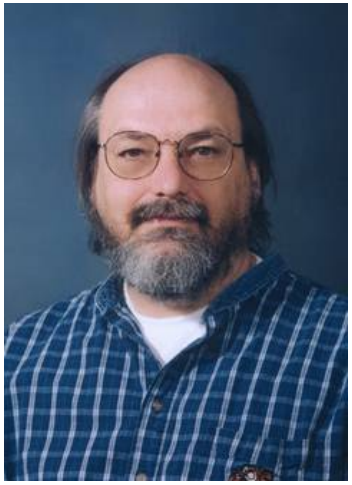
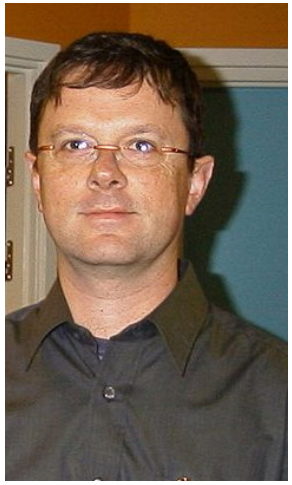
§1. История и основные особенности языка Go

21 сентября 2007 года – начало проектирования нового языка (Кен Томпсон, Роб Пайк и Роберт Гризмер);
январь 2008 года – начало разработки прототипа компилятора (Кен Томпсон), компилятор порождал код на языке C;
май 2008 – начало разработки front-end для gcc (Ян Тейлор);
конец 2008 – к проекту присоединился Расс Кокс;
11 ноября 2009 – первый релиз.



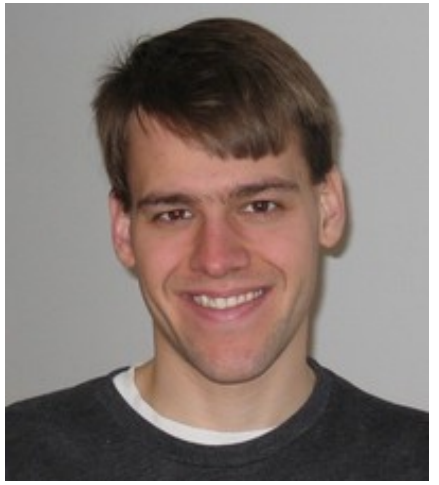
Кен Томпсон



Роб Пайк

Роберт Гризмер

Ян Тейлор



Расс Кокс

Go – язык системного программирования нового поколения.

Основные особенности:

1. простая система типов, не характерная для современных объектно-ориентированных языков (отсутствие наследования, статический duck-typing);

2. поддержка замыканий (функции могут быть вложены друг в друга, вложенная функция имеет доступ к локальным переменным объемлющей функции, указатели на функции – полноправные данные);

3. поддержка параллельно выполняемых сопрограмм, взаимодействующих через типизированные каналы (актуально для современных многоядерных процессоров);

4. высокая скорость компиляции благодаря лучшей структурированности исходного кода многофайловых проектов (система пакетов) и простой системе типов;

5. автоматическое управление памятью (сборка мусора);

6. безопасность (отсутствие арифметики указателей, проверки на выход за границы массивов во время выполнения программы);

7. непротиворечивая грамматика (легко написать эффективный синтаксический анализатор);

8. стандартизованная система оформления кода программы (утилита gofmt).

Установка Go под Linux.

Убедиться, что установлены пакеты mercurial, bison, awk.

В корне своего home-каталога выполнить команды

```
hg clone -u weekly https://code.google.com/p/go
cd go/src
./all.bash
```

В конец файла .bashrc добавить:

```
export GOROOT=$HOME/go
export GOBIN=$GOROOT/bin
export GOOS=linux
export GOARCH=amd64
export GOCORES=8
export PATH=$PATH:$GOBIN
```

(Для 32-разрядного Linux GOARCH=386)

Пример. (hello.go)

```
1 package main
2 import "fmt"
3 func main() {
4     fmt.Printf("Hello, World!\n")
5 }
```

Компиляция и запуск:

```
go build hello.go
./hello
```

§2. Лексика языка Go

Текст программы представлен в Unicode – разрешено использовать литеры любого алфавита для представления идентификаторов (имён переменных, функций, типов и т.д.), регистр букв различается (как и в C).

Два вида комментариев: многострочный /* ... */ и однострочный // ...

Целочисленные константы и константы с плавающей точкой записываются так же, как и в языке C.

Поддерживаются комплексные константы (они выглядят как обычная константа с плавающей точкой, за которой следует буква i).

Символьные константы по сравнению с языком C дополнены Escape-последовательностями \u*xxxx* и \U*xxxxxxxx* для записи любого символа Unicode (здесь *x* – 16-ричная цифра).

Строковые константы бывают двух видов: интерпретируемые и неинтерпретируемые.

Интерпретируемые унаследованы из языка C (записываются в двойных кавычках, могут содержать Escape-последовательности, должны располагаться в одной строчке программы).

Неинтерпретируемые записываются в обратных кавычках (клавиша на клавиатуре слева от 1, с буквой Ё :-)) и могут занимать несколько строчек программы. Escape-последовательности в них не учитываются.

Пример:

Интерпретируемые строки:

```
"int main()\n" +
"{\n" +
"    printf(\"Hi!\\n\");\n" +
"    return 0;\n" +
"}\n"
```

Неинтерпретируемая строка:

```
`int main()
{
    printf("Hi!\n");
    return 0;
}`
```

В языке C операторы должны заканчиваться точкой с запятой. В Go, во-первых, точка с запятой *разделяет* операторы (то есть после последнего оператора в блоке она не нужна), а во-вторых, в большинстве случаев точки с запятой можно не ставить, потому что компилятор сам умеет вставлять их в программу по следующему нехитрому правилу:

Правило. Когда программа разбивается на лексемы, точка с запятой автоматически вставляется после каждой непустой строчки, если в конце этой строчки стоит:

1. идентификатор;
2. константа;
3. ключевое слово break, continue, fallthrough или return;
4. спецсимвол $++$, $--$,),] или }

Исходный код:

```
1 package main

3 import "fmt"

5 const (
6   ALPHA = iota
7   BETA
8   GAMMA
9 )

11 func main() {
12   fmt.Printf("%d\n",
13       BETA,
14   )
15 }
```

Добавляемые «;»:

```
1 package main;

3 import "fmt";

5 const (
6   ALPHA = iota;
7   BETA;
8   GAMMA;
9 );

11 func main() {
12   fmt.Printf("%d\n",
13       BETA,
14   );
15 };
```

§3. Объявление переменных и именованных констант

Объявление переменных имеет вид

```
var список_переменных тип = список_значений
```

Пример.

```
var x float32
var p, q, r *int
var a, b, c int = 1, 2, 3
```

Несколько подряд идущих объявлений можно записывать в виде var-блока:

```
var (
    x float32
    p, q, r *int
    a, b, c int = 1, 2, 3
)
```

Тип переменной может быть выведен компилятором автоматически:

```
var x = 0.1
var a, b, c = 'Q', 2, 3.5
```

Внутри тел функций можно использовать сокращённую форму объявления переменных вида

```
список_переменных := список_значений
```

Пример.

```
x := 0.1
a, b, c := 'Q', 2, 3.5
```

Функция в Go может возвращать сразу несколько значений. При этом можно совмещать вызов функции с объявлением переменных, в которые будут записаны возвращаемые ею значения:

```
a, b, c := f()
```

В таком объявлении бывает удобно использовать так называемый *пустой идентификатор*, записываемый как символ подчёркивания:

```
a, _, c := f()
```

Пустой идентификатор реально не представляет новую переменную. То есть мы его используем, если какое-то значение, возвращаемое функцией, нам не нужно.

Объявление именованных констант имеет вид

```
const список_имён тип = список_константных_значений
```

Пример.

```
const PI float32 = 3.14
const A, B int = 10, 20
const x complex64 = 0.25i
```

Несколько подряд идущих объявлений можно записывать в виде const-блока:

```
const (
    PI float32 = 3.14
    A, B int = 10, 20
    x complex64 = 0.25i
)
```

Тип константы может быть выведен компилятором автоматически:

```
const x = 0.1
const u, v = 1.5, 'Q'
```

Внутри const-блока зарезервированный идентификатор iota представляет последовательность возрастающих целочисленных констант. В начале каждого блока iota сбрасывается в 0.

Пример.

```
const (
        a = iota  // a == 0
        b = iota  // b == 1
        c = iota  // c == 2
)
const (
        x = 1 << iota  // x == 1
        y = 1 << iota  // y == 2
        z = 1 << iota  // z == 4
)
```

При использовании iota очень часто возникает ситуация, когда объявления всех констант внутри const-блока совпадают с точностью до имени константы:

```
const (
        A int16 = 'a'+iota
        B int16 = 'a'+iota
        C int16 = 'a'+iota
)
```

В этом случае разрешается ограничиться указанием только имени константы для всех констант, кроме первой:

```
const (
        A int16 = 'a'+iota
        B
        C
)
```

§4. Основные операторы

В Go отсутствуют *операции* присваивания, инкремента и декремента. В место них используются *операторы* присваивания, инкремента и декремента.

Присваивание бывает простое, составное, кортежное и параллельное.

Простое: `a = b`.

Составное: `a += b` (имеет вид *op*=, где *op* – бинарная операция).

Кортежное: `a, b, c = f(x)` (если `f` возвращает несколько значений).

Параллельное: `a, b = b, a` (такой вот swap :-)).

Операторы инкремента и декремента записываются как `a++` и `a--`.

Оператор выбора в общем случае имеет вид

```
if простой_оператор; условие блок else оператор_if_или_блок
```

Блок, также как и в языке C, представляет собой последовательность операторов, заключённую в фигурные скобки. Видимость переменных и констант, объявленных внутри блока, ограничена блоком.

Простой оператор, расположенный сразу после if, чащё всего представляет собой сокращённое объявление переменных и может быть пропущен. Ветка else тоже может отсутствовать.

Пример.

```
if a, b := f(x); a < b {
        return 0
} else {
        fmt.Printf("Hi")
}
```

Условие не заключается в круглые скобки, и из-за этого тело оператора выбора должно быть блоком.

Основная форма записи цикла в Go похожа на цикл for языка C:

```
for простой_оператор; условие; простой_оператор блок
```

Допустима сокращённая форма (фактически, цикл while языка C):

```
for условие блок
```

Бесконечный цикл записывается просто как

```
for блок
```

Кроме того, особая форма цикла используется для перебора элементов массивов, срезов, отображений, а также для получения значений из типизированных каналов. Эту форму мы рассмотрим позже.

Для выхода из цикла можно применять оператор break, а для перехода к следующей итерации цикла – оператор continue.

Пример. Числа Фибоначчи

```
1 fmt.Print("1\n")
2 for a, b := 1, 1; b < 100; a, b = b, a+b {
3         fmt.Printf("%d\n", b)
4 }
```

Пример. Быстрое возведение в степень

```
1 res := 1
2 for n > 0 {
3         if n & 1 == 1 {
4                 res *= x
5         }
6         x *= x
7         n >>= 1
8 }
```

§5. Примитивные типы, массивы и указатели

Обозначение	Знак	Бит	Мин	Макс
int8	да	8	-128	127
uint8 (либо byte)	нет	8	0	255
int16	да	16	-32768	32767
uint16	нет	16	0	65535
int32	да	32	-2^{31}	$2^{31}-1$
uint32 (либо rune)	нет	32	0	$2^{32}-1$
int64	да	64	-2^{63}	$2^{63}-1$
uint64	нет	64	0	$2^{64}-1$
int	да	32 или 64		
uint	нет	32 или 64		
uintptr	нет	«дикий» указатель		
float32	да	32	$\pm 3.4 \cdot 10^{\pm 38}$ (~ 7 цифр)	
float64	да	64	$\pm 1.7 \cdot 10^{\pm 308}$ (~ 15 цифр)	
complex64	(float32, float32)			
complex128	(float64, float64)			

В Go можно объявлять одномерные массивы, индексируемые с нуля.

Тип массива записывается как

```
[размер] тип_элемента
```

Пример.

```
var A [10]int
const N = 100
var B, C, D [N]float64
var M [5][4]int  // эквивалентно [5]([4]int)
```

Для обращения к элементу массива используется операция индексации, записываемая также, как и в языке C:

```
A[i]
```

При этом выходить за границы массива запрещено.

Размер массива можно получить, если вызвать встроенную функцию len:

```
fmt.Printf("Size of A is %d\n", len(A))
```

Для перебора всех элементов массива в Go предусмотрена специальная форма цикла

```
for индекс, значение = range массив блок
```

Пример.

```
1  package main

3  import "fmt"

5  func main() {
6      var A [10]int
7      for i := 0; i < 10; i++ {
8          fmt.Scanf("%d", &A[i])
9      }
10     for i, x := range A {
11         fmt.Printf("A[%d] = %d\n", i, x)
12     }
13 }
```

В Go можно объявлять безымянные инициализированные массивы с помощью так называемых *составных литералов*.

Составной литерал записывается как

```
[размер] тип_элемента { список_значений }
```

Пример.

```
[10]int { 1, 2, 3 /* остальное - нули */ }
```

Если мы хотим, чтобы компилятор посчитал размер массива за нас, мы вместо размера ставим троеточие:

```
[...]int { 1, 2, 3 }  // массив размера 3
```

Пример.

```
1  package main

3  import "fmt"

5  func f(a [5]int) {
6      for i, x := range a {
7          fmt.Printf("%d) %d; ", i, x)
8      }
9  }

11 func main() {
12     f([...]int{ 10, 20, 30, 40, 50 })
13 }
```

Вывод:

```
0) 10; 1) 20; 2) 30; 3) 40; 4) 50;
```

Тип указателя в Go записывается как

```
*тип
```

Пример.

```
var p *int
var q **int
var ap *[10]int
```

Для разыменования указателя используется операция *:

```
*p
```

Для получения адреса ячейки памяти, используется операция &:

```
p = &(A[i])
```

В Go запрещены арифметика указателей, применение операции индексации к указателям, а также void-указатели.

§6. Срезы

Массивы редко используются при программировании на Go:
– при объявлении массива нужно указывать его размер, поэтому невозможно написать функцию, обрабатывающую массивы произвольного размера (в языке C такая функция принимала бы в качестве параметра указатель на первый элемент массива, но в Go к указателям неприменима операция индексации, поэтому это – не решение проблемы);
– использование массива в качестве контейнера для накопления некоторых значений неудобно, потому что размер массива фиксирован раз и навсегда (когда массив полностью заполняется, невозможно перевыделить память большего размера), и, кроме того, приходится заводить дополнительную переменную для хранения количества «занятых» элементов массива.

Вместо массивов в программах, написанных на Go, используются *срезы* (slices).

Срез – это структура данных, в которой хранится информация о фрагменте массива: указатель на первый элемент фрагмента, количество элементов во фрагменте, а также количество «занятых» элементов.

Тип среза записывается как

```
[] тип_элемента
```

Пример.

```
var A []int
var P []*int
var M [][]float64
```

К срезам применима операция индексации. При этом выходить за границы «занятых» элементов среза запрещено.

Длиной среза считается количество «занятых» элементов. Она возвращается встроенной функцией len.
Общее количество элементов среза – его *вместимость* – возвращает встроенная функция cap.
Неинициализированный срез имеет нулевую длину и нулевую вместимость. Считается, что он равен nil.

Пример.

```
1 package main

3 import "fmt"

5 func main() {
6     var a []int
7     fmt.Printf("%d %d\n", len(a), cap(a))
8 }
```

Вывод:

```
0 0
```

Составной литерал для среза записывается как

```
[] тип_элемента { список_значений }
```

Пример.

```
1 package main

3 import "fmt"

5 func main() {
6     a := []int { 1, 2, 3 }
7     fmt.Printf("%d %d\n", len(a), cap(a))
8 }
```

Вывод:

```
3 3
```

Срез можно получить, применив операцию *вырезания* (slicing) к массиву или другому срезу. Операция вырезания в общем виде записывается как

```
a [ start : follow ]
```

Здесь start – номер первого элемента вырезаемого фрагмента, а follow – номер элемента, непосредственно следующего за последним вырезаемым элементом.

Пример.

```
5  func main() {
6      array := [7]byte { 'a', 'b', 'c', 'd', 'e', 'f', 'g' }
7      slice := array[2:5]
8      for _, x := range slice {
9          fmt.Printf("%c ", x)
10     }
11 }
```

Вывод:

```
c d e
```

Номера start и follow в операции вырезания можно не указывать.

Значение по умолчанию для start – 0, а для follow – длина массива или среза.

Пример.

Если дан массив

```
array := [7]byte { 'a', 'b', 'c', 'd', 'e', 'f', 'g' }
```

то различные сочетания пропущенных start и follow дают нам:

`array[:4]`	`[]byte { 'a', 'b', 'c', 'd' }`
`array[2:]`	`[]byte { 'c', 'd', 'e', 'f', 'g' }`
`array[:]`	`[]byte { 'a', 'b', 'c', 'd', 'e', 'f', 'g' }`

Важно понимать, что операция вырезания не копирует элементы массива, то есть срез всего лишь запоминает координаты вырезаемого фрагмента.

Пример.

```
5  func main() {
6      A := [5]int { 1, 2, 3, 4, 5 }
7      b := A[2:]
8      b[1] = 10
9      for _, x := range A {
10         fmt.Printf("%d ", x)
11     }
12 }
```

Вывод:

```
1 2 3 10 5
```

Интересно, что вместимость среза, получаемого путём вырезания фрагмента массива, зависит от размера массива:

$$cap(slice) = len(array) - start$$

Пример.

```
5 func main() {
6       var A [100]int
7       slice := A[10:20]
8       fmt.Printf("%d %d", len(slice), cap(slice))
9 }
```

Вывод:

```
10 90
```

Пример можно усложнить, вырезав фрагмент среза.

Пример.

```
5  func main() {
6      var A [100]int
7      slice := A[10:20]
8      slice2 := slice[5:]
9      fmt.Printf("%d %d", len(slice2), cap(slice2))
10 }
```

Вывод:

```
5 85
```

При вырезании фрагмента среза значение follow может превышать длину среза. Тем самым мы получаем срез большей длины.

Однако, длину среза можно увеличивать лишь до значения, не превышающего его вместимости.

Пример.

```
5 func main() {
6     var A [100]int
7     slice := A[10:20]
8     slice2 := slice[:30]
9     fmt.Printf("%d %d", len(slice2), cap(slice2))
10 }
```

Вывод:

```
30 90
```

Копирование элементов одного среза в другой срез осуществляется встроенной функцией

```
func copy(dest, source []T) int
```

Эта функция копирует элементы среза source в срез dest. Длины срезов могут различаться, при этом копируется $min(len(source), len(dest))$ элементов.

Пример.

```
5  func main() {
6      A := [8]int { 0, 1, 2, 3, 4, 5, 6, 7 }
7      copy(A[1:3], A[5:])
8      for _, x := range A {
9          fmt.Printf("%d ", x)
10     }
11 }
```

Вывод:

```
0 5 6 3 4 5 6 7
```

§7. Динамические массивы

Массивы можно создавать в куче (так называется динамическая память) с помощью встроенной функции

```
func make(^[]T^, length, capacity) []T
```

Приведённый «прототип» функции make весьма условен, потому что такую функцию невозможно объявить на языке Go.

Функция make создаёт в куче массив размера capacity, тип элементов которого – T, и возвращает срез длины length из этого массива. При этом начальным элементом среза будет являться первый элемент массива.

Пример.

```
A := make([]int, 0, 100)
```

Существует сокращённая форма функции make:

```
func make([]T, capacity) []T
```

Она работает так же, как и полная форма, за исключением того, что длина возвращаемого среза становится равной capacity.

Пример.

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

Чрезвычайно удобный способ работы с растущими динамическими массивами обеспечивает встроенная функция append:

```
func append(slice []T, x ...T) []T
```

Эта функция добавляет в конец среза один или несколько новых элементов и возвращает новый срез, отличающийся от старого большей длиной.

В случае, если вместимости среза не хватает, функция append выделяет новый массив большего размера и копирует туда значения элементов старого массива. При этом возвращаемый срез будет указывать на новый массив.

Пример.

```
1  package main

3  import "fmt"

5  func main() {
6      s := make([]int, 0, 5)
7      s = append(s, 1)
8      for a, b := 1, 1; b < 100; a, b = b, a+b {
9          s = append(s, b)
10     }
11     for _, x := range s {
12         fmt.Printf("%d ", x)
13     }
14 }
```

Вывод:

```
1 1 2 3 5 8 13 21 34 55 89
```

§8. Строки

Для представления строк в Go предусмотрен встроенный тип string.

Фактически, строки представляют собой неизменяемые массивы типа byte, содержащие образ текста, закодированного в UTF-8.

Для доступа к отдельным байтам (но не кодовым точкам!) образа текста можно применять операцию индексации. При этом запись в элементы строки запрещена, а также к элементам строки запрещено применять операцию получения адреса &.

Допустимо применять к строкам операцию вырезания. При этом операция вырезания возвращает не срез, а строку, и работает на уровне байтов, а не кодовых точек.

Операция + применительно к строкам означает конкатенацию, причём в силу неизменяемости строк в результате конкатенации порождается новая строка.

Пример.

```
1 package main

3 import "fmt"

5 func main() {
6     s := "кошка"
7     fmt.Printf("len(%q) == %d\n", s, len(s))
8 }
```

Вывод:

```
len("\u043a\u043e\u0448\u043a\u0430") == 10
```

Операция преобразования значения в Go записывается как

```
T(x)
```

Здесь T – это тип результата преобразования, причём в случае, если T нельзя записать в виде одного идентификатора, T берётся в круглые скобки.

Операция преобразования очень полезна при работе со строками:
1. преобразование целого числа *x* к строке даёт строку, содержащую представление кодовой точки *x* в UTF-8;
2. преобразование байтового среза ([]byte) к строке даёт строку, содержащую те же самые байты, что и этот срез;
3. преобразование строки к []byte создаёт динамический массив байт, копирует туда байты строки и возвращает срез, указывающий на этот массив;
4. преобразование целочисленного среза ([]rune) к строке даёт строку, содержащую представление последовательности кодовых точек из среза в UTF-8;
5. преобразование строки к []rune создаёт динамический массив целых чисел, копирует туда последовательность кодовых точек, закодированную в строке, и возвращает срез, указывающий на этот массив.

Пример.

```
5 func main() {
6       s := "кошка"
7       a := ([]rune)(s)
8       for _, x := range a {
9           fmt.Printf("%c - %d\n", x, x)
10      }
11 }
```

Вывод:

```
к - 1082
о - 1086
ш - 1096
к - 1082
а - 1072
```

§9. Глобальные функции

Объявление глобальной функции записывается как

```
func имя сигнатура блок
```

Сигнатура описывает формальные параметры и возвращаемые значения функции:

```
параметры результат
```

Параметры записываются в виде

```
(пар1 тип1, ..., парM типM)
```

При этом, если у N идущих подряд параметров совпадает тип, то у первых $N-1$ параметров тип можно не указывать.

Возвращаемые значения в общем случае записываются так же, как параметры. Однако в случае единственного возвращаемого значения допустимо просто указать его тип.

Пример.

```
 5 func minmax(a, b int) (min, max int) {
 6     if a < b {
 7         min, max = a, b
 8     } else {
 9         max, min = a, b
10     }
11     return
12 }
```

Оператор return, увы, должен обязательно присутствовать :-(

Пример.

```
 5 func min(a, b int) int {
 6     if a < b {
 7         return a
 8     }
 9     return b
10 }
```

Допускается не указывать имена возвращаемых значений.

Пример.

```
5 func minmax(a, b int) (int, int) {
6     if a < b {
7         return a, b
8     }
9     return b, a
10 }
```

Если функция вообще ничего не возвращает, объявление возвращаемых значений просто пропускается.

Пример.

```
5 func hello(name string) {
6     fmt.Printf("Hello, %s!\n", name)
7 }
```

Операция вызова функции записывается как

```
f(пар1, ..., парM)
```

При этом если количество и типы возвращаемых значений некоторой функции g совпадают с количеством и типами формальных параметров функции f, то допустима следующая запись композиции этих функций:

```
f(g(...))
```

Пример.

```
5  func g(x int) (int, int) { return x, -x }

7  func f(a, b int) int { return a + b }

9  func main() {
10     fmt.Printf("%d", f(g(5)))
11 }
```

Если функция должна принимать произвольное количество фактических параметров, то последний формальный параметр трактуется как массив значений и объявляется как `имя ...тип`

Пример.

```
5  func concat(s ...string) string {
6      length := 0
7      for _, x := range s { length += len(x) }
8      result, pos := make([]byte, length), 0
9      for _, x := range s {
10         copy(result[pos:], ([]byte)(x))
11         pos += len(x)
12     }
13     return string(result)
14 }

16 func main() {
17     fmt.Print(concat("alpha", "beta", "gamma"))
18 }
```

Если мы хотим передать массив значений в функцию, принимающую произвольное количество фактических параметров, то можно использовать синтаксис, демонстрируемый следующим примером.

Пример.

```
16 func main() {
17     a := []string { "alpha", "beta", "gamma" }
18     fmt.Print(concat(a...))
19 }
```

§10. Функции как данные

В Go функции являются полноправными значениями.

Тип функции записывается как

```
func сигнатура
```

При этом имена параметров и возвращаемых значений в сигнатуре можно не указывать.

Пример.

```
var f func(a, b int)
var g func(x float32) string
var A []func([] byte) (int, int)
```

Операция вызова, естественно, применима не только к глобальным функциям, но и вообще ко всем значениям типа функция.

Значение типа функция называется *замыканием* и конструируется с помощью функционального литерала, записываемого в виде

```
func сигнатура блок
```

Если функциональный литерал находится внутри некоторого блока, то в его теле доступны все локальные переменные, видимые в этом блоке.

Пример.

```
5  func main() {
6      var x int
7      f := func(y int) int { return x + y }
8      x = 5
9      fmt.Printf("%d", f(10))
10 }
```

Вывод:

```
15
```

На практике особенно полезно передавать замыкания другим функциям.

Пример. «Универсальная» сортировка прямым выбором

```
5  func sort(n int, less func(int,int) bool, swap func(int,int)){
6      for j := n-1; j > 0; j-- {
7          k := j
8          for i := j-1; i >= 0; i-- {
9              if less(k, i) { k = i }
10         }
11         swap(j, k)
12     }
13 }

15 func main() {
16     A := []int { 5, 2, 6, 1, 3, 5, 0, 9 }
17     sort(len(A),
18          func (i, j int) bool { return A[i] < A[j] },
19          func (i, j int) { A[i], A[j] = A[j], A[i] },
20          )
21     for _, x := range A { fmt.Printf("%d ", x) }
22 }
```

С помощью нехитрого приёма можно создать рекурсивное замыкание.

Пример. Факториал

```
5  func main() {
6      var fact func(int) int
7      fact = func(x int) int {
8          if x == 0 {
9              return 1
10         }
11         return x*fact(x-1)
12     }
13     fmt.Printf("%d", fact(4))
14 }
```

§11. Сопрограммы, go-программы и каналы

Вообще, сопрограмма (coroutine) – это обобщение понятия подпрограммы. («Subroutines are special cases of ... coroutines.» – Дональд Кнут).

Сценарий выполнения подпрограммы: вызов – вычисления – возвращение значения.

Сценарий выполнения сопрограммы: вызов – вычисления – передача данных другой сопрограмме – вычисления – передача данных другой сопрограмме – ... – завершение работы.

Согласно классической точке зрения, из нескольких запущенных сопрограмм всегда работает только одна, переключение между сопрограммами происходит в момент передачи данных от одной сопрограммы к другой.

В go реализована поддержка так называемых go-программ (goroutines), которые отличаются от классических сопрограмм тем, что на многопроцессорной системе могут работать параллельно.

Пример. (numbers – go-программа)

```
1  package main

3  import "fmt"

5  func numbers(n int, ch chan int) {
6      for i := 0; i < n; i++ { ch <- i }
7      close(ch)
8  }

10 func main() {
11     nums := make(chan int)
12     go numbers(10, nums)
13     for x := range nums {
14         fmt.Printf("%d ", x)
15     }
16 }
```

Вывод:

```
0 1 2 3 4 5 6 7 8 9
```

Go-программа оформляется как обычная функция языка Go, не возвращающая значения. При этом допускаются go-программы, вложенные в другие функции.

Запуск go-программы осуществляется оператором go:

```
go вызов_функции
```

Пример. (Вложенная go-программа)

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

12     for x := range nums {
13         fmt.Printf("%d ", x)
14     }
15 }
```

Каналы – это типизированные очереди, через которые go-программы обмениваются данными.

Тип канала записывается как

```
chan тип_элемента_очереди
```

Каналы бывают синхронные и асинхронные.

Синхронный канал – это вырожденная очередь, имеющая нулевую длину. Когда go-программа помещает значение в синхронный канал, она блокируется до тех пор, пока другая go-программа не заберёт это значение из канала. Синхронные каналы создаются специальной формой функции make:

```
func make(^chan T^) chan T
```

Пример. (создание синхронного канала срезов типа int)

```
ch := make(chan []int)
```

Асинхронный канал – это очередь ненулевой длины. Когда go-программа помещает значение в асинхронный канал, она в общем случае не блокируется (блокировка происходит только в случае, если канал переполнен). Асинхронные каналы создаются специальной формой функции make:

```
func make(^chan T^, capacity) chan T
```

Пример. (создание асинхронного канала чисел с плавающей точкой)

```
ch := make(chan float64, 100)
```

Помещение значения в канал осуществляется оператором

```
ch <- x
```

Здесь ch – канал некоторого типа T, а x – значение типа T.

Пример.

```
ch := make(chan int, 100)
for i := 0; i < 100; i++ { ch <- i }
```

Для каналов работают встроенные функции len и cap.

Функция len возвращает количество элементов, реально лежащих в канале, а функция cap — вместимость канала.

Пример.

```
5 func main() {
6     nums := make(chan int, 100)
7     nums <- 10
8     nums <- 20
9     nums <- 30
10    fmt.Printf("%d %d\n", len(nums), cap(nums))
11 }
```

Вывод:

```
3 100
```

Приём значения из канала осуществляется унарной операцией

```
<-ch
```

Если канал пуст, то вызвавшая эту операцию go-программа блокируется до тех пор, пока в канале не появится значение.

Пример.

```
5 func main() {
6     nums := make(chan int, 10)
7     for i := 0; i < 10; i++ { nums <- i }
8     for i := 0; i < 10; i++ { fmt.Printf("%d ", <-nums) }
9 }
```

Пример. (самый простой dead lock – «автоблокировка» go-программы)

```
<-make(chan int)
```

Канал может быть закрыт с помощью вызова встроенной функции

```
func close(chan T)
```

В закрытый канал нельзя добавлять новые значения, однако из непустого закрытого канала можно вытаскивать значения, добавленные в него до закрытия.

Пример.

```
5  func main() {
6      nums := make(chan int, 10)
7      for i := 0; i < 10; i++ { nums <- i }
8      close(nums)
9      for i := 0; i < 10; i++ { fmt.Printf("%d ", <-nums) }
10 }
```

Вывод:

```
0 1 2 3 4 5 6 7 8 9
```

Операция приёма значения для закрытого канала работает своеобразно: когда в закрытом канале заканчиваются значения, она не блокирует go-программу, а начинает возвращать нули:

Пример.

```
5  func main() {
6      nums := make(chan int, 10)
7      for i := 0; i < 10; i++ { nums <- i }
8      close(nums)
9      for i := 0; i < 15; i++ { fmt.Printf("%d ", <-nums) }
10 }
```

Вывод:

```
0 1 2 3 4 5 6 7 8 9 0 0 0 0 0
```

Для того чтобы определить, закрыт ли канал, можно использовать специальную комбинацию кортежного присваивания и операции приёма значения из канала:

```
x, ok = <-ch
```

или

```
x, ok := <-ch
```

Это присваивание вынимает значение из канала ch, если канал непуст. При этом булевская переменная ok принимает значение true. Если же канал пуст, то возможны два варианта:

1. канал закрыт – тогда ok устанавливается в false;

2. канал не закрыт – тогда происходит ожидание появления в канале значения или закрытия канала.

Пример.

```
5  package main

7  import "fmt"

9  func main() {
10     nums := make(chan int, 10)
11     for i := 0; i < 10; i++ { nums <- i }
12     close(nums)
13     for {
14         x, ok := <-nums
15         if !ok { break }
16         fmt.Printf("%d ", x)
17     }
18 }
```

Вывод:

```
0 1 2 3 4 5 6 7 8 9
```

Для выборки всех значений из канала можно использовать специальную форму оператора for:

```
for переменная := range канал блок
```

Цикл завершается в случае закрытия и опустошения канала.

Пример.

```
5  func main() {
6      nums := make(chan int, 10)
7      for i := 0; i < 10; i++ { nums <- i }
8      close(nums)
9      for x := range nums {
10         fmt.Printf("%d ", x)
11     }
12 }
```

§12. Оператор select

Для изощрённой работы с каналами используется оператор select:

```
select {
case коммуникация: последовательность_операторов
...
case коммуникация: последовательность_операторов
default: последовательность_операторов
}
```

«Коммуникация» – это либо оператор, помещающий значение в канал, либо оператор, вытаскивающий значение из канала. Секция «default:» – необязательна.

Пример. (совершенно бессмысленный)

```
 5 select {
 6 case x := <-ch1:
 7     sum += x
 8 case s := <-ch2:
 9     fmt.Printf("%s\n", s)
10 case ch3 <- sum:
11     sum = 0
12 default:
13     sum *= 2
14 }
```

Алгоритм выполнения оператора select таков:
1. вычисляются все выражения, входящие в «коммуникации»;
2. выполняется поиск «коммуникаций», которые могут быть выполнены;
3. если такие «коммуникации» существуют, переход на 6;
4. если в операторе select есть секция «default:», то управление передаётся на соответствующую ей последовательность операторов, а после завершения этой последовательности выполнение оператора select заканчивается;
5. текущая go-программа блокируется до тех пор, пока хотя бы одна из «коммуникаций» не окажется выполнимой;
6. из выполнимых «коммуникаций» случайным образом выбирается одна и выполняется, после чего управление передаётся на соответствующую ей последовательность операторов.

Пример. (демонстрирует случайность выбора «коммуникации»)

```
5  func main() {
6      ch := make(chan int, 2)
7      for i, k := 0, 0; i < 12; i++ {
8          select {
9          case x := <-ch:
10             fmt.Printf("(#%d)", x)
11         case ch <- k:
12             fmt.Printf("(!%d)", k)
13             k++
14         }
15     }
16 }
```

Вывод:

```
(!0)(#0)(!1)(!2)(#1)(#2)(!3)(#3)(!4)(!5)(#4)(!6)
```

Оператор select позволяет выполнить неблокирующее чтение из канала, то есть вытащить из него значение, если оно там есть:

```
var (
    x int
    hasValue bool
)
select {
case x = <-nums:
    hasValue = true
default:
    hasValue = false
}
```

Пример. (go-программа unite выполняет слияние двух каналов в один)

```
5  func unite(in1, in2, out chan int) {
6      open1, open2 := true, true
7      for open1 || open2 {
8          var x int
9          select {
10         case x, open1 = <-in1:
11             if open1 {
12                 out <- x
13             }
14         case x, open2 = <-in2:
15             if open2 {
16                 out <- x
17             }
18         }
19     }
20     close(out)
21 }
```

```
23 func generate(start, finish int, out chan int) {
24     for i := start; i <= finish; i++ {
25         out <- i
26     }
27     close(out)
28 }

30 func main() {
31     ch1 := make(chan int)
32     ch2 := make(chan int)
33     ch := make(chan int)
34     go generate(1, 10, ch1)
35     go generate(50, 55, ch2)
36     go unite(ch1, ch2, ch)
37     for x := range ch {
38         fmt.Printf("%d ", x)
39     }
40 }
```

§13. Именованные типы данных и совместимость по присваиванию

Типовый литерал – это изображение типа данных в программе.

Типы данных, определяемые составным типовым литералом, – *неименованные*.

Пример.

```
*int
[]int
func (int, int) int
```

Типы, определяемые одним только именем, – *именованные*. Примитивные типы языка Go являются именованными.

Пример.

```
int
float64
```

Язык Go разрешает создавать пользовательские именованные типы данных. Объявление именованного типа выглядит как

```
type имя типовый_литерал
```

Пример.

```
type Color int
type Name string
type ColorSlice []Color
type Matrix [3][3]float32
type NameChan chan Name
type Filter func([]int) []int
```

Для понимания семантики многих конструкций языка Go важно понятие *базового типа*.

Пусть T – некоторый тип. Если T – примитивный тип или неименованный тип, то базовым типом для T является сам тип T.

Если T – пользовательский именованный тип, объявление которого выглядит как

```
type T типовый_литерал
```

тогда базовым типом для T будет являться базовый тип для типового литерала в объявлении.

(Нетрудно заметить, что определение базового типа – рекурсивное.)

Пример.

```
type Color int
type Colour Color
type NiceColour Colour
type ColorSlice []Color
type ColorSet ColorSlice
```

Базовый тип для `Color`, `Colour` и `NiceColour` — `int`.

Базовый тип для `ColorSlice` и `ColorSet` — `[]Color`.

Ещё одним важным понятием является *идентичность типов*.

Два типа T и V – идентичны тогда и только тогда, когда соблюдается одно из следующих условий:

1. T и V – один и тот же именованный тип;

2. T и V не являются именованными типами и представлены одинаковыми типовыми литералами.

Значение x типа Q можно присвоить переменной типа T, если соблюдается одно из следующих условий:

1. тип Q идентичен типу T;

2. тип Q и тип T имеют идентичные базовые типы, и хотя бы один из типов T и Q не является именованным типом;

3. T – это интерфейс (см. §16), и тип Q его реализует;

4. x – это nil, и T является указателем, функцией, срезом, каналом, отображением (см. §17) или интерфейсом;

5. x – это примитивная константа и базовый тип для T – примитивный тип, значение которого может изображаться такой константой.

Пример.

```
type IntSlice []int
type Ints []int
var (
    a []int = []int { 1, 2, 3 }
    x IntSlice
    y Ints
)
x = a  // ok
y = a  // ok
a = x  // ok
a = y  // ok
x = y  // ошибка!
```

```
type Color int
var (
    i int = 1
    c Color
)
c = 1  // ok
c = i  // ошибка!
i = c  // ошибка!
```

В типовом литерале, входящем в объявление пользовательского именованного типа, допустимо использовать имя самого объявляемого типа. Эта возможность полезна для создания динамических типов данных (списки, и т.д.).

При этом, в отличие от языка C, в Go можно объявлять совершенно инфернальные рекурсивные типы.

Пример. (безумный тип данных)

```
1 package main
2 import "fmt"
3 type T *T
4 func main() {
5     var w, x, y, z T
6     w, x, y, z = &x, &y, &z, nil
7     for i := w; i != nil; i = *i {
8         fmt.Printf("& ")
9     }
10 }
```

§14. Наборы методов

С любым пользовательским именованным типом (или с указателем на пользовательский именованный тип) можно связать множество специальных функций, называемых *методами*.

Метод – это глобальная функция, имеющая параметр-приёмник. Она объявляется как

```
func (приёмник) имя сигнатура блок
```

Приёмник – это специальный параметр, объявляемый как

```
имя_параметра пользовательский_именованный_тип
```

либо

```
имя_параметра *пользовательский_именованный_тип
```

Пример.

```
type IntSlice []int

func (slice IntSlice) Len() int { return len(slice) }

func (slice *IntSlice) Append(x int) {
    *slice = append(*slice, x)
}
```

С типом `IntSlice` связан только метод `Len`.

С указателем на `IntSlice` связан метод `Append`, а также метод `Len`. Дело в том, что набор методов, связанный с некоторым именованным пользовательским типом T, автоматически переходит и на тип $*T$.

Для вызова методов предусмотрен особый синтаксис:

```
параметр_приёмник.имя_метода(пар1, ..., парM)
```

Набор методов для типа параметра-приёмника должен содержать вызываемый метод.

Пример.

```
12 func main() {
13     a := make(IntSlice, 0, 5)
14     for i := 0; i < 10; i++ {
15         (&a).Append(i) // можно написать a.Append(i)
16     }
17     for _, x := range a {
18         fmt.Printf("%d ", x)
19     }
20 }
```

§15. Структуры

Тип-структура аналогичен структурам языка C и объявляется как

```
struct {
        объявления полей
}
```

Объявления полей в простейшем случае аналогичны объявлениям переменных, но не начинаются с ключевого слова var.

Пример.

```
struct {
    x, y int
    name string
}
```

Понятия «тег структуры» в языке Go не существует. Чтобы дать структуре имя, нужно объявить пользовательский именованный тип.

Пример.

```
type Point3D struct {
    x, y, z float64
}

type Node struct {
    prev, next *Node
    value string
}

type List struct {
    tail List // а вот так нельзя
    x int
}
```

Доступ к полю структуры осуществляется с помощью операции «точка»:

```
s.имя_поля
```

При этом, если s – указатель на структуру, запись `s.имя_поля` эквивалентна `(*s).имя_поля`. То есть в Go операция «точка» для указателей ведёт себя как операция -> языка C.

Пример.

```
1 package main
2 import "fmt"
3 func main() {
4     var s struct { x, y int }
5     s.x = 10
6     p := &s
7     p.y = 20
8     fmt.Printf("(%d, %d)\n", s.x, s.y)
9 }
```

Составной литерал для структуры записывается как

```
тип_структуры { значения_полей }
```

или как

```
тип_структуры { поле: значение, ..., поле: значение }
```

Пример.

```
Point3D { 10, 0, 20 }
Point3D { x: 10, z: 20 } // y будет 0
struct { a, b string } { "qwerty", "asdf" }
```

Поля в объявлении структуры могут быть анонимными. Объявление анонимного поля выглядит как

```
T
```

или как

```
*T
```

где `T` – пользовательский именованный тип.

Пример.

```
type Coords struct { x, y, z int }
type Object struct {
    name string
    Coords
}
```

Для доступа к анонимным полям используется имена их типов. Если анонимное поле объявлено как `T` или как `*T`, то его имя – `T`.

Кроме того, если тип анонимного поля представляет структуру или указатель на структуру, то можно напрямую обращаться к полям этой структуры.

Пример.

```
1 package main
2 import "fmt"
3 type Point struct { x, y int }
4 type Point3D struct {
5     Point
6     z int
7 }
8 func main() {
9     var p Point3D
10    p.x, p.y, p.z = 10, 20, 30
11    fmt.Printf("%d", p.Point.x)
12 }
```

Методы анонимных полей «наследуются» типом-структурой.

Если структура S содержит анонимное поле T, то набор методов типа S включает методы типа T.

Если структура S содержит анонимное поле $*T$, то набор методов типа S включает методы типа $*T$ (и, в том числе, методы типа T).

Если структура S содержит анонимное поле T или $*T$, то набор методов типа $*S$ включает методы типа $*T$ (и, в том числе, методы типа T).

§16. Интерфейсы

Тип-интерфейс определяет набор методов и объявляется как

```
interface {
        объявления методов
}
```

Каждое объявление метода имеет вид

```
имя сигнатура
```

Пример.

```
interface {
    Add(x, y int) int
    Mul(x, y int) int
}
```

Если некоторый интерфейс A содержит методы, определяемые интерфейсом B, то вместо дублирования этих методов в объявлении интерфейса A указывается имя интерфейса B.

Пример.

```
type AdditiveEvaluator interface {
    Add(x, y int) int
    Sub(x, y int) int
}

type Evaluator interface {
    AdditiveEvaluator
    Mul(x, y int) int
    Div(x, y int) int
}
```

Значение типа интерфейс представляет собой «коробочку», содержащую значение некоторого типа T (или $*T$), а также таблицу указателей на методы типа T (или $*T$), имена и сигнатуры которых совпадают с именами и сигнатурами методов интерфейса.

Создание значения типа интерфейс происходит либо автоматически при присваивании или передаче параметров в функцию, либо явно с помощью операции приведения типа.

Пример. (автоматическое создание значения типа интерфейс)

```
 8 type Point struct {
 9         x, y int
10 }

12 func (p Point) Len() float64 {
13         return math.Sqrt(float64(p.x*p.x + p.y*p.y))
14 }

16 type Measurable interface {
17         Len() float64
18 }

20 func main() {
21         p := Point{ 30, 40 }
22         var m Measurable = p // Автоматическое "заворачивание" p
23         fmt.Printf("%f\n", m.Len())
24 }
```

Пример. (явное создание значения типа интерфейс)

```
 8 type Point struct {
 9         x, y int
10 }

12 func (p Point) Len() float64 {
13         return math.Sqrt(float64(p.x*p.x + p.y*p.y))
14 }

16 type Measurable interface {
17         Len() float64
18 }

20 func main() {
21         p := Point{ 30, 40 }
22         m := Measurable(p) // Явное "заворачивание" p
23         fmt.Printf("%f\n", m.Len())
24 }
```

Полиморфизм – это возможность передачи в качестве фактического параметра в функцию значений разных типов.

Для этого формальный параметр функции объявляется с типом интерфейс, и тогда все значения, наборы методов которых удовлетворяют этому интерфейсу, могут быть переданы функции.

В качестве примера рассмотрим библиотечную функцию Sort из пакета «sort»:

```
func Sort(data Interface)
```

При этом тип Interface объявлен как

```
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less returns whether the element with index i should sor
    // before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

Пример.

```
 1 package main

 3 import (
 4         "fmt"
 5         "sort"
 6 )

 8 type IntSlice []int

10 func (s IntSlice) Len() int { return len(s) }

12 func (s IntSlice) Less(i, j int) bool { return s[i] < s[j] }

14 func (s IntSlice) Swap(i, j int) { s[i], s[j] = s[j], s[i] }

16 func main() {
17         a := IntSlice { 5, 3, 7, 3, 4, 9, 2 }
18         sort.Sort(a)
19         for _, x := range a { fmt.Printf("%d ", x) }
20 }
```

§17. Отображения

Отображение – это словарь, отображающий некоторое множество ключей K в множество элементов V. Тип отображения записывается как

```
map [типK] типV
```

Неинициализированное отображение имеет значение nil.

Для типа K должны быть определены операции сравнения == и !=. Таким образом, ключами отображения не могут быть функции, отображения и срезы. Если ключом отображния является тип-интерфейс, то наличие операций сравнения для упакованных в этот интерфейс значений проверяется во время выполнения программы.

Пример.

```
var d map[string]int
var x map[int]func(int)int
```

Инициализация отображения осуществляется специальной формой функции make:

```
func make(^map[K]V^, capacity) map[K]V
```

Параметр capacity задаёт начальный размер отображения (количество словарных пар). При добавлении словарных пар отображение может динамически расти.

Пример.

```
dict := make(map[string]int, 100)
```

Существует ещё одна форма функции make для инициализации отображений с начальным размером по умолчанию:

```
func make(^map[K]V^) map[K]V
```

Количество словарных пар в отображении возвращает встроенная функция len.

Добавление словарной пары (k,v) в отображение a осуществляется оператором присваивания

```
a[k] = v
```

При этом, если в отображении уже существует словарная пара с этим же ключом, она заменяется новой словарной парой.

Поиск элемента отображения по ключу осуществляется операцией индексации

```
a[k]
```

При этом, если в отображении отсутствует словарная пара с указанным ключом, то операция индексации возвращает нулевое значение (оно зависит от типа элемента отображения).

Пример.

```
1 package main

3 import "fmt"

5 func main() {
6         a := make(map[string]int, 16)
7         a["alpha"] = 10
8         a["alpha"] = 20
9         fmt.Printf("%d %d\n", a["alpha"], a["beta"])
10 }
```

Вывод:

```
20 0
```

Проверить, существует ли в отображении словарная пара с ключом *k*, позволяет специальная форма оператора присваивания:

```
v, ok = a[k]
```

или `v, ok := a[k]`.

Переменная *ok* получает булевское значение true, если пара с ключом *k* существует.

Пример.

```
5  func main() {
6      a := make(map[string]int)
7      a["alpha"] = 10
8      v, ok := a["alpha"]
9      fmt.Printf("%d %v\n", v, ok)
10 }
```

Вывод:

```
10 true
```

Для удаления словарной пары с ключом k из отображения m используется встроенная функция delete:

```
delete(m, k)
```

Если словарной пары с ключом k не существует, функция delete ничего не делает.

Пример.

```
5  func main() {
6      a := make(map[string]int)
7      a["alpha"] = 10
8      delete(a, "alpha")
9      v, ok := a["alpha"]
10     fmt.Printf("%d %v\n", v, ok)
11 }
```

Вывод:

```
0 false
```

Составной литерал для отображения записывается как

```
map[типK]типV { k1: v1, ... , kN: vN }
```

Пример.

```
1  package main

3  import "fmt"

5  func main() {
6      a := map[int]string {
7          10: "alpha", 20: "beta", 30: "gamma",
8      }
9      fmt.Printf("%s\n", a[20])
10 }
```

Вывод:

```
beta
```

Перебор всех словарных пар, содержащихся в отображении *a*, осуществляется специальной формой оператора for:

```
for k, v = range a блок
```

или `for k, v := range a блок`.

Пример.

```
5  func main() {
6      a := map[int]bool { 10: true, 20: false, 300: true }
7      for k, v := range a {
8          fmt.Printf("(%d, %v) ", k, v)
9      }
10 }
```

Вывод:

```
(300, true) (10, true) (20, false)
```

§18. Обработка аварий

Авария – это чрезвычайная ситуация, возникающая во время работы программы из-за неправильных входных данных или ошибок ввода/вывода.

Пример.

```
5 func f(a, b int) int { return a/b }

7 func main() {
8       var x int
9       fmt.Scanf("%d", &x)
10      fmt.Printf("%d\n", f(100,x))
11 }
```

Ввод:

```
0
```

Вывод:

```
panic: runtime error: integer divide by zero

[signal 0x8 code=0x1 addr=0x8048c28 pc=0x8048c28]

runtime.panic+0xa4 /home/skor/go/src/pkg/runtime/proc.c:1083
        runtime.panic(0x808d510, 0x977b51d8)
runtime.panicstring+0x95 /home/skor/go/src/pkg/runtime/runtime.
        runtime.panicstring(0x812d83c, 0x0)
runtime.sigpanic+0x10c /home/skor/go/src/pkg/runtime/linux/thre
        runtime.sigpanic()
main.f+0x28 /home/skor/work/iu9/curricula/informatics/sem2/lyx_
        main.f(0x64, 0x0, 0x977b51f8, 0x1)
main.main+0xd9 /home/skor/work/iu9/curricula/informatics/sem2/l
        main.main()
runtime.mainstart+0xf /home/skor/go/src/pkg/runtime/386/asm.s:9
        runtime.mainstart()
runtime.goexit /home/skor/go/src/pkg/runtime/proc.c:150
        runtime.goexit()
----- goroutine created by -----
_rt0_386+0xc1 /home/skor/go/src/pkg/runtime/386/asm.s:80
```

В языке Go предусмотрен оператор defer, который откладывает запуск некоторой функции до завершения текущей функции.

Пример.

```
5 func main() {
6     defer fmt.Printf("good bye\n")
7     fmt.Printf("hello and...\n")
8 }
```

Вывод:

```
hello and...
good bye
```

Важной особенностью оператора defer является то, что отложенная функция вызывается даже в том случае, когда в текущей функции произошла авария.

Пример.

```
 5 func f(a, b int) int {
 6     defer fmt.Printf("f() returns\n")
 7     return g(a, b)
 8 }

10 func g(a, b int) int {
11     defer fmt.Printf("g() returns\n")
12     return a/b
13 }

15 func main() {
16     var x int
17     fmt.Scanf("%d", &x)
18     fmt.Printf("%d\n", f(100,x))
19 }
```

Ввод:

```
0
```

Вывод:

```
g() returns
f() returns
panic: runtime error: integer divide by zero

[signal 0x8 code=0x1 addr=0x8048d1c pc=0x8048d1c]

runtime.panic+0xa4 /home/skor/go/src/pkg/runtime/proc.c:1083
        runtime.panic(0x808d540, 0x9786e1c8)
runtime.panicstring+0x95 /home/skor/go/src/pkg/runtime/runtime.
        runtime.panicstring(0x812d83c, 0x8048d0d)
runtime.sigpanic+0x10c /home/skor/go/src/pkg/runtime/linux/thre
        runtime.sigpanic()
main.g+0x80 /home/skor/work/iu9/curricula/informatics/sem2/lyx_
        main.g(0x64, 0x0, 0x0, 0x0)
main.f+0x8b /home/skor/work/iu9/curricula/informatics/sem2/lyx_
        main.f(0x64, 0x0, 0x0, 0x1)
main.main+0xd9 /home/skor/work/iu9/curricula/informatics/sem2/l
...
```

Если стек вызовов функций выглядит как

```
main -> fN -> ... -> f2 -> f1
```

и авария возникает в функции f1, то выполнение функции f1 прерывается и сначала выполняются все отложенные вызовы, связанные с функцией f1. Затем прерывается выполнение функции f2 и выполняются все отложенные вызовы, связанные с функцией f2. И так далее вплоть до функции main, после чего в стандартный поток ошибок (по умолчанию – в консоль) выводится описание аварии и стек вызовов функций в момент возникновения аварии.

Аварию можно в любой момент перехватить и избежать аварийного завершения программы. Для этого в одной из функций, вызов которых был отложен, нужно вызвать встроенную функцию recover:

```
func recover() interface{}
```

Эта функция возвращает либо nil, либо описание аварии, которое, как правило, представляет собой строку. Значение nil возвращается, если никакой аварии не произошло.

Пример.

```
 5 func f(a, b int) (result int) {
 6     defer func() {
 7         if x := recover(); x != nil {
 8             fmt.Printf("recovered from %q\n", x)
 9             result = 666
10         }
11     }()

13     result = a/b
14     return
15 }

17 func main() {
18     var x int
19     fmt.Scanf("%d", &x)
20     fmt.Printf("%d\n", f(100,x))
21 }
```

Ввод:

```
0
```

Вывод:

```
recovered from "runtime error: integer divide by zero"
666
```

Для того, чтобы можно было вручную вызывать аварии, предусмотрена встроенная функция panic:

```
func panic(interface{})
```

В качестве параметра функция panic принимает произвольное значение, описывающее аварию.

§19. Пакеты

Очевидно, что любую программу на языке Go можно оформить в виде одного файла. Однако, этот подход не очень удобен с практической точки зрения:
1. такую программу в силу большого размера неудобно редактировать;
2. совместная разработка программы несколькими программистами существенно затрудняется;
3. становится невозможной раздельная компиляция – для очень больших проектов это означает большое время компиляции;
4. ухудшается качество кода (появляется большое количество зависимостей в коде), что затрудняет отладку, сопровождение, а также повторное использование кода в других программах .

Самый простой путь по структуризации исходников программы – это вынос логически цельных фрагментов в отдельные файлы.

Пример (файл fib.go)

```
1 package main

3 func fib(n int) (seq []int) {
4         for a, b := 1, 1; n > 0; n-- {
5                 seq = append(seq, a)
6                 a, b = b, a+b
7         }
8         return
9 }
```

Пример (файл x.go)

```
1 package main

3 import "fmt"

5 func main() {
6         a := fib(10)
7         fmt.Printf("%v\n", a)
8 }
```

Компиляция всех файлов должна осуществляться одновременно (то есть все файлы сразу нужно подсунуть компилятору Go):

```
go build -o fibtest fib.go x.go
```

Ключ `-o` задаёт имя порожаемого исполняемого файла.

Вызов полученного исполняемого файла выглядит как

```
./fibtest
[1 1 2 3 5 8 13 21 34 55]
```

Следующим логическим шагом в деле структуризации исходников станет оформление отдельных частей программы в виде пакетов.

Пакет – это единица раздельной компиляции в языке Go. Каждый пакет имеет имя и состоит из одного или нескольких файлов. Пакет, содержащий точку входа в программу (функцию `main`), должен иметь имя `main`.

Принадлежность файла к пакету задаётся в самом начале файла синтаксической конструкцией

```
package имя_пакета
```

Пакет может *экспортировать* часть объявленных в нём сущностей (типы, переменные, константы, функции, методы), и также *импортировать* сущности из других пакетов. Имена экспортируемых сущностей должны начинаться с заглавной буквы. Импорт сущностей из других пакетов осуществляется конструкцией

```
import путь_к_пакету
```

Здесь «путь_к_пакету» – это строковый литерал, в простейшем случае содержащий имя импортируемого пакета.

Для доступа к импортированным сущностям используются так называемые *квалифицированные имена*:

```
имя_пакета.имя_сущности
```

Пример (новая версия файла fib.go – теперь в пакете `sequences`)

```
1 package sequences

3 func Fib(n int) (seq []int) {
4         for a, b := 1, 1; n > 0; n-- {
5                 seq = append(seq, a)
6                 a, b = b, a+b
7         }
8         return
9 }
```

Пример (файл pow.go – тоже в пакете `sequences`)

```
1 package sequences

3 func Powers(x int, n int) (seq []int) {
4         for p := 1; n > 0; n-- {
5                 seq = append(seq, p)
6                 p *= x
7         }
8         return
9 }
```

Тем самым из пакета `sequences` экспортируются две сущности: функции `Fib` и `Powers`.

Пример (файл main.go – в пакете `main`)

```
1  package main

3  import "fmt"
4  import "sequences"

6  func main() {
7          a := sequences.Fib(10)
8          fmt.Printf("%v\n", a)
9          b := sequences.Powers(2, 10)
10         fmt.Printf("%v\n", b)
11 }
```

В практике программирования ручной ввод команд для сборки проекта не приветствуется. Вместо этого используются специальные инструментальные средства, позволяющие отслеживать зависимости раздельно компилируемых частей программы, и на основании этой информации запускать компилятор только для тех исходных файлов, которые изменились с момента предыдущей сборки.

Самым распространённым и известным из таких инструментальных средств является программа `make`. Её преимуществом является универсальность, то есть она подходит для сколь угодно сложных проектов, написанных на любых языках программирования. Однако, применительно к проектам, написанным на языке Go, у программы `make` есть более простой и удобный в использовании аналог: программа go.

В случае использования программы go для сборки проекта, проект должен размещаться в отдельном каталоге, структура которого в случае нашего примера должна выглядеть следующим образом:

```
./
    src/
        sequences/
            fib.go
            pow.go
        x/
            main.go
```

Сборка проекта осуществляется командой

```
GOPATH=‘pwd‘ go install ./src/x
```

При этом в корне каталога проекта создаётся подкаталог bin, куда помещается исполняемый файл.

Распределение файлов проекта по подкаталогам открывает возможность дальнейшего структурирования файлов проекта, а именно: с помощью подкаталогов мыможем группировать родственные пакеты.

Пример

```
./
    src/
        graphs/
            sparse/
                файлы пакета sparse
            dense/
                файлы пакета dense
        algorithms/
            walk/
                файлы пакета walk
            mst/
                файлы пакета mst
            components/
                файлы пакета components
```

При такой организации файлов проекта при импорте пакета мы должны указывать относительный путь к его каталогу:

```
import "algorithms/components"
```

При этом в квалифицированных именах, используемых для доступа к сущностям, экспортируемым из пакета components, путь не указывается:

```
components.Tarjan
```

Может возникнуть ситуация, когда в проекте существуют два или больше пакетов с одним именем. Для того чтобы можно было одновременно импортировать такие пакеты, в конструкции import предусмотрена возможность переименования пакета:

```
import graphWalk "algorithms/walk"
```

§20. Средства ввода/вывода

В языке Go реализована полиморфная система ввода/вывода, базирующаяся на интерфейсах, объявленных в пакете io.

Интерфейс `io.Reader` реализуют типы, обеспечивающие потоковый ввод данных:

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

Метод `Read` считывает данные (откуда-то) в срез байт `p` и возвращает количество считанных байт через `n` и описание ошибки через `err`. Описание ошибки имеет тип `error`:

```
type error interface {
    Error() string
}
```

Если ошибки не произошло, `err` равно `nil`.

Интерфейс `io.Writer` реализуют типы, обеспечивающие потоковый вывод данных:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Метод `Write` записывает (куда-то) данные, взятые из среза байт `p`, и возвращает количество записанных байт через `n` и описание ошибки через `err`.

Многие функции стандартной библиотеки осуществляют ввод/вывод через эти интерфейсы. Например, функция `fmt.Fprintf` обеспечивает форматный вывод строк:

```
func Fprintf(w io.Writer, format string, a ...interface{})
        (n int, error error)
```

Работа с файлами осуществляется посредством методов типа `File`, объявленного в пакете os.

Создание нового файла выполняется функцией `os.Create`:

```
func Create(name string) (file *File, err error)
```

Открытие существующего файла осуществляется функцией `os.Open`:

```
func Open(name string) (file *File, err error)
```

Тип `os.File` реализует интерфейсы `io.Reader` и `io.Writer`, то есть имеет методы `Read` и `Write`.

После работы с файлом полагается этот файл закрыть. Для закрытия файла предназначен метод `Close`:

```
func (file *File) Close() error
```

Пример

```
1  package main
2
3  import (
4          "fmt"
5          "os"
6  )
7
8  func main() {
9          if f, err := os.Create("numbers.txt"); err == nil {
10                 for i := 0; i < 100; i++ {
11                         fmt.Fprintf(f, "%d ", i)
12                 }
13                 f.Close()
14         } else {
15                 fmt.Printf("i/o error: %s", err.String())
16         }
17 }
```