). Отметим ещё, что имя импортируемого пакета указывается в кавычках и что наши собственные пакеты импортируются точно также, как и библиотечные. При импорте нескольких пакетов список их имён заключается в круглые скобки, а имена разделяются точкой с запятой. Недопустимо импортировать не используемый пакет — компилятор выдаст ошибку.

Далее мы объявляем ту самую функцию ***main***, о которой упоминалось выше. Как видим, объявляются функции со словом ***func***, аргументы перечисляются в круглых скобках, которые должны присутствовать и в случае отсутствия аргументов, как у нас. Тело функции заключается в фигурные скобки, как и во многих других языках (далеко не во всех). Обращение к функции из другого пакета должно быть с указанием имени пакета через точку.

С помощью стрелки ***(→)*** я указал результат выполнения программы. Так будем поступать и в дальнейшем, заменяя стрелкой слова «получим», «будет выведено» и тому подобное. При копировании кода для выполнения эти стрелки и текст за ними надо убрать.

1.2 Переменные и константы

Переменные можно объявлять разными способами. Видимо, основным надо считать объявление со словом ***var*** по схеме:

var name type = expression

Например:

var x int = 25

Переменная ***x*** имеет тип ***int***, который далее не может быть изменён, можно только присвоить переменной другое значение:

x = 7 * 9

Объявить переменную можно и без её инициализации:

var x int

При этом она получит значение, равное ***0***. Текстовые переменные в аналогичном случае имеют значение ***""*** (пустая строка), а булевые — ***false***. Некоторые другие сущности (ещё познакомимся с ними), по умолчанию получают специальное значение ***nil***. Значит, в Go исключаются неопределённые переменные.

Явное указание типа можно опускать:

var x = 25

При этом компилятор сам выведет тип по контексту. Ясно, что нельзя объявить просто ***var x***, здесь будет неопределённость. Допустимо объявлять тип сразу нескольких переменных:

var i, j, k int

Или даже так:

var b, f, s = true, 2.3, "four"

Переменные ***b, f, s*** получили типы ***bool, float64, string*** соответственно. Отметим, что по умолчанию числа с плавающей точкой получают тип ***float64***, что соответствует типу ***double*** в других языках. Типу ***float*** в Go соответствует тип ***float32***. Используя групповое присваивание можно обмениваться значениями переменных без промежуточного переменного:

x, y = y, x

Вторая форма объявления (краткая) имеет вид:

name := expression

То-есть, без слова ***var***, но со знаком присваивания ***(:=)***. Тип здесь всегда выводится компилятором. Эта краткая форма объявления переменных не допустима на уровне пакета и применяется только в теле функций. Кажется, что присутствие этой второй формы даёт не слишком много пользы, а путаница в некоторых ситуациях вполне возможна.

Для изменения значения уже объявленной ранее переменной применяется знак равенства:

x = 0.325

Кроме переменных, в Go можно использовать константы, которые нельзя изменить после первоначальной инициализации. Константы объявляются также, как и переменные, но со словом ***const***:

const c float64 = 12.34

Или без указания типа:

const s = "Пушкин"

Чтобы не повторять слово ***const***, можно объявить несколько констант так:

```
const (
    a = 0
    b = 1
```

```
    c = 2
)
```

Или в одной строке:

const (a = 0; b = 1; c = 2)

Также можно объявлять и переменные:

var (x = 35; y = true; z = "Мир")

1.3 Указатели

Как и в языке Си, допускается использование указателей. Указатель — это адрес переменной. С его помощью можно считывать и изменять значение переменной, не используя её имя. Применяется такой синтаксис: пусть объявлена и инициализирована переменная ***x***:

var x string = "Москва"

Тогда ***&x*** – адрес переменной ***x*** и мы можем этим адресом инициализировать какую-нибудь переменную, например:

p := &x

Это и будет указателем, а его тип - ****string***. Обращаться к содержимому переменной ***x*** теперь можно с помощью указателя, пользуясь выражением ****p***:

fmt.Println(*p) → Москва

Можно изменять значение ***x***:

****p = "Ленинград"***

fmt.Println(x) → Ленинград - значение ***x*** изменилось

Фактически ****p*** – это псевдоним для ***x***. При объявлении указателя без инициализации он принимает значение ***nil***:

```
var q *string
fmt.Println(q) → nil
```

Указатели можно передавать функциям в качестве аргументов. Посмотрим на примере:

```
package main
import "fmt"
func main()  {
    var x = 5
    var y = f(&x)
    fmt.Println(y) → 36
    fmt.Println(x) → 6
}
func f(p *int) int  {
```

```
    *p++
    var a = *p * *p
    return a
}
```

Здесь в теле функции ***main*** вызывается функция ***f***, аргументом которой является указатель. При вызове функции ***f*** :

var y = f(&x)

****p*** будет указывать на переменную ***x*** и иметь тип ****int***. После списка аргументов объявляемой функции требуется указывать тип возвращаемого результата (у нас ***int***). Результат возвращается с помощью директивы ***return***. Разумеется, что вводить дополнительную переменную ***a*** не обязательно, а можно так:

return *p * *p

Директива ****p++*** является инкрементом, увеличивающим значение ***x*** на единицу. Соответственно, имеется и декремент: ****p--***

Функция ***f*** может быть объявлена как до, так и после функции ***main***, компилятору это безразлично.

Еще одним способом создания переменной является применение встроенной функции ***new***. Выражение ***new (T***) создает не именованную переменную типа ***T***, инициализирует ее нулевым значением и возвращает ее адрес, который представляет собой значение типа ****T***.

p := new(int)

Здесь ***p***, имеет тип ****int*** и указывает на не именованную переменную типа ***int***

fmt.Println(*p) → 0

****p = 2*** - Устанавливает значение этой переменной равным ***2***

fmt.Println(*p) → 2

Переменная, созданная с помощью ***new***, ничем не отличается от обычной локальной переменной, у которой берется адрес, за исключением того, что нет необходимости придумывать (и объявлять) ее имя.

Ясно, что в наших примерах от указателей не много пользы, но далее мы ещё увидим, когда применение их действительно целесообразно.

Отметим здесь попутно тот факт, что в Go применяется только кодировка UTF-8. Это позволяет, в частности, пользоваться

кириллицей наравне с латиницей. В частности, наш пример мы могли бы написать и так:

```
package main
import "fmt"
func main() {
	var ш = 5
	var ю = ж(&ш)
	fmt.Println(ю) → 36
	fmt.Println(ш) → 6
}
func ж(ы *int) int {
	*ы++
	var ё = *ы * *ы
	return ё
}
```

Пожалуй, это первый язык, позволяющий делать такие вещи.

1.4 Объявление типов

Новый тип объявляется с помощью директивы ***type***, которая позволяет присвоить типу имя. Созданием своих собственных типов данных мы займёмся позже. Но иногда бывает полезно дать собственное имя и базовому типу. Посмотрим на примере:

```
package main
import "fmt"
type C float32
type F float32
func main() {
	var c C = 100
	var f F = cf(c)
	fmt.Println(f) → 212
}
func cf(x C) F {
	var y = F(x * 9/5 + 32)
	return y
}
```

Эта программа переводит температуру из шкалы Цельсия на шкалу Фаренгейта. Мы ввели два именованных типа ***C*** и ***F***. Для того, чтобы использовать эти типы в функциях ***main*** и ***cf***, они должны

объявляться глобально, то-есть за пределами этих функций. Функция ***cf*** принимает аргумент типа ***C*** и возвращает результат типа ***F***. Выражение ***F(x * 9/5 + 32)*** переводит значение типа ***C*** (в круглых скобках) в тип ***F***. Такой способ перевода применим и для базовых типов, для которых это допустимо. Например, мы можем написать:

var p float32 = 12.7
var s = int(p)
fmt.Println(s) → 12

При переводе вещественного числа в целое возвращается целая часть числа без округления.

Хотя в примере выше на самом деле все переменные имеют вид ***float32***, они поделены на два сорта и теперь нельзя передать функции ***cf*** параметр типа ***F***. В более сложных ситуациях разделение переменных на сорта предохраняет нас от возможных ошибок.

2. Составные типы

2.1 Массивы и срезы

Массивы в Go фиксированной длины и это существенно ограничивает область их применения. Все элементы массива могут быть только одного типа. Объявляется массив просто, например:

```
package main
import "fmt"
func main()  {
        var m[5] int
        m[2] = 77
        fmt.Println(m) → [0 0 77 0 0] - элементы разделены пробелами
}
```

Здесь мы объявили массив ***m*** размером ***5*** с элементами типа ***int***. Элементы можно извлекать и изменять по индексу. Если элементы не инициализированы, они принимаются по умолчанию: ***0, "", false, nil***. Функция ***len*** возвращает размер массива:

fmt.Println(len(m)) → 5

Для инициализации элементов массива списком используется блок:

var m = [5]string{"Анна", "Фёкла", "Виктор"}

В левой части размер и тип теперь указывать необязательно.

fmt.Println(m) → [Анна Фёкла Виктор] - пустые строки при выводе не отображаются, хотя, конечно, присутствуют
fmt.Println(len(m)) → 5 – хотя мы инициализировали только три элемента, размер массива по-прежнему равен ***5***.
Для массива можно вводить именованный тип:
type t [5]string
var m = t{"Анна", "Фёкла", "Виктор"}
При этом размер массива является частью типа, то-есть массивы разной длины имеют разные типы.

С помощью именованного типа можно создавать двумерный массив (матрицу):
type t [3]int
m1 := t{1, 2, 3}
m2 := t{4, 5, 6}
m := [2]t{m1, m2}
fmt.Println(m) → [[1 2 3] [4 5 6]]
fmt.Println(len(m)) → 2
Элементы массива ***m*** представлены вложенными массивами ***m1*** и ***m2***, размер массива ***m*** равен ***2***. Для извлечения элемента из матрицы применяется двойной индекс:
fmt.Println(m[1][2]) → 6

При инициализации элементов блоком, размер массива можно устанавливать автоматически. Для этого на месте размера указывается многоточие (три точки):
var m = [...]string{"Анна", "Фёкла", "Виктор"}
fmt.Println(m) → [Анна Фёкла Виктор]
fmt.Println(len(m)) → 3

Индексацию можно задать принудительно и не обязательно упорядоченно. При этом применяется такой синтаксис:
var m = [...]string{7: "Москва", 2: "Париж", 5: "Берлин"}
fmt.Println(m) → [Париж Берлин Москва] - порядок восстанавливается
fmt.Println(m[5]) → Берлин
fmt.Println(len(m)) →8
Размер массива равен наибольшему индексу плюс единица. Все не инициализированные элементы равны пустой строке.

Имеется возможность извлекать из массива отдельные куски (подмассивы), и они называется срезами. Посмотрим на пример:

var m = [...]int{1,2,3,4,5,6,7,8,9}
var s = m[3:7]
fmt.Println(s) → [4 5 6 7] - элемент с индексом ***7*** не входит в срез
fmt.Println(s[2]) → 6
fmt.Println(len(s)) → 4
fmt.Println(cap(s)) → 6

Срез ничем не отличается от массива, его элементы имеют свою собственную индикацию. По этой причине там, где это не приводит к неопределённости, я часто буду использовать привычный термин «массив» вместо «срез». Функция ***len*** даёт размер среза, а функция ***cap*** – число элементов от начала среза до конца базового массива.
var s1 = m[:4] → [1 2 3 4] - если левая граница не указана, она принимается равной ***0***
var s2 = m[5:] → [6 7 8 9] - если правая граница не указана, она принимается равной ***len(m)***

Если границы выходят за пределы базового массива, генерируется ошибка. Срез может использоваться, как базовый массив для получения нового среза. При этом новый срез может выходить за пределы исходного среза:
var s1 = s[:6]
fmt.Println(s1) → [4 5 6 7 8 9] - срез ***s1*** длинее среза ***s***

Имеется встроенная функция ***make***, позволяющая создавать срез:
m := make([]int, 5)
fmt.Println(m)→ [0 0 0 0 0]
При этом создаётся и базовый массив, не имеющий имени, элементы которого доступны только через срез. К срезу ***m*** можно применить, в частности, функцию ***cap***:
cap(m) → 5
То-есть срез и базовый массив имеют одинаковый размер. Но можно сделать базовый массив больше, чем срез, для этого надо указать ещё один аргумент:
n := make([]int, 5, 8)
fmt.Println(cap(n)) → 8
Теперь не именованный базовый массив имеет размер ***8*** (напоминаю, что функция ***cap*** возвращает число элементов от начала среза до конца базового массива, а срез ***m*** начинается с нулевого индекса). Тот же срез ***n*** может быть создан и так:
n := make([]int, 8)[:5]

Это надо понимать так, что сначала создаётся срез размером ***8***, как и базовый массив, а затем выделяется вторичный срез длиной ***5***. Не знаю, можно ли как-то использовать этот не именованный базовый массив.

Функция ***append*** позволяет наращивать срез, добавляя в него один или несколько элементов:

n = append(n, 1, 2, 3)
fmt.Println(n) → [0 0 0 0 0 1 2 3]

Значит, элементы добавляются в конец среза.
Пустой срез создаётся так:

var m[]int
fmt.Println(m) → []

В него можно добавлять элементы без ограничения:

m = append(m, 1, 2, 3)
fmt.Println(m) → [1 2 3]
m = append(m, 11, 12, 13)
fmt.Println(m) → [1 2 3 11 12 13]

Эта возможность устраняет проблему, связанную с тем, что в Go массивы имеют фиксированный размер. По этой причине почти всегда вместо массивов используются срезы.

Используя пустой срез как тип, можно создавать матрицы произвольных размеров:

```
type t []int
m1 := t{1, 2, 3}
m2 := t{4, 5, 6, 7}
m := []t{m1, m2}
fmt.Println(m) → [[1 2 3] [4 5 6 7]]
fmt.Println(m[1][3]) → 7
```

Элементы массива ***m*** – массивы разной длины, и значит правило, что массивы разной длины имеют разные типы, здесь нарушается.

Можно, конечно, инициировать элементы двумерного массива и с помощью блоков:

```
type t []int
m := []t{{1, 2, 3},{4, 5, 6, 7}}
fmt.Println(m) → [[1 2 3] [4 5 6 7]]
```

2.2 Отображения

Отображение иначе ещё называют хеш-таблицей (или просто хеш,

что мне нравится больше из-за краткости), словарём, ассоциированным списком, иногда map. Тип отображения записывается, как ***map[K]V***, где ***K***- тип ключей а ***V*** – тип значений. Типы ключей и значений могут быть любыми, но в конкретном отображении все ключи, а также и все значения должны быть одного типа. Функция ***make*** создаёт пустой хеш:

h := make(map[string]int)

fmt.Println(h) → map[] - так обозначается пустой хеш.

Заполнение хеша элементами выполняется также, как и массива:

h := map[string]int{"Иван": 25, "Сергей": 19, "Татьяна": 17}

fmt.Println(h) → map[Иван:25 Сергей:19 Татьяна:17]

Доступ к значениям хеша такой же, как и к элементам массива, только роль индексов теперь выполняют ключи. Хеш можно пополнять новыми членами:

h["Пётр"] = 31

fmt.Println(h) →map[Иван:25 Сергей:19 Татьяна:17 Пётр:31]

Как видим, элементы хэша не упорядочены и могут располагаться в произвольном порядке. Значения можно изменять:

h["Сергей"] = 39

fmt.Println(h) → map[Иван:25 Сергей:39 Татьяна:17 Пётр:31]

Функция ***delete*** позволяет удалять элементы хеша:

delete(h, "Сергей")

fmt.Println(h) → map[Пётр:31 Иван:25 Татьяна:17]

Функция ***len*** возвращает размер хеша:

fmt.Println(len(h)) → 3

Проверить, есть ли элемент с заданным ключом можно так

a, ok := h["Иван"]

fmt.Println(a, ok) → 25 true

Переменная ***ok*** (её часто так называют, хотя, конечно, это не обязательно) равна ***true***, если такой ключ есть. В противном случае получим ***false*** и нулевое значение.

Значениями хеша могут быть составные типы, например, хеши или срезы:

h := map[int][]string{1: {"a", "b"}, 2: {"c", "d", "e"}}

fmt.Println(h) → map[1:[a b] 2:[c d e]]

Значения в хеше ***h*** представлены срезами разной длины.

2.3 Структуры

Структуры позволяют объединять некоторое количество именованных данных произвольного типа в некое единое целое. Элементы структуры в Go называют полями. Для примера создадим структуру, содержащую некоторые данные, характеризующие студента:

```
type Student struct {
        ID      int
        Name    string
        Age     int
        Address string
    }
```

Создавая структуру, мы создаём новый тип, о чём свидетельствует слово ***type*** перед именем структуры. Теперь можно объявить переменную типа ***Student***, представляющую конкретный экземпляр структуры:

```
var Petrov Student
```

Поля структуры ***Petrov*** можно инициализировать конкретными значениями:

```
Petrov.Name = "Иван"
fmt.Println(Petrov) → {0 Иван 0 }
```

Не инициализированные поля принимают нулевые значения. Инициализацию полей можно выполнять с помощью хеша:

```
Petrov = Student{ID: 125, Name: "Сергей", Age: 20, Address:
"Васюки"}
fmt.Println(Petrov) → {125 Сергей 20 Васюки}
```

- выводятся только значения полей.

Инициализировать поля можно и просто перечислением значений, но тогда надо запоминать порядок полей:

```
Titova := Student{12, "Анна", 19, "Москва"}
fmt.Println(Titova) → {12 Анна 19 Москва}
```

С полями структур можно работать, как с обычными переменными, в частности можно применять указатели на них. Обычно при объявлении типа поля записывают по одному в строке, как в примере выше, но однотипные поля можно и объединять:

```
type Student struct {
        ID,Age          int
        Name, Address string
```

}

Имеет значение, с какой буквы — заглавной или прописной начинается идентификатор поля. Это влияет на область видимости поля, поля с заглавной буквой видимы в других пакетах программы. Впрочем, о видимости дальше поговорим отдельно.

3. Ветвление и циклы

3.1 Инструкция if

Инструкция ***if*** в Go имеет обычную структуру: слово ***if,*** условное выражение, блок, слово ***else***, блок. После ***else*** может снова стоять ***if*** для проверки второго условия, потом третьего и так далее. В Go не логические значения не преобразуются в логические автоматически, поэтому в условном выражении всегда применяются операторы сравнения. Особенностью инструкции ***if*** в Go является то, что между оператором ***if*** и условным выражением может располагаться произвольное выражение (в документации оно называется необязательной инструкцией). Например:

```
package main
import ("fmt"; "math")
func f(x float64) float64 {return math.Sin(x)}
func main()  {
        x := 4.2
        if y := f(x); y < 0 {fmt.Printf("%7.4f", -y)} else
        {fmt.Printf("%7.4f", y)} → 0.8716
}
```

Здесь использована функция форматированного вывода ***Printf***, для которой дополнительно задаётся формат вывода (практически также, как и в любом другом языке). О вводе — выводе в Go поговорим позже. Обращаю внимание на то, что переменная ***y*** не видна за пределами инструкции ***if***.

Ветвь ***else*** может отсутствовать, например:

```
if 5 > 3 {fmt.Println("5 больше 3")} →  5 больше 3
```

Если условие даёт ***false***, строка просто пропускается.

3.2 Инструкция switch

Переключатель ***switch*** имеет обычную форму. Посмотрим на пример:

```
package main
import "fmt"
func f(min, v, max int) int {
switch {
        case v < min:
                return min
        case v > max:
                return max
        }
        return v
}
func main()  {
        fmt.Println(f(2, 9, 7))  → 7
}
```

Также, как и в инструкции ***if***, здесь после директивы ***switch*** может стоять необязательная инструкция, а после неё через точку с запятой может располагаться выражение. На мой взгляд, эта дополнительная возможность на практике мало что даёт.

Есть и ещё один вариант переключателя ***switch***:

```
switch {
        case v < min:
                return min
        case v > max:
                return max
        default:
                return v
        }
        panic("unreachable")
```

Оператор ***panic*** в этом примере никогда не будет выполнен, тем не менее он должен присутствовать, или можно вместо него использовать ***return***.

Переключатель ***switch*** имеет возможность выбора варианта не только по значению, как в примере выше, но также и по типу. Синтаксис такого варианта посмотрим на примере:

```
package main
import "fmt"
```

```
func f(x interface{}) {
	switch x.(type) {
	case bool:
		fmt.Printf("Параметр имеет тип bool\n", x)
	case float64:
		fmt.Printf("Параметр имеет тип float64\n", x)
	case int, int8, int16, int32, int64:
		fmt.Printf("Параметр имеет тип int\n", x)
	case uint, uint8, uint16, uint32, uint64:
		fmt.Printf("Параметр имеет тип unsigned int\n", x)
	case nil:
		fmt.Printf("Параметр равен nil\n", x)
	case string:
		fmt.Printf("Параметр имеет тип string\n", x)
	default:
		fmt.Printf("Параметр имеет тип unknown\n", x)
	}
}
func main()  {
		f("Киев")
}
```

Функция ***f*** принимает переменную ***x***, тип которой указан, как ***interface{}***, что в Go означает любой тип. Выражение ***x.(type)*** называется переключателем типа, указывает, что выбор надо делать не по значению переменного, а по его типу. В одном операторе ***case*** можно проверять сразу несколько типов, для чего надо перечислить их через запятую. При вызове функции ***f*** переменная ***x*** принимает строку ***"Киев"***, результат имеет вид:

Параметр имеет тип string

Надо сказать, выбор по типу не часто встречается в практике.

3.3 Цикл for

В языке Go имеются две разновидности инструкции ***for***, простая инструкция ***for*** и ***for ... range***. Первая разновидность ничем не отличается от цикла ***for*** в других языках, включая самые ранние:

```
package main
import ("fmt")
func f(n int) {
```

```
        for i := 0; i < n; i++  {
                fmt.Print(i, " ")
        }
}
func main()  {
                f(5)
}
```

Получим: ***0 1 2 3 4***

Второй вариант цикла ***for*** посмотрим на примере распечатки аргументов командной строки. Язык Go, как и все современные языки, позволяет при запуске программы из командной строки вводить дополнительные аргументы, например:

go run prog.go Люда Лена Люба Нина

То-есть, аргументами являются значения типа ***string***, вводимые через пробел и без кавычек после команды запуска. Эти аргументы доступны в массиве ***Args***, содержащемся в пакете ***os***, который, значит, надо импортировать. Причём, элемент с индексом ***0*** содержит полное название файла программы, а следующие элементы и являются аргументами из командной строки. Программа будет выглядеть примерно так:

```
package main
import ("fmt"; "os")
func main() {
        var s, пробел string
        for _, x := range os.Args[1:] {
                s += пробел + x
                пробел = " "
        }
        fmt.Println(s) → Люда Лена Люба Нина
}
```

Выражение вида ***range m*** возвращает попарно индексы и элементы массива ***m***. (Не очень ясно, почему тут использовано слово ***range***, обычно его употребляют для других целей). Поскольку индекс у нас не используются, то на месте идентификатора для него указывается знак ***(_)*** (placeholder). Мы не можем тут указать какое-нибудь иное имя, поскольку тогда будет ошибка — не используемая переменная. Все аргументы мы собираем в строковой переменной ***s***, так как знак ***(+)*** для переменных типа ***string*** является знаком конкатенации. Перед

каждым аргументом добавляем пробел с помощью переменной ***пробел*** (Go позволяет использовать кириллицу). Причём сначала переменная ***пробел*** по умолчанию инициализирована пустой строкой и, значит, перед первым словом пробела не будет. Используем не весь массив ***Args***, а его срез ***Args[1:]***, то-есть отбрасываем не нужный нам нулевой элемент. В итоге результат будет точно таким, как мы напечатали в командной строке.

Иногда в цикле, наоборот, используются только индексы массива и не используются значения. В этом случае знак ***(_)*** указывать не обязательно:

```
package main
import ("fmt")
func main() {
	f([]int{1, 2, 3, 4, 5})
}
func f(m []int)  {
	for i := range m {fmt.Println(i)}
}
```

Хотя, можно и так:

```
for i, _ := range m {fmt.Println(i)}
```

В пакете ***strings*** есть функция ***Join***, которая позволяет из массива (или строки) сформировать строку с нужными нам разделителями. С помощью ***Join*** то же самое, можно получит так:

fmt.Println(strings.Join(os.Args[1:], " ")) → Люда Лена Люба Нина

Надо только не забыть импортировать пакет ***strings***.

Наконец, срез ***Args[1:]*** можно вывести, как есть:

fmt.Println(os.Args[1:]) → [Люда Лена Люба Нина]

4. Строки

Работа с текстом в Go имеет некоторые особенности, поэтому рассмотрим переменные типа string более подробно.

4.1 Строки, как массивы и срезы

Строки в Go содержат неизменяемую последовательность байтов и обладают многими свойствами массива. В частности, функция ***len*** возвращает длину строки в байтах:

s := "Hello"
fmt.Println(len(s)) → 5
Буква латиницы представлена в UTF-8 одним байтом, поэтому результат равен числу букв в слове ***Hello***. Но посмотрим, что будет, если использована кириллица:
s := "Привет"
fmt.Println(len(s)) → 12
Некоторые буквы кириллицы представлены в памяти несколькими байтами, поэтому мы получили размер переменной ***s*** больше, чем ***6*** букв кириллицы. И это значит, что с обработкой текста не на латинице имеют место проблемы, о которых и придётся поговорить далее.

Извлечение байта из строки выполняется по индексу, как и для массивов:
s := "Hello"
fmt.Println(s[2]) → 108
Ещё того не легче, извлекается байт (не переводится в знак), да ещё и в десятичном формате. Посмотрим на кириллицу:
t := "Привет"
fmt.Println(t[9]) → 181
Действительно, байтов больше, чем букв и мы даже не знаем, какой букве, или, может быть, части буквы соответствует число ***181***.

Перевод числа в строку выполняет функция string:
fmt.Println(string(108)) → l
Но так не получится с кириллицей:
fmt.Println(string(t[3])) - нет результата

Можно извлечь подстроку, аналог среза:
fmt.Println(s[1: 4]) → ell
fmt.Println(t[3: 9]) → ив
С латиницей всё в порядке, с кириллицей проблема, так как индексы крайних байтов не соответствуют началу и концу соответствующих знаков кириллицы. Если байт бессмысленный, Go выводит знак вопроса в ромбике.

Поскольку строки неизменяемы, нельзя присваивать байтам новые значения:
s[2] = 'g' - ошибка
Неизменяемость позволяет двум копиям строки располагаться в одной и той же памяти, что очень важно, если учесть, что чаще всего обрабатываются очень большие массивы текста.

4.2 Строковые литералы

Строковый литерал это последовательность байтов, заключённая в двойные кавычки, представляющая строковое значение (тип string). (Мне кажется увлечение всякими терминами, наподобие этого, не сильно облегчает понимание существа дела). В строковые литералы можно включать любые символы UTF-8. В строковые литералы можно вставлять также «управляющие последовательности», начинающиеся с обратного слэша ***(\)***. Одна их группа обрабатывает управляющие коды, такие, как символ перевода на новую строку ***\n***, а вторая - позволяет вставлять в строку произвольные байты, такие, как шестнадцатеричные и восьмеричные управляющие последовательности.

fmt.Print("С Новым Годом!\nС Новым Счастьем!\n") →
С Новым Годом!
С Новым Счастьем!

Не форматированный строковый литерал (raw string literal) записывается с помощью обратных одинарных кавычек ***(' . . . ')*** вместо двойных. Внутри такой строки управляющие последовательности не обрабатываются. Этот литерал может состоять из нескольких строк текста:

const GoUsage = 'Go - инструмент для работы с исходным текстом Go.
Использование:
go команда [аргументы] '

4.3 Некоторые приёмы работы с UTF-8

Пусть требуется проверить: содержит или нет одна строка другую. Рассмотрим по отдельности три возможных случая. Сначала проверим не содержится ли подстрока в начале другой строки (префикс):

func pre(s, p string) bool { return len(s) >= len(p) && s[:len(p)] == p }
fmt.Println(pre("Жили-были дед да баба", "Жили")) → true

Теперь поищем подстроку в конце строки (суффикс):

func suf(s, su string) bool { return len(s) >= len(su) && s[len(s)-len(su):] == su }
fmt.Println(suf("Жили-были дед да баба", "баба")) → true

И наконец, будем искать подстроку с произвольным расположением в строке:

```
func subs(s, sub string) bool {
        for i := 0; i < len(s); i++ {
                if pre(s[i:], sub) {
                return true
                }
        }
        return false
}
```

fmt.Println(subs("Жили-были дед да баба", "были")) → true

Здесь мы в цикле анализируем срез, укорачивая его путём отбрасывания передних знаков. Если подстрока присутствует в тексте, то она на каком-то этапе окажется префиксом, что мы и зафиксируем с помощью функции ***pre***. Можно было бы с таким же успехом воспользоваться и функцией ***suf***, отбрасывая знаки с конца строки. В библиотечном пакете strings присутствуют подобные этим функции.

В пакете unicode/utf8 имеется ряд функций для работы с текстом в этой кодировке. В частности, есть функция, позволяющая получить размер строки не в байтах, а в знаках:

```
package main
import ("fmt"; "unicode/utf8")
func main() {
        s := "Hello, Россия!"
        fmt.Println(len(s)) → 20  - размер в байтах
        fmt.Println(utf8.RuneCountInString(s)) → 14 – размер в знаках
}
```

Значит, функция ***RuneCountInString*** из пакета ***utf8*** определяет размер строки в знаках. (Не знаю, почему тут принят такой синтаксис — ***unicode/utf8***). Функция ***rune*** преобразует знаки в числа (руны), а формат ***%x*** переводит их в шестнадцатеричные числа:

```
г := []rune(s)
fmt.Printf("%x\n",г) → [48 65 6c 6c 6f 2c 20 420 43e 441 441 438 44f 21]
```

Обратное преобразование выполняет функция ***string***:

fmt.Println(string(г)) → Hello, Россия!

Для работы со строками имеются четыре стандартных пакета: ***bytes, strings, strconv*** и ***unicode***. Пакет ***strings*** предоставляет множество функций для поиска, замены, сравнения, обрезки, разделения и объединения строк. Пакет ***bytes*** содержит аналогичные функции для работы со срезами байтов ***[]byte***. Пакет ***strconv*** предоставляет функции для преобразования целых чисел, чисел с плавающей точкой и булевых значений, в строковое представление и обратно. И пакет ***unicode*** содержит такие функции, как ***IsDigit, IsLetter, IsUpper*** и ***IsLower***. Эти функции возвращают булево значение, а смысл их понятен из названия.

```
package main
import ("fmt";"unicode")
func main() {
    fmt.Println(unicode.IsUpper('Ш')) → true
}
```

В пакете ***strings*** имеются функции ***ToUpper*** и ***ToLower***, которые возвращают новые строки с преобразованиями, примененными к каждому символу исходной строки:

```
fmt.Println(strings.ToUpper("мама")) → МАМА
```

Покажем ещё несколько примеров использовани функций из пакета strings:

```
s := "Я шагаю по Москве"
substr := "шагаю"
fmt.Println(strings.Contains(s, substr)) → true
fmt.Println(strings.Count(s, "о")) → 2
fmt.Println(strings.Fields(s)) → [Я шагаю по Москве]
fmt.Println(strings.HasPrefix(s, "Я")) → true
fmt.Println(strings.Index(s, "по")) → 14
```

- к сожалению, возвращает индекс байта, не знака.

```
s := []string{"Я", "шагаю", "по", "Москве"}
fmt.Println(strings.Join(s, "|")) → Я|шагаю|по|Москве
```

Преобразования между числовыми значениями и их строковыми представлениями выполняются с помощью функций из пакета ***strconv***. Для преобразования целого числа в строку можно воспользоваться функцией ***strconv.Itoa***

```
x := 123
fmt.Println(strconv.Itoa(x)) → 123
```

Ещё лучше применить функцию ***Sprintf*** из пакета ***fmt***:

```
x := 12.54
y := fmt.Sprintf("%7.3f", x)
fmt.Println(y) → 12.540
```

5. Функции

Мы уже познакомились с функциями, а теперь рассмотрим их более подробно.

5.1 Объявление функций

Посмотрим на пример:

```
package main
import ("fmt"; "math")
func main() {
        fmt.Println(f(3, 4))
}
func f(x, y float64) float64 {
        return math.Sqrt(x*x + y*y)
}
```

Объявленная нами функция ***f*** вычисляет длину гипотенузы треугольника по заданным длинам катетов. Новую функцию нельзя объявить в теле другой функции, в примере функция ***f*** должна быть объявлена впереди или после функции ***main***, как кому нравится. Переменные ***x*** и ***y*** называют параметрами, а передаваемые функции конкретные значения ***3*** и ***4*** — аргументами. Впрочем, это не общепринято, иногда называют наоборот, и я буду этими терминами пользоваться, как мне будет казаться удобнее.

Аргументам ***x*** и ***y*** (мне так их называть нравится больше), необходимо присвоить тип (у нас ***float64***). Тип можно указать для каждого переменного отдельно, или для всей группы совместно. Если в списке аргументов разные типы, их тоже можно объявлять группами, например:

```
func f(i, j, k int, s, t string) string {  . . .  }
```

После списка аргументов должен быть указан тип возвращаемого результата — (тоже ***float64***). Без указания типов компилятор зафиксирует ошибку. Переменные ***x*** и ***y*** – локальны и видимы только в теле функции, в частности, их тип не может быть указан вне функции.

К большому сожалению (довольно часто приходится повторять эти слова) функции в языке Go не возвращают последнее вычисленное значение, как это делается во всех современных языках. Результат всегда возвращается оператором ***return***. Иногда результат представлен несколькими переменными, тогда их можно объединить в массив:

```
func f(x, y float64) []float64 { return []float64{x, y} }
f(3,4) → [3, 4]
```

Пришлось повторить объявление типа массива дважды. Для массива можно ввести локальную переменную:

```
func f(x, y float64) (r []float64) { r = []float64{x, y}; return r }
```

Но это ничего не меняет.

Таким же образом можно применять идентификатор для любого результата и указывать его при указании типа результата, например:

```
func f(x, y int) (z int) { z = x - y; return }
```

Аргумент у оператора ***return*** можно тогда опустить, а можно и оставить: ***return z***.

Используя идентификаторы, можно организовать возвращение функцией нескольких результатов, не прибегая к созданию массива:

```
package main
import ("fmt")
func main() {
        a, b := f(7, 2)
        fmt.Println(a, b) → 9 5
}
func f(x, y int) (p, r int)  {
        p = x + y; r = x - y
        return
}
```

Можно, конечно, написать ***return p, r.***

Если какой-то результат функции не используется, при групповом присваивании надо применить знак ***(_)***:

```
a, _ := f(7, 2)
fmt.Println(a) → 9
```

Наконец, можно программировать и так:

```
package main
import ("fmt")
func main() {
        fmt.Println(f(2, 7, "Hello")) → 2 7 Hello
```

```
}
func f(x, y float64, z string) (float64, float64, string) { return x, y, z }
```

В этом случае также не надо создавать массив результатов.

Функции в Go, как и переменные, тоже имеют свой тип, иногда его называют сигнатурой. Этот тип можно определить так:

```
fmt.Printf("%T\n", f) → func(float64, float64, string) (float64, float64, string)
```

Применяется форматированный вывод, где буква ***T*** указывает, что надо вывести тип. В информации о типе для функции указывается слово ***func***, типы аргументов и тип результата. Всегда можно узнать тип библиотечной функции:

```
fmt.Printf("%T\n", math.Sin) → func(float64) float64
```

К сожалению, в Go нет возможности применения именованных аргументов, поэтому передаваемые функции значения всегда должны располагаться в том порядке, как они перечислены в списке аргументов.

Функции в Go могут быть в правой части оператора присваивания и тогда объявленная таким способом переменная сама будет функцией:

```
package main
import ("fmt")
var φ = f
func main() {
	φ([]int{1, 2, 3, 4, 5})
}
func f(m []int)  {
	s := 0
	for _, a := range m {s += a}
	fmt.Println(s) → 15
}
```

Объявленную и инициализированную функцией переменную ***φ*** можно вызывать вместо функции ***f***. Как видим, оператор присваивания и объявление функции могут находится в любом месте программы.

5.2. Рекурсия

Язык Go, как и все современные языки, допускает рекурсию. Самым популярным примером рекурсии является, несомненно, вычисление факториала числа:

```
package main
import ("fmt")
func main() {
        fmt.Println(fact(5)) → 120
}
func fact(n int) int  {
        var r int
        if n == 0 || n == 1 {r = 1} else {r = n * fact(n - 1)}
        return r
}
```

На тему рекурсии так много написано во всех книгах по программированию, что мне тут, пожалуй, нечего больше добавить. Нашу программу вычисления факториала после небольшой корректировки можно выполнить почти на любом языке программирования.

Практически во всех книгах по программированию приводится пример программы, определяющей является ли заданное слово палиндромом (ROTOR, НАГАН и тому подобное). Покажу и я пример такой программы на Go:

```
package main
import ("fmt"; "unicode/utf8")
func main() {
        fmt.Println(f("pomop")) → true
}
func f(word string) bool {
        if utf8.RuneCountInString(word) <= 1 { return true }
        first, i := utf8.DecodeRuneInString(word)
        last, j := utf8.DecodeLastRuneInString(word)
        if first != last { return false }
        return f(word[i : len(word)-j])
}
```

Для решения этой задачи использована рекурсия. При каждом

новом вызове рекурсивной функции ей передаётся срез, получаемый путём отбрасывания первой и последней буквы. Использование функций из пакета ***utf8*** (с очень длинными названиями) требуется для анализа текста не на латинице. Функция ***RuneCountInString*** определяет количество знаков в строке, а функции ***DecodeRuneInString*** и ***DecodeLastRuneInString —*** выделяют первую и последнюю букву и расстояние в знаках для них от начала и конца строки соответственно. Только для английских слов можно было бы использовать одну лишь индексацию.

5.3 Аргументы функций

Функции в Go могут принимать в качестве своих аргументов всё, что имеет тип. В частности аргументами могут быть другие функции, поскольку, как мы видели, функции имеют тип (сигнатуру). Посмотрим на такой пример:

```
package main
import ("fmt")
func main() {
        var m = []int{1, 2, 3, 4}
        fmt.Println(f(m, g)) → [1 4 9 16]
        fmt.Println(f(m, h)) → [1 8 27 64]
}
func g(x int)int { return x * x }
func h(x int)int {return x * x * x}
func f(a []int, φ func(int)int) []int  {
        var s []int
        for _,i := range a {
                s = append(s, φ(i)) }
                return s
}
```

Здесь функция ***f*** принимает два аргумента: массив целых чисел ***a*** и функцию ***φ***, имеющую тип ***func(int)int***. Вызываем функцию ***f*** дважды: первый раз передаём ей функцию ***g***, возводящую целое число в квадрат; второй раз — функцию ***h*** возводящую целое число в куб. Результат возвращаем в виде массива ***s***, сформированного в цикле с помощью функции ***append***, добавляющей элемент в массив.

Аргументами функции могут быть даже массивы, элементы которых представлены функциями:

```
package main
import ("fmt")
func main() {
	fmt.Println(f(4, m)) →[16 64 40]
}
func α(x int)int { return x * x }
func β(x int)int {return x * x * x}
func γ(x int)int {return x * 10}
var m = []func(int)int {α, β, γ}
func f(ξ int, n []func(int)int) []int {
	var s []int
	for _, y := range n {
		s = append(s, y(ξ))
		}
		return s
}
```

Массив ***m*** теперь состоит из функций и имеет тип ***[]func(int)int***. В цикле ***for*** эти функции выполняются последовательно, а из возвращаемых ими результатов формируется результирующий массив ***s***. Кстати, в примере использованы греческие буквы в качестве идентификаторов, что позволяет делать язык Go.

Можно создавать функции с переменным числом аргументов. Для этого используется массив, способный принимать любое число аргументов (в терминологии Go это будет срез). Применяется такой синтаксис:

m ...int

Это значит, что ***m*** – массив целых чисел переменной длины. Применять такой синтаксис можно только в списке аргументов. Посмотрим всё это на примере функции ***Min***, выбирающей наименьшее значение из списка аргументов:

```
package main
import ("fmt")
func main() {
	fmt.Println(Min(9, 2, 12, -5)) → -5
}
func Min(n int, m ...int) int {
	for _, x := range m {
		if x < n { n = x }
```

```
        }
        return n
}
```

В этом случае в теле функции ***m*** рассматривается, как массив, содержащий столько элементов, сколько мы ввели при вызове функции. При этом переменная ***n*** получит значение ***9***, а массив ***m*** будет равен ***[2 12 -5]***. Функции с таким списком аргументов можно передать и массив, например ***s***, при этом первый элемент массива надо присвоить переменной ***n***, а срез остальных элементов передать массиву ***m***:

```
Min(s[0], s[1:])
```

Допустимо передавать функции ***Min*** только одно значение, которое получит переменная ***n***, а массив ***m*** в этом случае будет пустым, хотя, конечно, он будет создан:

```
fmt.Println(Min(27)) → 27
```

Если для некоторой функции ***f*** список аргументов определить так:

```
f(m ...int)
```

то эту функцию можно будет вызывать или с произвольным числом значений, или совсем без аргументов. Можно также перед массивом неопределённой длины ***m*** поставить не одну, а несколько переменных:

```
Min(n int, a int, b int, m ...int)
```

Ясно, что переменные ***a*** и ***b*** надо использовать отдельно, впрочем, как это ни странно, если их и не использовать, то компилятор не выдаёт ошибку. Эти дополнительные переменные могут даже иметь другой тип:

```
Min(n int, a float64, b string, m ...int)
```

Покажем ещё один вариант:

```
package main
import ("fmt")
func main() {
        var s = []int{7, -15, 2, -9, 4}
        fmt.Println(Min(9, s[2:] ...)) → -9
}
func Min(n int, m ...int) int {
        for _, x := range m {
                if x < n { n = x }
        }
```

```
        return n
}
```

Используя конструкцию ***s[2:] …*** мы передаём функции ***Mi***n не весь массив ***s***, а только его срез, начиная с индекса ***2***. Таким образом, элемент со значением ***-15*** не был использован при анализе на минимум.

5.4 Анонимные функции (замыкания, closures)

В Go анонимные функции (не имеющие имени) называют замыканиями (closures). Можно было бы объяснить смысл этого термина, но это мало что даёт для понимания существа дела. Лучше посмотрим на примерах.

Анонимная функция объявляется точно так же, как и именованная, только после слова ***func*** указывается не имя, а сразу список аргументов с их типами и тип результата, например:
func (x float64, y float64) float64 { x*math.Sin(y) }.
Мы уже видели, что функции могут быть аргументами других функций. Анонимную функцию также можно применять в качестве аргумента.

```
package main
import ("fmt"; "math")
func main() {
        r := f(5, 2.3, func(x, y float64)float64 { return x*math.Sin(y)})
        fmt.Printf("x*math.Sin(y) = %6.3f\n", r)
}
func f(α, β float64, φ func(α, β float64)float64)float64 {
                return φ(α, β)
}
```

На экран будет выведено: ***x*math.Sin(y) = 3.729***

Объявленная функция ***f*** имеет в списке аргументов две переменных ***α*** и ***β*** типа ***float64*** и функцию ***φ*** типа ***func(α, β float64)float64)***. Возвращаемый результат также имеет тип ***float64***. При вызове функции ***f*** можно было бы на место ***φ*** передать какую-нибудь обычную именованную функцию, однако мы, чтобы не вводить новое имя функции, используем closure. Очевидно, что такой приём имеет смысл только в том случае, если анонимная функция имеет не очень длинный код.

В некоторых языках программирования, например в Groovy, closures играют очень важную роль. В частности, они применяются в таких мощных и эффектных структурах, как итераторы. В нашем же примере, пожалуй, почти никакой явной пользы от применения closure мы не имеем.

Мы уже видели, что функциями можно инициировать переменные, которые сами становятся функциями. В этой же роли могут использоваться и анонимные функции. Посмотрим на ещё один пример:

```
package main
import ("fmt")
func main() {
        f(5, g) → 0 1 4 9 16 25
}
var g = func(x int) {fmt.Print(x * x, " ") }
func f(n int, φ func(a int)) {
        for i:=0; i <= n; i++ {φ(i)}
}
```

Переменная ***g*** инициирована анонимной функцией и использована в качестве аргумента при вызове функции ***f***. Обратите внимание на то, что мы не указали тип результата для функций ***f*** и ***g***, поскольку они только выводят что-то на экран и ничего не возвращают. Ясно, что в этом примере ничего не изменится если бы мы объявили функцию ***g*** как обычно, без применения closure. И тем не менее, встречаются программы, в которых closure дают существенный эффект. Отметим ещё, что переменную ***g*** можно было бы объявить не на уровне пакета, а в теле функции ***main***, при этом можно использовать сокращённую форму присваивания ***(:=)***.

Иногда требуется создать несколько похожих функций. Вместо того чтобы создавать их по отдельности, часто используются так называемые фабричные функции, то есть функции, возвращающие другие функции. Посмотрим на примере:

```
package main
import ("fmt"; "bufio"; "os")
func main() {
        fmt.Print(" Введите текст: ")
        r := bufio.NewScanner(os.Stdin)
        r.Scan()
```

```
        t := r.Text()
        v := f(t)
        fmt.Println(v(", World!"))
}
func f(a string) func(string) string {
        return func(b string) string {
                if a == "Hello" { return a + b }
                return a
        }
}
```

Объявленная здесь функция ***f*** принимает переменную ***a*** типа ***string*** и возвращает функцию типа ***func(string) string***. Эта функция возвращается оператором ***return*** в виде анонимной функции (closure). Её действия состоят в том, что если переменная ***a*** имеет значение ***Hello***, то к этому слову добавляется значение аргумента ***b*** анонимной функции. При всяком другом слове просто возвращается значение переменной ***a***.

В примере запрограммирован ввод текста с клавиатуры. В Go это делается так: во-первых надо импортировать два дополнительных пакета ***bufio*** и ***os***. Потом создаётся вспомогательная переменная ***r*** с помощью функций ***bufio.NewScanner*** и ***os.Stdin***. Выражение ***r.Scan()*** осуществляет ввод с клавиатуры в буфер. После этого можно введённый текст извлечь командой ***r.Text()***. Короче говоря, не скажешь, что это высший класс. Мы позже ещё рассмотрим ввод-вывод из файла, которые реализуются подобным же образом.

Но вернёмся к нашей фабричной функции. Вызов её с помощью выражения ***v := f(t)*** инициирует переменную ***v*** анонимной функцией, возвращаемой фабричной функцией ***f***. Значит, переменная ***v*** сама становится функцией, с помощью которой мы и получаем результат. Результат можно получить, например, такой:

Введите текст: Hello → Hello, World!
Введите текст: Good bye → Good bye

Конечно, пример примитивен и имеет целью показать только технику. Но встречаются ситуации, когда такие функции приносят выгоду. Кстати, эта программа может работать только с латиницей, при вводе текста на кириллице возникают проблемы разбираться с которыми надо отдельно.

Посмотрим на ещё один пример использования фабричной функции:

```
package main
import ("fmt")
func main() {
        g := f()
        fmt.Println(g()) → 1
        fmt.Println(g()) → 4
        fmt.Println(g()) → 9
        fmt.Println(g()) → 16
}
func f() func() int {
        var x int
        return func() int {
                x++
                return x * x
        }
}
```

В строке

```
func f() func() int {
```

объявляется функция ***f***, не имеющая аргументов и возвращающая анонимную функцию (closure) также без аргументов, которая в свою очередь возвращает число типа ***int***. В теле функции ***f*** объявлена локальная переменная ***x*** (напоминаю, что при этом она неявно получает значение ***0***). Далее оператором ***return*** возвращается анонимная функция, которая затем и объявляется. Эта функция увеличивает значение ***x*** на ***1*** и возвращает квадрат ***x***.

В теле функции ***main*** анонимной функции присваиваем имя ***g*** и четырежды вызываем её (она не имеет аргументов). В результате получаем квадраты чисел, увеличивающихся при каждом вызове ***g*** на ***1***.

Таким образом, внутренняя анонимная функция имеет доступ к переменной ***x***, объявленной в теле внешней функции ***f,*** может изменять эту переменную и её изменённое значение сохраняется при выходе из анонимной функции. В связи с этой особенностью анонимных функций их и называют замыканиями.

5.5 Обобщенные функции

Этим термином в Go обозначают функции, способные принимать аргументы любого типа. Такой обобщённый тип обозначается загадочным кодом ***interface{}*** (фигурные скобки тут обязательны). Применим этот приём на рассмотренном ранее примере поиска минимального элемента в массиве. Только теперь составим программу так, чтобы элементы массива были произвольного типа. Кстати, ещё раз используем оператор ***switch*** с выбором по типу.

```
package main
import ("fmt")
func main() {
	fmt.Println(Min("Коля", "Петя", "Вася", "Саша")) → Вася
}
func Min(a interface{}, b ...interface{}) interface{} {
	m := a
	for _, x := range b {
		switch x := x.(type) {
		case int:
			if x < m.(int) { m = x }
		case float64:
			if x < m.(float64) { m = x }
		case string:
			if x < m.(string) { m = x }
		}
	}
	return m
}
```

Эта программа способна работать с тремя типами: ***int***, ***float64***, и ***string***. Именно эти типы мы перечислили в операторах выбора ***case***. Для того, чтобы начать работу с типом, к которому относятся значения, переданные функции, введён оператор присваивания ***m := a***, определяющий тип по контексту. Выражение ***x.(type)*** возвращает тип переменной ***x.*** Соответственно, выражения ***m.(int)***, ***m.(float64)*** и ***m.(string)*** преобразуют значения переменной ***m*** из типа ***interface{}*** к соответствующему базовому типу. В строке ***switch x := x.(type)*** хотя и есть оператор присваивания, сама переменная ***x*** значения не меняет. И вообще, эта конструкция применима только в операторе ***switch***. Короче говоря, чем разбираться в деталях, легче просто

запомнить технику и использовать. В Go встречается много вещей подобного рода.

Приведём ещё один пример обобщённой функции, которая может принимать аргументы любого типа, но при этом не использует явно тип ***interface{}***. Составим программу, которая проверяет есть ли в массиве элемент с заданным значением. Если такой элемент есть, возвращается его индекс, если нет — возвращается ***(-1).***

```
package main
import ("fmt")
func main() {
	x := []int{2, 4, 6, 8}
	y := []string{"C", "B", "K", "A"}
	fmt.Println(
	f(len(x), func(i int) bool { return x[i] == 5 }),
	f(len(x), func(i int) bool { return x[i] == 6 }),
	f(len(y), func(i int) bool { return y[i] == "Z" }),
	f(len(y), func(i int) bool { return y[i] == "A" })) → -1 2 -1 3
}
func f(n int, g func(i int) bool) int {
	for i := 0; i < n; i++ {
		if g(i) { return i }
	}
	return -1
}
```

Функция ***f*** принимает целое число ***n*** и функцию ***g***, которая на самом деле является closure. Эта функция ***g*** принимает целое число и возвращает результат типа ***bool***. Как только функция ***g*** вычислит значение ***true***, цикл ***fo***r заканчивается и функция ***f*** вернёт текущее значение параметра цикла ***i***. Если же функция ***g*** не разу не вернёт ***true*** цикл отработает до конца и функция ***f*** вернёт значение ***(-1).***

В примере рассматривается два массива. Элементы массива ***x*** имеют тип ***int*** а элементы массива ***y – string***. Функция ***f*** вызывается ***4*** раза. Аргументу ***n*** передаётся размер массива, а аргументу ***g*** передаются анонимные функции, получающие в цикле ***for*** переменную цикла ***i*** и проверяющее элемент массива с этим индексом на равенство заданному значению. Соответственно, в результате получаем 4 числа. Тип элементов массивов используется только в условных выражениях проверки на равенство, сама же обобщённая

функция ***f*** ничего не знает ни о массивах ни о типах их элементов и её можно применять в самых разных ситуациях. Посмотрим на такой пример:

```
package main
import ("fmt"; "math")
func main() {
	i := f(math.MaxInt32,
	func(i int) bool { return i > 0 && i%27 == 0 && i%51 == 0 })
	fmt.Println(i) → 459
}
func f(n int, g func(i int) bool) int {
	for i := 0; i < n; i++ {
		if g(i) { return i }
	}
	return -1
}
```

Выражение ***math.MaxInt32*** возвращает максимальное целое число на ***32*** разрядном регистре, то-есть это очень большое число. Условное выражение возвращает ***true*** для числа, удовлетворяющего трём условиям: положительное, делится без остатка на ***27*** и делится без остатка на ***51***. Таким образом, число ***459*** есть наименьшее из чисел удовлетворяющих этим трём условиям. Чтобы найти следующее такое число надо цикл начать с ***460: for i := 460; i < n; i++***. Ответ равен ***918*** и это, оказывается, просто в два раза больше предыдущего результата. Здесь мы не используем никаких массивов, а просто перебираем натуральные числа по порядку и проверяем их на выполнение заданных условий.

Наберёмся терпения и посмотрим на ещё один пример использования closure:

```
package main
import ("fmt")
func main() {
	m := []int{4, -3, 2, -7, 8, 19, -11, 7, 18, -6}
	even := f(m, func(i int) bool { return i%2 == 0 })
	fmt.Println(even) → [4 2 8 18 -6]
}
func f(s []int, g func(int) bool) []int {
	r := []int{}
```

```
        for i := 0; i < len(s); i++ {
                if g(s[i]) {
                        r = append(r, s[i])
                }
        }
return r
}
```

Здесь анонимная функция позволяет отбирать из массива чисел чётные числа из которых формируется новый массив. Программу легко изменить для отбора чисел по какому-нибудь другому критерию. Но можно ли программу приспособить для фильтрации массивов с элементами произвольного типа? Придётся программу перестроить, например, так:

```
import ("fmt")
func main() {
        m := []string{"X15", "T14", "X23", "A41", "L19", "X57",
"A63"}
        var r []string
        f(len(m), func(i int) bool { return m[i][0] == 'X' },
                func(i int) { r = append(r, m[i]) })
        fmt.Println(r) → [X15 X23 X57]
}
func f(n int, g func(int) bool, φ func(int)) {
        for i := 0; i < n; i++ {
                if g(i) { φ(i) }
        }
}
```

В список аргументов функции ***f*** мы добавили функцию, накапливающую элементы в итоговом массиве. И, значит, теперь сама функция ***f*** напрямую не имеет дела с массивами, а только руководит анонимными функциями, ей передаваемыми при вызове. Это и позволяет сделать функцию ***f*** универсальной с точки зрения типа элементов исходного массива. В примере из заданного массива отбираются элементы типа string, у которых начальная буква ***X***.

Изменим тело функции main на такой код:

```
var r int64 = 1
f(26, func(i int) bool { return i%2 != 0 },
        func(i int) { r *= int64(i) })
```

fmt.Println(r) → 7905853580625
Теперь функция ***f*** использована уже не для фильтрации массива, а получения произведения натуральных нечётных чисел из диапазона ***1...25***.

6. Программирование в объектно-ориентированном стиле на Go

В языке Go не используются такие средства, как классы и экземпляры. Тем не менее, авторы Go утверждают, что возможность создавать собственные типы на основе структур позволяет реализовать технику программирования в стиле ООП. Ну, что же, посмотрим, так ли это на самом деле.

6.1 Методы

Ранее мы уже рассматривали пример создания структуры ***Student***, а теперь используем структуру ***Point*** для вычисления расстояния между двумя точками на плоскости:

```
package main
import ("fmt"; "math")
func main() {
	fmt.Println(d(Point{1, 2}, Point{4, 6} )) → 5
}
type Point struct{ x, y float64 }
func d(r, q Point) float64 {
	return math.Sqrt(math.Pow((q.x-r.x),2) + math.Pow((q.y-r.y),2))
}
```

Структура ***Point***, представляющая пользовательский тип, имеет два поля ***x*** и ***y*** типа ***float64***, определяющие положение точки на плоскости. Список аргументов объявленной функции ***d*** включает две переменных ***r*** и ***q***, имеющих тип ***Point***. Ссылки на поля структуры имеют точечную нотацию: ***r.x, r.y*** и так далее. Функция ***d*** возвращает расстояние между двумя точками, вычисленное по их координатам. Вообще-то в пакете ***math*** есть функция ***Hypot***, вычисляющая гипотенузу по катетам и можно использовать более простое выражение:

```
return math.Hypot(q.x-p.x, q.y-p.y)
```

При вызове функции ***d*** поля структуры ***Point*** инициируются конкретными значениями.

Итак, фактически ***Point*** похож на класс ООП и при инициализации полей создаются объекты вроде экземпляров класса ***Point***. На самом деле, вместо терминов класс и объект или экземпляр в Go рекомендованы названия тип и значение. Применение термина значение в этом смысле мне кажется необычным и поскольку чаще это слово имеет совсем другой смысл, и я всё-таки буду использовать термины класс, экземпляр и объект. В примере, пока эти экземпляры (значения) имеют только поля, но нет методов. К созданию методов мы теперь и приступим. На самом деле в Go это сделать просто. Например, нашу функцию ***d*** можно преобразовать в метод с помощью такого синтаксиса:

```
func (r Point)  d(q Point) float64 {
        return math.Hypot(q.x-r.x, q.y-r.y)
}
```

Выражение ***(r Point)*** (скобки обязательны) называется приемником и это аналог похожих приёмников со служебными словами this или self в других языках. В Go имя приёмника может быть любым и у нас использовано имя одного из экземпляров (вообще-то ***r*** и ***q*** можно поменять местами и в данном случае ничего не изменится). Обычно для приёмника используют первую букву имени типа, то-есть в нашем случае ***p***, но тогда и имя ***r*** надо заменить на ***p***. Программа теперь будет выглядеть так:

```
package main
import ("fmt"; "math")
func main() {
        p := Point{1, 2}
        q := Point{4, 6}
        fmt.Println(p.d(q)) → 5
}
type Point struct{ x, y float64 }
func (p Point) d(q Point) float64 {
        return math.Hypot(q.x-p.x, q.y-p.y)
}
```

Выражение ***p.d(q)*** можно трактовать так: для объекта ***p*** вызывается метод ***d*** с аргументом ***q***. Это, конечно, не вполне в традициях ООП, но что-то общее всё-таки есть. К сожалению, нет той простоты и

ясности, что мы имеем в классах и объектах настоящего ООП и дальше будет только хуже.

Чтобы показать, как можно использовать метод ***d*** в более сложной ситуации рассмотрим пример вычисления длины полигона, заданного точками на плоскости. Для задания полигона используем массив с элементами типа ***Point***. Для удобства создадим новый тип, например, с именем ***Путь***:

```
type Путь []Point
```

То-есть, тип ***Путь*** представляет массивы с элементами типа ***Point***. Затем объявим новый метод с именем ***dlina***, который применяется к объектам типа ***Путь***. Чтобы не повторять код приведу всю программу целиком:

```
package main
import ("fmt"; "math")
func main() {
        perim := Путь{ {1, 1}, {5, 1}, {5, 4}, {1, 1} }
fmt.Println(perim.dlina()) → 12
}
type Point struct{ x, y float64 }
type Путь []Point
func (p Point) d(q Point) float64 {
    return math.Hypot(q.x-p.x, q.y-p.y)
}
func (n Путь) dlina() float64 {
    s := 0.0
    for i := range n {
        if i > 0 { s += n[i-1].d(n[i]) }
    }
    return s
}
```

Не нарушая правила, приёмник метода ***dlina*** мы обозначили по первой букве типа ***Путь***, то-есть русской ***n***. Полигон в примере представлен вершинами треугольника, поэтому заданный массив точек обозначен ***perim***. Метод ***dlina*** вызывается для объекта ***n***, представленного массивом ***perim***, и не имеет аргументов, поэтому скобки пустые. Как видим, метод ***d*** использован при создании другого метода ***dlina***. При этом метод ***dlina*** отличается от метода ***d*** тем, что он вызывается для объекта типа ***Путь*** (в точечной нотации), а

аргументов в скобках справа не имеет. В каком-то смысле можно считать, что метод ***d*** бинарный, а метод ***dlina*** унарный.

Естественно, возникает вопрос — можно ли создавать методы для любого типа. Попробуем такой вариант:

```
package main
import ("fmt")
func main() {
        n := 5
        r := n.f()
        fmt.Println(r)
}
func (n int) f() int {
        return n * n    - здесь будет сообщение об ошибке
}
```

Здесь мы попытались создать метод ***f*** для типа ***int***. Однако, компилятор протестует — для базовых типов создавать методы нельзя. А вот такой простой приём позволяет обойти этот запрет:

```
package main
import ("fmt")
func main() {
        var x N = 5
        r := x.f()
        fmt.Println(r) → 25
}
type N int
func (n N) f() N {
        return n * n
}
```

Тут мы ввели пользовательский тип ***N***, представляющий базовый тип ***int*** и теперь метод ***f*** работает. В любой пользовательский тип можно добавить один или более методов.

Пользовательские типы могут преобразовываться в базовые, например, так: ***y := int(x).*** Также легко можно делать и обратное преобразование. Эти преобразования не затратны, поскольку выполняются на этапе компиляции. Типы, полученные непосредственно из базовых, могут использоваться на тех же правах, что и базовые. Так, в примере выше мы для пользовательского типа ***N*** применили операцию умножения.

6.2 Методы с указателем в роли получателя

Вызов метода, как и функции, создаёт копию значения аргумента. Иногда требуется обновить значение, а не создавать копию, например, если значение занимает много места в памяти — большой массив или текст. Как мы уже знаем, в этих случаях надо использовать указатели на переменные. Чтобы обновить переменную приёмника в методе, её надо присоединить к типу указателя. Для типа ***Point*** типом указателя будет ****Point***. Посмотрим, как это надо делать на примере:

```
package main
import ("fmt")
func main() {
        r := &Point{1, 2}
        r.f(2)
        fmt.Println(*r) → { 2 4 }
}
type Point struct{ x, y float64 }
func (p *Point) f(α float64) {
        p.x *= α
        p.y *= α
}
```

Значит, приёмник определяется так: (***p *Point)***. То-есть, вместо типа используем указатель типа. Вместо значения (экземпляра) создаём указатель на структуру: ***r := &Point{1, 2}***. И тогда вызов метода для этого указателя ***r.f(2)*** приводит к обновлению полей структуры ***Point***, что мы и видим при выводе результата; естественно, что оператору печати передаётся указатель на значение ****r***. Как видим, как и в случае с переменными, сам экземпляр (значение) остаётся безымянным, работаем только с указателем на него. Впрочем, если это зачем-то надо, то имя экземпляра можно и ввести, например, так:

```
p := Point{1, 2}
pr := &p
pr.f(2)
fmt.Println(p) → { 2, 4 }
```

Или так:

```
p := Point{1, 2}
(&p).f(2)
fmt.Println(p) → { 2, 4 }
```

Если в приёмнике метода указатель, то при вызове метода можно использовать не только указатель, но и саму переменную. При этом компилятор автоматически «разыменует» переменную, то-есть использует вместо неё указатель. Например, можно программировать и так:

```
package main
import ("fmt")
func main() {
        r := Point{1, 2}
        r.f(2)
        fmt.Println(r)  → {2, 4}
}
type Point struct{ x, y float64 }
func (p *Point) f(α float64) {
        p.x *= α
        p.y *= α
}
```

Здесь указатель только в приёмнике, а при вызове метода использовано значение. Тем не менее, метод работает, как с указателем — поля структуры изменились.

6.3 Интерфейсы

Интерфейсы определяют тип (сигнатуру) методов и их самих можно считать типами. Интерфейсы это абстракции и невозможно создавать экземпляры интерфейсов. Но зато можно создавать переменные с типами интерфейсов. Этим переменным можно присваивать значения любого конкретного типа, обладающего методами интерфейса. Для начала всё это кажется запутанным и надо разбираться на примерах.

Создадим интерфейс:

```
type Exchanger interface {
        M()
}
```

Значит, интерфейс, как и тип, создаётся со служебным словом ***type***. Далее следует название интерфейса. Это произвольное слово, но разработчики Go рекомендуют применять слова с окончанием на «er», что похоже на название профессии. После идёт служебное слово ***interface*** и блок с телом интерфейса, содержащим методы.

В интерфейсе ***Exchanger*** присутствует только один метод ***M()***, который ничего не принимает и ничего не возвращает, то-есть ничего не имеет, кроме имени. Сам по себе такой интерфейс бесполезен, чтобы получить от него какую-то пользу, надо создать пользовательский тип с методами, определяемыми интерфейсом. Создадим, например, такой тип в виде структуры:

type Pair struct{ α, β string }

Структура ***Pair*** имеет два поля типа ***string*** и пока никак с интерфейсом не связана. Связь появится, если мы прикрепим к типу ***Pair*** метод интерфейса ***Exchanger***, например, так:

```
func (pa *Pair) M() {
        pa.α, pa.β = pa.β, pa.α
}
```

При этом мы здесь определили и функциональность метода ***M()*** - он просто меняет местами поля в структуре. В приёмнике мы использовали указатель на тип, а это значит, что поля будут переставляться «на месте», то-есть в экземпляре типа ***Pair***.

Типов, связанных с интерфейсом ***Exchanger*** через метод ***M(),*** можно создавать сколько угодно. Например:

```
type Point [2]int
func (po *Point) M() { po[0], po[1] = po[1], po[0] }
```

Тип ***Point*** представлен массивом с двумя элементами типа ***int***, а метод ***M()*** обменивает индексы у этих элементов.

Типы ***Pair*** и ***Point*** совершенно разные, но поскольку они удовлетворяют требованиям одного и того же интерфейса, то можно создавать экземпляры этих типов и передавать их функциям, принимающим значения типа ***Exchanger***. Интересно, что Go не требует никаких дополнительных мер по организации связи между типами и интерфейсом, достаточно одного только использования типами методов интерфейса.

Создадим ещё функцию, определяющую формат вывода результатов нашего примера:

```
func (pa Pair) String() string {
        return fmt.Sprintf("%q+%q", pa.α, pa.β)
}
```

На самом деле это метод интерфейса ***fmt.Stringer*** из стандартной библиотеки и при передаче ему экземпляров типа ***Pair*** (а значит и всех экземпляров типа ***Exchanger***) поля будут выводиться в кавычках,

что обеспечивается форматом ***%q,*** и со знаком + между ними. Функция ***Sprintf*** позволяет создавать строки нужного формата для последующего вывода их с помощью функции ***Print***.

Наконец, создадим ещё функцию ***ex***, принимающую произвольное количество объектов типа ***Exchanger*** и применяющую к ним в цикле метод ***M()***:

```
func ex(s ...Exchanger) {
	for _, exchanger := range s {
		exchanger.M()
	}
}
```

В целом программа с созданием нескольких значений типа ***Exchanger***, с вызовами метода ***M()*** и пользовательской функции ***ex()***, принимающей значения типа ***Exchanger***, имеет вид:

```
package main
import ("fmt")
func main() {
	a := Pair{"Люся", "Наташа"}
	b := Pair{"Петя", "Вова"}
	p := Point{5, -3}
	fmt.Println("Сначала: ", a, b, p)
	a.M() // Интерпретируется как: (&a).M()
	b.M() // Интерпретируется как: (&b).M()
	p.M() // Интерпретируется как: (&p).M()
	fmt.Println("После #1:", a, b, p)
	ex(&a, &b, &p)
	fmt.Println("После #2:", a, b, p)
}
type Exchanger interface {
	M()
}
type Pair struct{ α, β string }
func (pa *Pair) M() {
	pa.α, pa.β = pa.β, pa.α
}
type Point [2]int
func (po *Point) M() { po[0], po[1] = po[1], po[0] }
func (pa Pair) String() string {
```

```
        return fmt.Sprintf("%q+%q", pa.α, pa.β)
}
func ex(s ...Exchanger) {
        for _, exchanger := range s {
                exchanger.M()
        }
}
```

И таков будет результат:

```
Сначала:  "Люся"+"Наташа" "Петя"+"Вова" [5 -3]
После #1: "Наташа"+"Люся" "Вова"+"Петя" [-3 5]
После #2: "Люся"+"Наташа" "Петя"+"Вова" [5 -3]
```

Механизм интерфейсов в Go без сомнения весьма эффективен. В частности, на базе интерфейсов в языке реализуется что-то вроде наследования, столь популярного в языках ООП. Интерфейсы же обеспечивают и своеобразный полиморфизм. Правда, для начала всё это выглядит несколько громоздким и не достаточно систематизированным. Но, видимо, дело в приобретении опыта на практике.

6.4 Ещё о структурах

Начнём с примера:

```
package main
import ("fmt")
func main() {
        s := [][2]int{{4, 6}, {}, {-7, 11}, {15, 17}, {14, -8}}
        f(s)
}
func f(m [][2]int) {
        for _, x := range m {
                fmt.Printf("(%d, %d) ", x[0], x[1])
        }
}
```

Функция ***f*** просто выводит на экран пары двумерного массива, заключая их в круглые скобки. Кстати, значение ***{}*** в данном контексте трактуется, как ***{0, 0}***. Результат имеет вид:

```
(4, 6) (0, 0) (-7, 11) (15, 17) (14, -8)
```

То же самое можно получить, используя анонимную структуру, что во многих случаях более просто и удобно:

```
package main
import ("fmt")
func main() {
        s := []struct{ x, y int }{{4, 6}, {}, {-7, 11}, {15, 17}, {14, -8}}
        f(s)
}
func f(m []struct{ x, y int })  {
        for _, p := range m {
                fmt.Printf("(%d, %d) ", p.x, p.y)
        }
}
```

Здесь массив ***m*** имеет элементы в виде экземпляров структуры с полями ***x*** и ***y***. Хотя структура не имеет имени, в цикле ***for*** мы вызываем её экземпляры, давая им имя ***p*** и это позволяет обращаться к полям ***x*** и ***y.***

Структуры могут иметь поля двух типов: агрегированные и встроенные. Посмотрим на примере:

```
type Point struct {
        color.Color →  Анонимное (безымянное) поле (встраивание)
        x, y int →  Именованные поля (агрегирование)
}
```

Здесь ***color.Color*** – это имя типа из пакета ***image/color***, а ***x*** и ***y*** – имена полей типа ***int***. Анонимные поля вроде ***color.Color*** называются встраиваемыми, а поля с именем — агрегируемыми.

Структуры, как и интерфейсы или другие типы, могут включаться в другие структуры на правах полей. Если поле содержит только название структуры, оно будет встроенным, а если структуре дать имя — агрегированным. Поля встроенного поля обычно доступны непосредственно, с помощью оператора точки ***(.)***, без упоминания имени типа. Впрочем, лучше посмотреть на примере:

```
package main
import ("fmt")
func main() {
        Петров := Person{"Иван", "Марья", []string{"Коля", "Валя",
"Лена"}}
        fmt.Println(Петров.Жена) → Марья
        Бухгалтер := Clerk{Петров, 1980, "Москва"}
        fmt.Println(Бухгалтер.Дети) → [Коля Валя Лена]
```

```
        Бухгалтер.Дети = append(Бухгалтер.Дети, "Юра")
        fmt.Println(Бухгалтер.Дети) → [Коля Валя Лена Юра]
        fmt.Println(Петров.Дети) → [Коля Валя Лена]
}
type Person struct {
        Имя string
        Жена string
        Дети []string
}
type Clerk struct {
        Person
        ГодРождения int
        Адрес string
}
```

Структура ***Person*** встроена в структуру ***Clerk*** в качестве её поля без указания имени. Создав экземпляр структуры ***Clerk*** под названием ***Бухгалтер*** имеем возможность доступа и по чтению и по записи, например, к полю ***Дети***, не упоминая имя типа ***Person***, хотя указание типа допустимо, если возможна неоднозначность. При изменении поля ***Бухгалтер.Дети*** создаётся копия, а исходное поле ***Петров.Дети*** не изменилось. Для того, чтобы это изменение произошло, надо использовать указатель.

При встраивании в структуру полей, имеющих методы, имеются особенности использования этих методов. Посмотрим на примере:

```
package main
import ("fmt")
func main() {
        v := Task{}
        fmt.Println(v.IsZero(), v.f(), v) → true 0 {[] 0}
        v.Add("One")
        v.Add("Two")
        fmt.Println(v.IsZero(), v.f(), v) → false 2 {[One Two] 2}
}
type Task struct {
        s []string
        Count
}
func (t *Task) Add(x string) {
```

```
        t.s = append(t.s, x)
        t.Inc()
}
func (t *Task) f() int {
        return int(t.Count)
}
type Count int
func (c *Count) Inc() { *c++ }
func (c Count) IsZero() bool { return c == 0 }
```

Структура ***Task*** имеет агрегированное поле ***s []string*** — массив с элементами типа ***string*** и встроенное (анонимное) поле типа ***Count.*** Пользовательский тип ***Count*** на базовом типе ***int*** объявлен в конце программы и имеет два метода: ***Inc()*** (принимает указатель) и ***IsZero().***

Метод ***Add*** выполняет два действия. В строке ***t.s = append(t.s, x)*** к массиву ***s*** добавляется значение ***x*** с изменением массива ***s*** на месте, поскольку в приёмнике метода — указатель. Тут всё понятно. А вот в строке ***t.Inc()*** к экземпляру структуры ***Task*** применяется метод анонимного поля типа ***Count***. На самом деле здесь метод ***Inc*** работает со своим собственным аргументом ***c***. Причём начальное значение ***c*** инициировано нулём при объявлении метода ***Inc***, а при вызове этого метода значение ***c*** увеличивается на ***1***. Фактически при вызове ***Tasks.Inc()*** и других методов типа ***Count*** относительно экземпляра типа ***Tasks*** этим методам передается значение типа ***Count*** (или ****Count***), а не ***Tasks***. Поскольку у нас в приёмнике указатель ****Count***, то переменная ***c*** тоже изменяется на месте. Подобным же образом работает и метод ***IsZero***, только тут в приёмнике не указатель, а сам экземпляр типа ***Count***.

И совсем уж интересно работает функция ***f***. В выражении ***int(t.Count)*** метод ***Inc*** даже и не упомянут, как и переменная ***c***. Тем не менее такой вызов работает, как ***c.Inc()***. Переменная ***c*** тут появилась по той же причине, что и в методе ***Add***, а о том, что к ней надо применить именно метод ***Inc*** компилятор узнаёт, видимо, по указанному типу ***int***. В общем, как говорится, с таким языком не соскучишься. Выведенные результаты показывают, что всё работает именно так, как я описал на словах.

7. Пакеты, файлы, область видимости

7.1 Пакеты и файлы

Пакеты в Go служат тем же целям, что и библиотеки или модули в других языках программирования, поддерживая модульность, инкапсуляцию, раздельную компиляцию и повторное использование. Исходный текст пакета располагается в одном или нескольких файлах ***.go***. Покажем на примере, как это надо делать. Используем снова уже знакомую программу перевода температуры из одной шкалы на другую. Для учебных целей организуем эту программу по-другому, создав пакет, располагающийся в двух файлах и головную программу импортирующую этот пакет. Импортируемый пакет назовём ***tempconv*** и две его составные части запишем в файлы: ***pacet.go***:

```
package tempconv
import ("fmt")
type Celsius float64
type Fahrenheit float64
const (
        AbsoluteZeroC Celsius = -273.15
        FreezingC Celsius = 0
        BoilingC Celsius = 100
)
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
func (f Fahrenheit) String() string { return fmt.Sprintf("%g°F", f) }
```

и в ***conv.go***:

```
package tempconv
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }
func FToC(f Fahrenheit) Celsius { return Celsius((f-32)* 5/9) }
```

В первом файле у нас расположены объявления типов, констант и функция ***String()***, формирующая строки для вывода в операторе Print - раньше мы с этим уже познакомились. Во втором файле — объявления двух функций, выполняющих преобразование температуры.

Для того, чтобы этим пакетом можно было воспользоваться, надо в каталоге ***src*** создать папку с названием, совпадающим с названием пакета, то-есть ***tempconv*** и поместить в эту папку файлы пакета ***pacet.go*** и ***conv.go***. Головной пакет ***main*** расположим в файле

rabfc.go, который может находиться где угодно, у меня, например, по адресу ***C:\User\Boris\Documents\Go***:

```
package main
import ("fmt"; "tempconv")
func main() {
        fmt.Printf("Бррррр! %v\n", tempconv.AbsoluteZeroC)
        fmt.Println(tempconv.CToF(tempconv.BoilingC))
}
```

Теперь надо перейти в директорию ***C:\User\Boris\Documents\Go*** и выполнить, как обычно, команду:

go run rabfc.go

Получим результат:

Бррррр! -273.15°C
212°F

Рядом с файлом ***rabfc.go*** появится рабочий файл ***rabfc.exe***, который в дальнейшем можно использовать без компиляции.

Файлы импортируемого пакета можно расположить и в другом месте (не в папке ***src***), но тогда надо в «переменные среды» переменную ***GOPATH*** приравнять адресу папки, содержащей папку t***empconv.*** Есть также возможность указывать полный путь для импортируемого пакета непосредственно в команде ***import***, и тут возможны всякие варианты, но надо разбираться с этим самостоятельно.

Обращаю внимание на то, что никакой предварительной компиляции файлов пакета не требуется, всё будет выполнено командой ***go run***. Если посмотрите содержимое папки ***src***, то увидите там много стандартных пакетов, которыми можно пользоваться точно также, как и созданным нами пакетом. Все эти пакеты представлены файлами с расширением ***go***. Один пакет может располагаться, как видим, в двух и более файлах с разными названиями и этот пакет будет импортироваться точно также, как если бы он был расположен в одном файле.

7.2 Область видимости

Для ограниченной области видимости в Go используется понятие блок. Синтаксически — это некая часть программного кода, заключённого в фигурные скобки. Например, это может быть тело функции или цикла. Имя, объявленное внутри синтаксического блока,

не видимо вне блока. Кроме того, областями видимости могут быть и другие группы объявлений, которые не охватываются фигурными скобками в исходном тексте явно. Иногда их называют лексическими блоками. Самостоятельным лексическим блоком является весь исходный текст программы, именуемый всеобщим блоком (universe block), блок для каждого пакета, для каждого файла, для каждой инструкции ***for, if*** и ***switch***, для каждого ***case*** в конструкции ***switch*** или ***select***.

Любой пользовательский объект, объявленный в лексическом блоке, видим только внутри этого блока. Объявления встроенных типов, функций и констант наподобие ***int, len*** и ***true*** находятся во всеобщем блоке, и обратиться к ним можно на протяжении всей программы. К объявлениям вне любой функции, т.е. на уровне пакета, можно обратиться из любого файла в том же самом пакете. Импортированные пакеты видимы на уровне файлов, так что к ним можно обращаться из того же файла (но не из другого файла в том же пакете) без отдельной директивы ***import***.

Программа может содержать несколько объявлений одного и того же имени при условии, что все объявления находятся в различных лексических блоках. Если внутри блока объявляется переменная с тем же именем, что и переменная вне блока, то внешняя переменная не видима (затеняется локальной переменной) внутри блока и недоступна на время выполнения блока.

В языке Go имеется ещё один механизм управления видимостью: не экспортируемыми (то есть видимыми только внутри пакета, где они объявлены) являются идентификаторы, начинающиеся с буквы в нижнем регистре, а экспортируемыми (видимыми в любом пакете, импортирующем данный пакет) являются идентификаторы, начинающиеся с буквы в верхнем регистре. Иногда это правило позволяет просто решить вопрос управления видимостью в пакетах.

Рассмотрим пример, иллюстрирующий применение функции ***init***, область видимости и применение некоторых стандартных функций:

```
package main
import ("fmt"; "os"; "log")
func main() {
        fmt.Println(cwd)
}
var cwd string
```

```
func init() {
	var err error
	cwd, err = os.Getwd()
	if err != nil {
		log.Fatalf("Ошибка os.Getwd failed: %v", err)
	}
}
```

Функция ***Getwd()*** из пакета ***os*** возвращает два значения: каталог, в котором находится файл с нашей программой (рабочий каталог), и сообщение об ошибке, если каталог не может быть прочитан. Название каталога присваивается переменной ***cwd*** типа ***Str***, а сообщение об ошибке — переменной ***err*** типа ***error***. Обе эти переменные объявлены на уровне пакета. Если значение ***err*** не равно ***nil***, функция ***Fatalf*** из пакета ***log*** выводит это значение с пояснительным текстом. Функция ***main()*** выводит на экран содержимое переменной ***cwd***, хотя и не обращается явно к функции ***init()***, формирующей это содержимое. Функция ***init()*** расположена после функции ***main()***, но она выполняется раньше и автоматически, без вызова.

7.3 Ввод-вывод

На примерах мы довольно хорошо познакомились с программированием вывода на экран. Был также рассмотрен пример программы по вводу с клавиатуры. Займёмся теперь проблемой чтения из файла и вывода в файл. Составим программу ввода, например, из файла с именем ***rab1.go***:

```
package main
import ("fmt"; "os"; "bufio")
func main() {
	fname := "C://Users//Boris//Documents//Go//rab1.go"
	fil, _ = os.Open(fname)
	defer fil.Close()
	r := bufio.NewReader(fil)
	var s string
	s, _ = r.ReadString('@')
	fmt.Println(s)
}
```

Проанализируем все действия по-порядку. Переменную ***fname***

инициируем полным именем файла в виде строки. Напоминаю, что двойной слеш использован для экранизации, так как одиночный слеш воспринимается, как управляющий символ. Затем объявляем вспомогательную переменную ***fil***, инициируя её неким значением из пакета ***os***. Если здесь вывести на экран тип переменной fil с помощью команды ***fmt.Printf("%T\n", fil)*** (напоминаю, что мы этот приём уже использовали) то получим значение ****os.File***. Значит, это ссылка на какой-то объект из пакета ***os***. Для нас имеет значение только то, что следующая строка ***fil, _ = os.Open(fname)*** открывает файл для чтения. На месте знака ***(_)*** можно поставить переменную типа ***error***, например ***e error***, и тогда переменная ***e*** получит значение ***null***, если файл нормально откроется, или сообщение об ошибке, если что-то не так (например, такой файл не существует).

Наконец, строка ***r := bufio.NewReader(fil)*** читает информацию из файла, инициируя переменную ***r***, при помощи функции ***NewReader*** из пакета ***bufio***. Однако, радоваться ещё рано - информация пока имеет внутреннее представление и для нас недоступна. Для преобразования содержимого переменной ***r*** в текстовое представление применяется встроенная функция ***ReadString***, которая преобразует кусок текста до первого встреченного знака, указанного, как аргумент функции. Если указать ***r.ReadString('\n')***, то получим только первую строку файла, которой и будет инициирована строковая переменная ***s***. Обращаю внимание на то, что знак задаётся в одиночных кавычках. Если же указать какой-то знак, заведомо отсутствующий в файле (я применил **@**), переменная ***s*** будет равна всему тексту файла в виде одной строковой переменной, которую теперь можно, например, вывести на экран. Функция ***ReadString*** также возвращает информацию об ошибке, которую можно передать переменной типа ***error***, если указать её на месте знака ***(_)***. Наша же программа ошибок не анализирует.

Строка ***defer fil.Close()*** закрывает файл. Инструкция ***defe***r принадлежит механизму сборки мусора. Работой инструкции ***defer*** управляет компилятор, главное, что файл будет закрыт и буфер освобождён.

И опять я должен сказать, что вся эта канитель — далеко не шедевр. Возможно, что создатели Go технику обмена информацией намеренно не довели до высокого уровня, сохраняя известную гибкость и свободу действий, что будет полезно, если в дальнейшем

на базе Go будут создаваться новые языки программирования более высокого уровня.

Приведу ещё вариант программы, когда файл читается построчно в цикле:

```
package main
import ("fmt"; "os"; "bufio"; "io")
func main() {
	fname := "C://Users//Boris//Documents//Go//rab1.go"
	fil := os.Stdin
	defer fil.Close()
	fil, _ = os.Open(fname)
	r := bufio.NewReader(fil)
	var s string
	fl := false
	var e error
	for !fl {
		s, e = r.ReadString('\n')
		fmt.Print(s)
		if e == io.EOF {fl = true}
	}
}
```

Для управления выходом из цикла здесь использована переменная ***fl*** типа ***bool***, предварительно инициированная значением типа ***false***. Когда будет достигнут конец файла, функция ***ReadString*** вернёт константу ***EOF*** из пакета ***io*** и тогда переменная ***fl*** получит значение ***true*** по которому и произойдёт выход из цикла. Напоминаю, что цикл вида ***for !fl { ... }*** будет работать до тех пор, пока выражение ***!fl*** не получит значение ***false***.

В нашей программе мы только выводим строки файла на экран, но нетрудно запрограммировать какие-то действия над строками. Например, их можно загрузить в массив для последующего использования, или обрабатывать их здесь же. Если функции ***ReadString*** передать аргумент ***(' ')***, то текст будет разбит по пробелам, то-есть на отдельные слова. Можно даже разбить текст по каким-то буквам и тому подобное.

Запись в файл выполняется аналогично. Например, запишем в файл заданную строку:

```
package main
```

```
import ("os"; "bufio")
func main() {
        fname := "C://Users//Boris//Documents//Go//rab1.go"
        fil := os.Stdout
        defer fil.Close()
        fil, _ = os.Create(fname)
        r := bufio.NewWriter(fil)
        s := "Hello, World!\n"
        r.WriteString(s)
        defer r.Flush()
}
```

Вместо ***Stdin, Open*** и ***NewReader*** теперь используются ***Stdout***, ***Create*** и ***NewWrite***r из тех же пакетов. Запись в файл выполняется в строке r***.WriteString(s)***, где переменная ***s*** типа ***string*** передаётся встроенной функции ***WriteString*** в качестве аргумента. Инструкция ***defer r.Flush()*** выталкивает содержимое буфера в файл. Без этой строки часть информации из конца файла может быть утеряна (останется в буфере). Обязательна и инструкция закрытия файла ***defer fil.Close()***, без которой файл не будет закрыт и запись в него не будет выполнена.

Если запись выполняется в несуществующий файл, этот файл будет создан заново. Если же запись будет в существующий файл, то записанная в него ранее информация стирается. В Go есть также функция ***os.OpenFile()***, позволяющая управлять режимами и разрешениями, с помощью которой можно запрограммировать запись в файл без его обнуления.

Не трудно организовать цикл, аналогичный тому, что мы рассмотрели для чтения из файла, в котором можно запрограммировать вывод в файл строк, слов, или даже букв последовательно. Видимо, наиболее часто приходится работать выполняется с двумя файлами параллельно — из одного файла информация считывается, а в другой записывается после требуемой обработки.

7.4 Немного о параллельном программировании

В Go имеются простые средства, позволяющие параллельное программирование. Техника базируется на идее обмена данными между программами, а не на совместном их использовании. Есть

возможность создания каналов, предназначенных как для односторонней, так и двухсторонней передачи данных. В качестве параллельных программ используются так называемые ***go***-подпрограммы, которые очень легко создать и которые фактически представляют «потоки выполнения».

Сначала продемонстрируем работу ***go***-подпрограммы на простом примере без передачи данных, когда одновременно вычисляется большое число Фибоначчи и выводится на экран анимация в виде вращения.

```
package main
import("fmt"; "time")
func main() {
        go spinner(100 * time.Millisecond)
        const n = 43
        res := fib(n)
        fmt.Printf("\rFibonacci(%d) = %d\n", n, res)
}
func spinner(delay time.Duration) {
        for {
                for _, r := range `-\|/` {
                        fmt.Printf("\r%c", r)
                        time.Sleep(delay)
                }
        }
}
func fib(x int) int {
        if x < 2 { return x }
        return fib(x-1) + fib(x-2)
}
```

Функция ***main*** вызывает ***go***-подпрограмму, которой в качестве аргумента передаётся функция ***spinner***. Работа ***go***-подпрограммы заключается в выполнении бесконечного цикла, в котором всё время повторяется вызов функции ***spinner***. Эта функция выполняет вывод на экран через заданный промежуток времени (100 * time.Millisecond) в одной и той же позиции последовательно знаки ****, | и /, имитируя таким образом вращение. Больше ничего ***go***-подпрограмма не делает. Затем вызывается функция ***fib***, которая вычисляет число Фибоначчи для аргумента ***43***, затрачивая на это заметный промежуток времени.

Как только происходит выход из функции ***fib*** с выдачей результата, программа заканчивает работу, одновременно останавливается и ***go***-подпрограмма. Таким образом, вращение и вычисление числа Фибоначчи происходят параллельно.

Теперь посмотрим на более содержательный пример, когда имеет место обмен информацией между параллельными программами. Рассмотрим простейшую задачу преобразования полярных координат в декартовы. В параллельной программе будем выполнять само преобразование, а всё остальное — в головной программе.

```
package main
import ("bufio"; "fmt"; "math"; "os")
type ρθ struct { ρ float64; θ float64 }
type xy struct { x float64; y float64 }
func main() {
	вопрос := make(chan ρθ)
	defer close(вопрос)
	ответ := f(вопрос)
	defer close(ответ)
	φ(вопрос, ответ)
}
func f(вопрос chan ρθ) chan xy {
	rab := make(chan xy)
	go func() {
		for {
			ρθ := <-вопрос
			θ := ρθ.θ * math.Pi / 180.0
			x := ρθ.ρ * math.Cos(θ)
			y := ρθ.ρ * math.Sin(θ)
			rab <- xy{x, y}
		}
	}()
return rab
}
const res = "ρ = %.02f θ = %.02f° ® x=%.02f y=%.02f\n"
func φ(вопрос chan ρθ, ответ chan xy) {
	r := bufio.NewReader(os.Stdin)
	fmt.Println("Введите радиус и угол в градусах, например: 12.5
90, " + "или (Ctrl+Z, Enter  для выхода)")
```

```
    for {
        fmt.Printf("Радиус и угол: ")
        s, err := r.ReadString('\n')
        if err != nil { break }
        var ρ, θ float64
        if _, err := fmt.Sscanf(s, "%f %f", &ρ, &θ); err != nil {
            fmt.Fprintln(os.Stderr, "ошибка ввода")
            continue
        }
        вопрос <- ρθ{ρ, θ}
        xy := <-ответ
        fmt.Printf(res, ρ, θ, xy.x, xy.y)
    }
    fmt.Println()
}
```

Опять проанализируем все строки по-порядку с соответствующими комментариями. Итак, первые две строки

type ρθ struct { ρ float64; θ float64 }
type xy struct { x float64; y float64 }

создают две структуры с элементами типа ***float64***. Поля структуры ***ρθ*** представляют исходные данные (радиус ***ρ*** и угол ***θ*** в полярных координатах), а поля структуры ***xy*** — результат (декартовы координаты точки ***x*** и ***y***).

Функция ***main*** имеет очень лаконичный код — здесь только вызовы встроенных и пользовательских функций. Строка

вопрос := make(chan ρθ)

создаёт канал с помощью встроенной функции ***make*** которой передаётся синтаксическая конструкция ***chan ρθ*** (***chan*** — ключевое слово) и сохраняет этот канал в переменной ***вопрос***. Функции ***make*** можно через запятую передать ещё один аргумент — целое число, определяющее размер буфера в строках для хранения передаваемых данных. По умолчанию этот аргумент принимается равным нулю, а это означает, что канал может использоваться только тогда, когда на другом конце канала ожидается приём данных. Мы назвали этот канал ***вопрос*** и предназначен он для отправки структуры ***ρθ*** для обработки в параллельной программе. Строка

defer close(вопрос)

предназначена для закрытия канала ***вопрос*** и освобождения памяти после того, как канал станет не нужен. Далее строка

ответ := f(вопрос)

вызывает пользовательскую функцию ***f***, передавая ей канал вопрос. Функция ***f*** возвращает канал ответ (типа ***chan xy***), то-есть с помощью этого канала передаётся в обратном направлении структура ***xy***. Инструкция d***efer close(ответ)*** закрывает канал ***ответ***. Строка

φ(вопрос, ответ)

вызывает пользовательскую функцию ***φ***, принимающую оба канала ***вопрос*** и ***ответ***.

Функция ***f*** выполняет преобразование координат, а функция ***φ*** — реализует взаимодействие с пользователем. Проанализируем теперь работу этих функций.

Объявление функции ***f*** имеет вид:

func f(вопрос chan ρθ) chan xy { }

Значит, функция ***f*** принимает в качестве аргумента канал вопрос типа ***chan ρθ*** и возвращает канал типа ***chan xy***. Сначала строка

r***ab := make(chan xy)***

создаёт канал ***rab*** типа ***chan xy***, предназначенный для передачи результата — структуры ***xy***.

Дальнейшие действия выглядят довольно туманными. За созданием канала следует инструкция ***go***. Этой инструкции передается вызов анонимной функции, которая будет выполняться в отдельной, асинхронной ***go***-подпрограмме. За инструкцией ***go*** следует инструкция ***return***, возвращающая канал ***rab*** вызывающей программе. В инструкции ***go*** создается (и вызывается) анонимная функция, которая выполняет бесконечный цикл, ожидающий (не блокирующий выполнение других ***go***-подпрограмм и функции ***main***, в которой ***go***-подпрограмма была запущена) появления запросов в канале ***вопрос***, в данном случае значений типа ***ρθ struct***. Когда в канале появятся полярные координаты, анонимная функция вычислит соответствующие им декартовы координаты (с помощью пакета ***math*** из стандартной библиотеки) и отправит ответ типа ***xy struct*** в канал ***ответ***. Выражение ***xy{x, y}*** инициирует поля структуры ***xy*** только что вычисленными значениями декартовых координат.

Для передачи данных по каналам используется специальный оператор ***(<-)***. Унарный вариант этого оператора, извлекает полярные

координаты из канала ***вопрос***, инициируя поля стркутуры ***ρθ***. А двухместная версия оператора ***(<-)*** отправляет структуру ***xy*** в канал rab и на этом работа ***go***-подпрограммы заканчивается. Обращаю внимание, что после закрывающей фигурной скобки для тела анонимной функции надо поставить пустые круглые скобки. Объяснить можно тем, что скобки эти должны свидетельствовать, что ***go*** является функцией.

В функции ***φ*** нет ничего сложного. В качестве аргументов она получает каналы ***вопрос*** и ***ответ***. Для инициирования полей структуры ρθ запрограммирован ввод исходных данных с клавиатуры — эта операция нам уже знакома. С помощью метода ***fmt.Sscanf*** выполняется контроль ввода по условию, что должно быть введено два значения типа ***float64***. Использование ссылок ***&***ρ, ***&θ*** обусловлено требованиями самого метода ***Sscanf***, для того, чтобы понять его работу, надо изучить его отдельно. Ввод выполняется в бесконечном цикле для многократного повторения ввода, а выход из цикла задан по условию ввода ***Ctrl+Z***.

Очевидно, что в нашем очень простом случае мы ничего не получаем от применения параллельного программирования. Положительный эффект может быть только при соблюдении следующих условий:

- Решаемая задача должна быть большой по объёму вычислений.
- В программе должны быть части, вычисления в которых можно выполнять по отдельности, то-есть без взаимной связи.
- Иногда просто надо выполнить какие-то процессы параллельно, как, например, в примере с числами Фибоначчи.

Тогда самостоятельные фрагменты программы можно реализовать в виде отдельных go-подпрограмм, которые будут выполняться параллельно с основной вызывающей программой, сокращая затраты времени или воспроизводя нужные эффекты.

Я думаю, что для первого знакомства с языком Go приведённых сведений достаточно. Конечно, многие темы я лишь только затронул. Например, совершенно недостаточно места уделено таким очень важным для Go структурам, как интерфейсы. Параллельное программирование также требует более детального рассмотрения. Почти совсем не рассматривались способы обработки текста и регулярные выражения в частности. Впрочем, эта техника не

слишком трудная для понимания, к тому же она мало отличается от соответствующих методов из других языков. Средства для веб-программирования и графические интерфейсы на мой взгляд вообще следует рассматривать отдельно. Лучше всего было бы этой тематике посвятить отдельную книгу.