UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO - CAMPUS GOIABEIRAS
DEPARTAMENTO DE INFORMÁTICA
LINGUAGENS DE PROGRAMAÇÃO 2016/2

GO: A LINGUAGEM DE PROGRAMAÇÃO DA GOOGLE

ANDRÉ GUASTI LOZER
ARTHUR DE A. NEVES
THAIS PIMENTA MENEZES

AGENDA

- Introdução
- Motivação e Objetivos da Linguagem
- Sintaxe
- Características da Linguagem
- Avaliação da Linguagem
- Referências Bibliográficas

INTRODUÇÃO

- Linguagem de prgramação C-like;
- Criada por equipe de engenheiros do Google;
- Se tornou código aberto em Novembro de 2009;
- Go 1.0 foi lançada em Março de 2012:
  - Especificação da linguagem;
  - Bibliotecas padrão;
  - Ferramentas customizadas;
  - Atualmente na versão 1.7.3.

Motivação e Objetivos da Linguagem

- Motivação: descontentamento com a complexidade de C++, Java e outras
- Objetivos:
  - Ser estaticamente tipada;
  - Ser eficiente e de alta confiabilidade;
  - Não requerer IDEs e suportar rede e multiprocessamento.
- Além disso, queriam uma linguagem mais adaptada para a realidade atual da computação (programação distribuída, nuvem, multicore CPUs)

Sintaxe

- Palavras-chave:

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

- Não há palavras reservadas
- Desvio incondicional é implementado
- Não há uso de ponto-e-vírgula

Identificadores pré-definidos

- Tipos:

bool	**byte**	**complex64**	**complex128**	**error**	**float32**	**float64**
int	**int8**	**int16**	**int32**	**int64**	**rune (char UTF-8)**	**string**
uint	**uint8**	**uint16**	**uint32**	**uint64**	**uintptr**	

- Constantes:

true	**false**	**iota**

- Nulo> **nil**
- Funções:

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

Operadores - Aritmética

- Adição: +
- Subtração: -
- Multiplicação: *
- Divisão: /
- Divisão (resto): %
- Incremento: ++
- Decremento: --

Operadores - Comparação

- Igual: ==
- Não igual: !=
- Maior: >
- Menor: <
- Maior/igual: >=
- Menor/igual: <=

Operadores - Lógicos e Bit-a-Bit

- Lógicos:
  - AND: &&
  - OR: ||
  - NOT: !
- Bit-a-bit:
  - AND: &
  - OR: |
  - XOR: ^
  - Shift esquerda/direita: <</>>

Operadores - Atribuição

• Simples: =	A=B
• Soma: +=	A+=B eq A=A+B
• Subtrativo: -=	A-=B eq A=A-B
• Multiplicativo: *=	A*=B eq A=A*B
• Divisor: /=	A/=B eq A=A/B
• Divisor (resto): %=	A%=B eq A=A%B
• Shift: <<= ou >>=	A<<=2 eq A=A<<2
• AND bit-a-bit: &=	A&=2 eq A=A&2
• XOR bit-a-bit: ^=	A^=2 eq A=A^2
• OR bit-a-bit	A|=2 eq A=A|2

Declaração de Variáveis

- Comando *var* declara uma ou mais variáveis

```
var a string = “nome” //char a[]=”nome”;
var a, b int 1, 2 // int a=1; int b=2;
```

- Há inferência de tipo

```
var c = false //infere booleano
```

Declaração de Variáveis

- Variáveis declaradas sem valor recebem valor-zero

```
var d int //int d=0;
```

- O comando de atribuição := também pode ser utilizado para declarar e inicializar uma variável

```
e := 5 //Equivalente a var e int = 5
```

Laços de repetição

- Go possui apenas o comando *for* para repetição;

```
//For normal
for j := 7; j <= 9; j++ {
        fmt.Println(j)
}
```

```
// While
i:=0;
for i <= 3 {
    fmt.Println(i)
    i = i + 1
}
```

```
//Loop infinito + break
for {
    fmt.Println("loop")
    break
}
```

Laços de repetição

- Também pode ser utilizado como *for* + *range* (equivalente a *for-each*);

```
nums := []int{2, 3, 4}
for i, num := range nums {
        if num == 3 {
                fmt.Println("index:", i)
        }
}
```

Condicionais

- Go apresenta as mesmas estruturas condicionais de C

```
if num := 9; num < 0 {          //num declarado aqui
        fmt.Println(num, “eh negativo")
} else if num < 10 {            //num pode ser acessado aqui
        fmt.Println(num, “tem 1 digito")
} else {
        fmt.Println(num, “tem varios digitos")
}
```

Condicionais

- Switch
  - Aceita qualquer tipo de dado;
  - Executa apenas o primeiro caso satisfeito (dispensa *break*);

```
switch time.Now().Weekday() {
case time.Friday:
        fmt.Println("hoje eh sexta =D")
case time.Monday, time.Tuesday, time.Wednesday, time.Thursday:
        fmt.Println("hoje nao eh sexta nem fds =(")
default:
        fmt.Println("Ufa! Eh fds, posso descansar.")
}
```

Condicionais

- Switch
  - Para a execução de multiplos casos, usa-se *fallthrough*;

```
s := "palavras"
switch {
case len(s)%2 == 0:
        fmt.Println("Tamanho eh par")
        fallthrough
case len(s)%4 == 0:
        fmt.Println("Tamanho eh multiplo de 4")
        fallthrough
case len(s)%8 == 0:
        fmt.Println("Tamanho eh multiplo de 8")
default:
        fmt.Println("Nao eh multiplo de 2, 4 ou
8.")
}
```

Funções

- Uso da palavra-chave *func*;
- Argumentos definidos entre parênteses, no padrão:

```
func funcaoA(argA tipoargA, argB tipoargB, ...) tiporetornoA
```

- Argumentos de mesmo tipo seguidos podem ter o tipo listado apenas no ultimo:

```
func soma(a, b int) int {
    return a + b
}
```

Funções

- Pode haver múltiplos valores de retorno, que podem ser nomeados;

```
func Div2(a int, b int) (int,int) {
        return a/2, b/2
}
```

- Chamando a função:

```
mult, div := Div2(4,2)
```

- Resultado: 2, 1

Funções com lista de parâmetros variável

- A lista de parâmetros de funções pode ser variável
- Declaração de parâmetros variáveis deve ser a última na lista
- Elementos de slices podem ser passados como parâmetros para tais funções

```
// Função que recebe número arbitrário de inteiros
func sum(nums ...int) {
	fmt.Print(nums, " ")
	total := 0
	for _, num := range nums {
		total += num
	}
	fmt.Println(total)
}
```

Closures

- Funções podem ser:
  - Passadas como parâmetros para outras funções
  - Retorno de funções
  - Atribuidas a variáveis
- Cada innstância de uma função apresenta suas próprias variáveis internas
- Saída do exemplo:
  - 1
  - 2
  - 3
  - 1

```
package main
import (fmt")
func intSeq() func() int {
	i := 0
	return func() int {
		i += 1
		return i
	}
}

func main() {
	//Essa instância da função terá seu próprio i na memória,que será alterado a cada chamada
	nextInt := intSeq()
	fmt.Println(nextInt())
	fmt.Println(nextInt())
	fmt.Println(nextInt())
// Outro i será criado para uma nova instância
	newInts := intSeq()
	fmt.Println(newInts())
}
```

Recursão

- Go dá suporte a recursão;
- O resultado do exemplo será 120;

```
func fatorial(n int) int {
	if n == 0 {
		return 1
	}
	return n * fatorial(n-1)
}
func main() {
	fmt.Println(fatorial(5))
}
```

Vetores

- GO difere de C apenas na declaração de arrays
- Deve-se definir o tipo de dados e o tamanho do vetor
- Há a função ***len*** para verificar o tamanho do vetor
- Declaração:

```
var a [5]int
```

- Atribuição de valor:

```
a[2]=15
```

Vetores - Exemplo

```
Saída:
emp: [0 0 0 0 0]
set: [0 0 0 0 100]
get: 100
len: 5
dcl: [1 2 3 4 5]
2d:  [[0 1 2] [1 2 3]]
```

```go
package main
import "fmt"
func main() {
	var a [5]int // Inicialização
	fmt.Println("emp:", a)
	a[4] = 100 // Acesso
	fmt.Println("set:", a)
	fmt.Println("get:", a[4])
	fmt.Println("len:", len(a))
//Tamanho
	b := [5]int{1, 2, 3, 4, 5} // Array
literal
	fmt.Println("dcl:", b)
	var twoD [2][3]int // Array 2D
	for i := 0; i < 2; i++ {
		for j := 0; j < 3; j++ {
			twoD[i][j] = i + j
		}
	}
	fmt.Println("2d: ", twoD)
}
```

Slices

- Estrutura chave na linguagem
- Como vetores, porém mais flexíveis
- Definidos apenas por tipo (tamanho variável)
- Capacidade diferente de tamanho

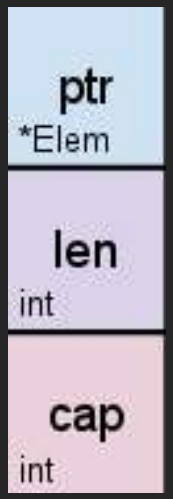
```
import "fmt"
func main() {
	s := []int{2, 3, 5, 7, 11, 13}
	printSlice(s)
// Slice the slice to give it zero length.
	s = s[:0]
	printSlice(s)
	// Extend its length.
	s = s[:4]
	printSlice(s)
	// Drop its first two values.
	s = s[2:]
	printSlice(s)
}

func printSlice(s []int) {
	fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)}
```

Slices - Exemplo

```
package main
import "fmt"
func main() {
// Não são presos ao tamanho,
apenas ao tipo
	s := make([]string, 2, 3)
	fmt.Println("emp:", s)
// Acesso igual a arrays
	s[0] = "a"
	s[1] = "b"
	fmt.Println("set:", s)
	fmt.Println("get:", s[1])
//'len' retorna o tamanho
//'cap' retorna a capacidade
	fmt.Println("len:", len(s))
	fmt.Println("cap:", cap(s))
// 'append' extende a capacidade
	s = append(s, "d")
	s = append(s, "e", "f")
	fmt.Println("apd:", s)
// Pode-se copiar slices
	c := make([]string, len(s))
	copy(c, s)
	fmt.Println("cpy:", c)
// Pode-se cortar slices para criar
slices menores. Os dois slices apontam
para a mesma estrutura na memória
	d := []byte{'g', 'a', 't', 'o'}
	e := d[:2]
// e == []byte{'g', 'a'}
	e[0] = 'r'
// e == []byte{'r', 'a'}
// d == []byte{'r', 'a', 't', 'o'}
}
```

Slices - Alocação em Memória

Estrutura armazenada:

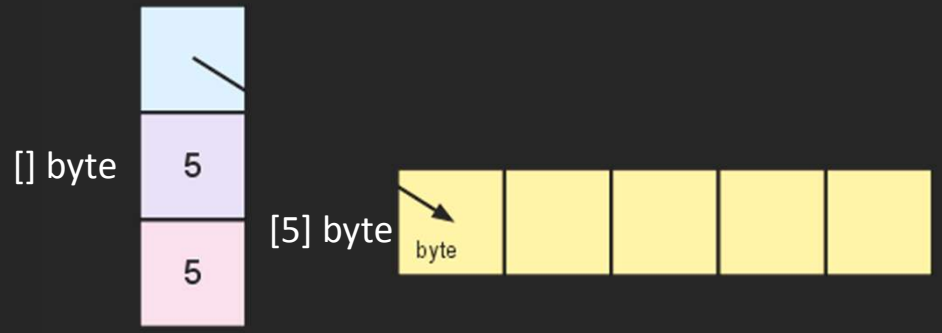


Ponteiro para elemento (vetor)

Tamanho do slice

Capacidade do slice

make([]byte, 5)



Mapas

- Similar a vetores e slices
- Índices (chaves) não precisam ser inteiros
- Forma:

```
map[KeyType]ValueType //Deve ser inicializado com make
```

  - KeyType precisa ser de qualquer tipo comparável
  - ValueType pode ser de qualquer tipo, inclusive um mapa
- Função make deve ser utilizada para inicializar um mapa

```
m = make(map[string]int)
```

Mapas - Exemplo

```
package main
import "fmt"
func main() {
//Cria mapa vazio
	m :=
make(map[string]int)
//Adiiciona valor
	m["k1"] = 7
	m["k2"] = 13
//Imprime
	fmt.Println("map:", m)
//Acesso
	v1 := m["k1"]
	fmt.Println("v1: ", v1)
// 'len' retorna o número de
pares no mapa
	fmt.Println("len:", len(m))
// Remove um par
	delete(m, "k2")
	fmt.Println("map:", m)
//Acesso retorna se chave
existe
	_, prs := m["k2"]
	fmt.Println("prs:", prs)
// Inicialização direta
	n := map[string]int{"foo":
1, "bar": 2}
	fmt.Println("map:", n)
}
```

Ponteiros

- Ponteiros inferidos ou declarados com *tipo
- Acesso ao endereço de uma variável com &
- Para acessar o valor apontado, utiliza-se *ponteiro
- Ressalva: não é permitido realizar aritmática de ponteiros

  ptr++, ptr+=1 ou outros não funcionam - *Confiabilidade*

Ponteiros - Exemplo

```
package main
import "fmt"
func zeraPorValor(ival int) {
	ival = 0
}
func zeraPorReferencia(iptr *int) {
	*iptr = 0
}
func main() {
	i := 1
	ptr := &i
	fmt.Println("inicial:", i)
	zeraPorValor(i)
	fmt.Println("porValor:", i)
	zeraPorReferencia(ptr)
	fmt.Println("porReferencia:", i)
	fmt.Println("Acesso usando o ponteiro:",*ptr)
	fmt.Println("ponteiro:", ptr)
}
```

Resultado:

inicial: 1

porValor: 1

porReferencia: 0

Acesso usando o ponteiro: 0

ponteiro: 0xc0820022d0

Estruturas

- Declaração semelhante a C
- Acesso a campos usando '.' (ponto)

```
package main
import "fmt"
type person struct {
        name string
        age int
}
func main() {
// Criação de uma nova estrutura
        fmt.Println(person{"Bob", 20})
// Construtor aceita passagem de parâmetros por nome
        fmt.Println(person{name: "Alice", age: 30})
// Campos omitidos recebem valor-zero
        fmt.Println(person{name: "Fred"})
// Acesso feito por meio de '.'
        s := person{name: "Sean", age: 50}
        fmt.Println(s.name)
// Mesmo com ponteiros usa-se '.'
        sp := &s
        fmt.Println(sp.age)
// Structs são mutáveis
        sp.age = 51
        fmt.Println(sp.age)
}
```

Resultado:

```
{Bob 20}
{Alice 30}
{Fred 0}
Sean
50
51
```

Métodos

- Funções que operam em tipos específicos
- Não existe “this” ou “self”
- Métodos podem ser definidos para ponteiros ou “tipos de receptores” (receiver types)
- Cláusula receptador - indica em que tipo de objetos eles operam

```
package main
import "fmt"
type rect struct {
        width, height int
}
func (r *rect) area() int {
        return r.width * r.height
}
func (r rect) perim() int {
        return 2*r.width + 2*r.height
}
func main() {
        r := rect{width: 10, height: 5}
        fmt.Println("area: ",r.area())
        fmt.Println("perim:",r.perim())
        rp := &r
        fmt.Println("area: ",rp.area())
        fmt.Println("perim:",rp.perim())
}
```

Resultado:
area: 50
perim: 30
area: 50
perim: 30

Interfaces

- Definição: conjunto de assinaturas de métodos
- Marca da orientação a objetos de Go
- Um valor de tipo interface pode conter qualquer valor que implemente aqueles métodos
- Interfaces são implementadas implicitamente
- Não há herança, subclasse ou a palavra-chave "implements"
- Polimorfismo: inclusão

Interfaces - Exemplo

```
package main
import ("fmt","math")

type Abser interface {
	Abs() float64
}

func main() {
	var a Abser
	f := MyFloat(-math.Sqrt2)
	v := Vertex{3, 4}

	a = f  // a MyFloat implements Abser
	a = &v // a *Vertex implements Abser
	// A linha a seguir gera um erro, pois v é um Vertex (não *Vertex) e não implementa Abser
	a = v
	fmt.Println(a.Abs())
}

type MyFloat float64

func (f MyFloat) Abs() float64 {
	if f < 0 {
		return float64(-f)
	}
	return float64(f)
}

type Vertex struct {
	X, Y float64
}

func (v *Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X+v.Y*v.Y)
}
```

Resultado: 5

Goroutines

- Forma de implementação de paralelismo
- Comando colateral de Go: go
- Executa a função passada como parâmetro em um thread paralelo
- Comando defer adia a execução da função até que a função que a chamou termine

Goroutines - Exemplo

```
package main
import "fmt"
func f(from string) {
	for i := 0; i < 3; i++ {
		fmt.Println(from, ":", i)
	}
}
func main() {
	f("direct")
	go f("goroutine")

	go func(msg string) {
		fmt.Println(msg)
	}("going")

	var input string
	fmt.Scanln(&input)
	fmt.Println("done")
}
```

Resposta:
direct : 0
direct : 1
direct : 2
goroutine : 0
going
goroutine : 1
goroutine : 2
<enter>
done

Canais (Channels)

- Mecanismo de comunicação entre goroutines
- Send e receive bloqueantes - comunicação apenas quando ambos os lados estão prontos
- Sintaxe:
  - Criação:

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

  - Comunicação:

```
ch <- v    // Envia v ao channel ch.
v := <-ch  // Recebe de ch e  armazena o valor em v.
```

Canais com Bufer (Buffered Channels)

- Só bloqueiam quando o buffer está cheio

```
package main
import "fmt"
func main() {
	messages := make(chan string,
2)
	messages <- "buffered"
	messages <- "channel"
	fmt.Println(<-messages)
	fmt.Println(<-messages)
}
```

Canais - Direção

- Canais em parâmetros para funções podem ter direção especificada através do operador <-
- Caso não seja especificada, o compilador interpreta como bidirecional
- Tentar fazer uma operação na direção contrária gera erro de compilação

```
package main
import "fmt"
func ping(pings chan<- string,
msg string) {
	pings <- msg
}
func pong(pings <-chan string,
pongs chan<-
	string) {
	msg := <-pings
	pongs <- msg
}
func main() {
	pings := make(chan string,
1)
	pongs := make(chan string,
1)
	ping(pings, "passed
message")
	pong(pings, pongs)
	fmt.Println(<-pongs)
```

Canais - Select

- Semelhante a um switch em C
- Permite executar tratamentos diferentes para dados recebidos de canais diferentes
- Útil em programação multi-thread

```
package main
import "time"
import "fmt"
func main() {
        c1 := make(chan string)
        c2 := make(chan string)
        go func() {
                time.Sleep(time.Second * 1)
                c1 <- "um"
        }()
        go func() {
                time.Sleep(time.Second*2)
                c2 <- "dois"
        }()
        for i := 0; i < 2; i++ {
                select {
                case msg1 := <-c1:
                        fmt.Println("Recebido de
c1:", msg1)
                case msg2 := <-c2:
                        fmt.Println("Recebido de
c2:", msg2)}}}
```

Canais - Range

- É possível usar range para iterar sobre elementos de um channel
- A função *close* fecha um canal. É necessário utilizá-la antes de utilizar um canal num *range* (*deadlock*)
- Apos *close*, dados continuam disponíveis no canal

```
package main
import "fmt"
func main() {
	queue := make(chan string, 2)
	queue <- "one"
	queue <- "two"
	close(queue)
	for elem := range queue {
		fmt.Println(elem)
	}
}
```

Resultado:
one
two

CARACTERÍSTICAS DA LINGUAGEM

- Aplica os pilares da programação orientada a objetos: encapsulamento, herança e polimorfismo

Sobrecarga de métodos e operadores

- Go não permite a sobrecarga de operadores
  - Justificativa: sobrecarga de operadores é questão de conveniência e adiciona complexidade desnecessária
- Sobrecarga de métodos também não é permitida
  - Questão de simplificação
  - Ter uma variedade de métodos com o mesmo nome e assinatura diferente pode ser útil, mas adiciona complexidade desnecessária

Escopo

- Variáveis: escopo estático, delimitado por {} (chaves) - Não pode abrir chaves depois de newline ;
- Variáveis definidas em blocos internos nãio são visíveis em blocos externos;
- Funções, structs, constantes e variáveis globais:
  - Identificador inicia em letra maiúscula: acessível fora do pacote (exportado)
  - Identificador inicia em letra minúscula: não é exportado
  - Substitui uso de public e private - especificação é implícita
- Todas as funções de pacotes padrão de Go tem identificador iniciado em letra maiúscula;

Escopo - Exemplo

```
package main
import "fmt"
// global variable declaration
var g int = 20

func main() {
   //local variable declaration
      fmt.Printf ("value of g =
%d\n", g)
      var g int = 10
      fmt.Printf ("value of g =
%d\n", g)
}
```

Resultado:

value of g = 20

value of g = 10

Tempo de vida e o Coletor de lixo

- O coletor de lixo se encarrega de desalocar espaço de memória não mais utilizado
- Usa uma versão melhorada do algoritmo marcar-varrer
- O coletor de lixo tem controle sobre o tempo de vida de variáveis e estruturas
- Go prioriza baixa latência do coletor de lixo (~10ms em Go 1.5)

Tipagem

- Tipagem estática, o tipo da variável não pode ser mudado em outro ponto do programa
- O tipo pode ser declarado ou inferido a partir do contexto

```
package main
import "fmt"
type pessoa struct {
        nome string
        idade int
}
func main() {
        var p1 pessoa = pessoa{"Joaozinho",10}
        p := pessoa{"Pedrinho",11}
        p = "Biscoito"
        fmt.Println(p1)
        fmt.Println(p)
}
```

Não compila > ‘p := “biscoito”’

Tipos primitivos

bool	**byte**	**complex64**	**complex128**	**error**	**float32**	**float64**
int	**int8**	**int32**	**int64**	**rune**	**string**	**uintptr**
uint	**uint8**	**uint16**	**uint32**	**uint64**		

- Go é formatado usando-se UTF-8, e suporta “code-points” em Unicode (rune)
- Todos os tipos tem um valor-zero associado (int: 0, string: ””, bool: false)
- int tem o mesmo tamanho de uint, que pode ser 32 ou 64 bits.
- Floats são codificados usando-se IEEE-754
- uintptr: tem o tamanho necessário para representar um endereço de memória
- byte: apelido para uint8
- complex64 tem partes real e complexa do tipo float32
- complex128 usa float64
- Go não tem: char, decimal, enum ou void

Tipos compostos

- Como citamos anteriormente temos arrays, slices e mapas
- Não tem conjunto potência e união
- Não suporta estruturas recursivas, mas podemos contornar isso com ponteiros, como no exemplo abaixo

ERRO!!

```
type pessoa struct {
      irmao pessoa
      nome string
      idade int
}
```

Forma correta:

```
type pessoa struct {
      irmao *pessoa
      nome string
      idade int
}
```

Constantes e tipos Enumerados

- Go possui constantes: utiliza-se a palavra-chave *const* para declarar
- Go não tem tipos Enumerados de forma explícita
- Usa constante e iota para faze-lo, iota é do tipo int e é resetado para 0 a cada bloco de declaração de constante
- O problema é saber o intervalo de valores válidos

Constantes e tipos Enumerados - Exemplo

```go
type Season uint8
const (
	Spring = Season(iota)
	Summer
	Autumn
	Winter
)
func (s Season) String() string {
	name := []string{"spring", "summer", "autumn", "winner"}
	i := uint8(s)
	switch {
	case i <= uint8(Winter):
		return name[i]
	default:
		return strconv.Itoa(int(i))
	}
}
```

```go
func main() {
	const n = 500000000
	const d = 3e20 / n
	fmt.Println(d)
	fmt.Println(int64(d))
	s := Summer
	fmt.Println(s)
	s = Season(9)
	fmt.Println(s)
}
```

Resultado:
6e+11
600000000000
summer
9

Tipagem forte e conversão

- Go é fortemente tipada, todo e qualquer erro de tipo é detectado em tempo de compilação e de execução
- Não existe coerção, a conversão de tipos é explicita

```
package main
import ("fmt","math")
func main() {
	var x, y int = 3, 4
	//sqrt espera float64
	var f float64 = math.Sqrt(float64(x*x + 
y*y))
	var z uint = uint(f)
	fmt.Println(x, y, z)
}
```

Tipagem forte e o “void” de Go

- Interfaces são implementadas implicitamente, todos os tipos implementam a “interface vazia”, podemos usar interface{} para referirmos qualquer tipo de dado
- Type casts de interface{} são feitos por meio de type assertions
  - Type assertion de uma variável x para o tipo T: x.(T)
  - O que nos retorna 2 valores, o primeiro é do tipo T e o segundo é do tipo booleano
  - Se a informação de x é do tipo T, então o valor convertido para T será retornado e true, se não retornará o valor-zero do tipo T e false

Tipagem forte e o “void” de Go - Exemplo

```
package main
import "fmt"
func main() {
	array :=
[5]interface{}{1,"matheus",1.56,fmt.Println,true}
	for _,k := range array {
//type assertion do tipo int
		if _ , ehString := k.(int) ; ehString
{
			fmt.Println(k,"eh um int")
		}else{
			fmt.Println(k,"nao eh um int")
		}
	}
}
```

Resultado:

1 eh um int

matheus nao eh um int

1.56 nao eh um int

0x45aea0 nao eh um int

true nao eh um int

Persistência e Serialização

- Go apresenta biblioteca para interação com bancos de dados: database/sql;
- Suporta Postgres, MySQL, Oracle, DB2, MS SQL Server, entre outros;
- Além disso, também implementa serialização através da biblioteca encoding, com suporte a vários de tipos de codificação: base64, binary, CSV, JSON, XML, e mais.

Alocação de memória

- Existem duas primitivas para alocação dinâmica de memória: new e make
  - Make só pode ser usada para inicializar arrays, slices, mapas e channels, e retorna uma estrutura alocada dinamicamente
  - New é usada pra todo o resto dos casos e retorna um ponteiro para uma posição de memória (toda nula)

```
package main
func main() {
// allocates slice structure; *p == nil; rarely useful
        var p1 *[]int = new([]int) var v1 []int = make([]int, 100)
// the slice v now refers to a new array of 100 ints
// Desnecessariamente complexo:
        var p2 *[]int = new([]int)
        *p2 = make([]int, 100, 100)
// Idiomático:
        v2 := make([]int, 100)
}
```

Curto-circuito e Efeitos colaterais

- Go tem curto-circuito em expressões condicionais apenas
- Funções tem efeitos colaterais, um exemplo é a leitura de um arquivo ou do terminal: avança o cursor automaticamente
- ++ e -- com expressões não são permitidos
  - -- e ++ pré-fixados não tem sentido, logo não existem

Curto-circuito e Efeitos colaterais - Exemplo

```
package main
import ("fmt")
func main() {
	var a = 2
	var b = 10
	a++
	fmt.Println(a)
	a--
	fmt.Println(a)
// Erro: syntax error: unexpected --, expecting )
	var c = (a--)*b
// Erro: syntax error: unexpected ++, expecting )
	if (a < b) || (a == b++) {
		fmt.Println(true)
	}
}
```

Modularização

- Funções podem ser criadas em arquivos separados e todo arquivo tem um pacote(package) correspondente
- Chamadas externas de funções e estruturas de um pacote são feitas usando-se:

  <nome do pacote>.<identificados da função/estrutura

  - Exemplo: fmt.Println()
- Obs.:
  - A função main fica no pacote de mesmo nome
  - A passagem de parâmetros é somente por cópia
  - A momento da passagem de parâmetros é normal

“Herança”

- Go não permite herança, pois não é OO
  - Consegue simular com composição por meio de campos anônimos
- É considerado uma implementação implícita como Duck typing,
  - Se faz “quack” como um pato, e anda como um pato, então provavelmente é um pato

“Herança” - Exemplo

```
ackage main
import "fmt"
type Animal struct {
	especie string
}
func (a Animal) String() string {
	return fmt.Sprint("Eu sou um
",a.especie)
}
type Cachorro struct {
	Animal
	raça string
}
func (c Cachorro) String() string {
	return fmt.Sprint("Eu sou um
",c.raça)
}
```

```
func main() {
	c := Cachorro{Animal:
Animal{"cachorro"},
	raça: "€?Pastor Alemão"}
	fmt.Println(c.especie)
	fmt.Println(c.raça)
	fmt.Println(c.Animal)
	fmt.Println(c)
}
```

Resultado:
cachorro
Pastor alemão
Eu sou um cachorro
Eu sou um Pastor
Alemão

Exceções

- Go não tem implementação de um sistema de exceções, ela usa seu poder de múltiplos retornos
- Erros são o último parâmetro de retorno de uma função e implementam a interface error
- Se uma função retornar mais de um erro, é usado polimorfismo e type assertion

Tratamento de erros

```
package main
import "errors"
import "fmt"
func alistarNoExército(idade int)
(string, error) {
        if idade < 18 {
                return "Rejeitado" ,
&NovoDemais{idade}
        }else if idade > 100{
                return "Rejeitado",
errors.New("Não queremos
                        ninguém com mais de
100 anos")
        }
        return "Bem-vindo", nil
}
type NovoDemais struct {
        idade int
}
func (nd *NovoDemais) Error() string {
        return fmt.Sprintf("Faltam %d anos
para você poder
                se alistar.", 18 - €?
nd.idade)
}
func main() {
        c := [3]int{12,150,20}
        for i := range c {
                if r , ok :=
alistarNoExército(c[i]) ; ok != nil {
                        fmt.Println(r,"->
causa:",ok)
                }else{
                        fmt.Println(r)
                }
        }
}
```

Resultado:

Rejeitado -> causa: Faltam 6 anos para você poder se alistar.
Rejeitado -> causa: Não queremos ninguém com mais de 100 anos
Bem-vindo

Bibliotecas

- Go tem MUITAS bibliotecas já embutidas na linguagem
- Dentre elas, podemos citar:
  - compress: contém funções para compactação de arquivos
  - crypto: implementação de diversos tipos de criptografia
  - sync: ferramentas para sincronização (mutexes, semáforos, etc)
  - flag: funções para tratamento de flags de entrada de programas
  - image: biblioteca para lidar com imagens 2D (ex. JPEDG, GIF e PNG)
  - math: funções matemáticas e big numbers
  - net: funções para implementação de protocolos de internet
  - os: interação com o SO (qualquer SO)
  - Dentre muitas outras…

Características OO

- Apesar de ser estruturado, como já mencionado , Go consegue simular características OO e melhor do que C
  - Classes: structs com métodos
  - Encapsulamento: variáveis exportadas ou não
  - Modificadores de acesso: apenas package private ou public
  - Herança: ao invés de herança, composição (campos anônimos)
  - Polimorfismo: através de interfaces

Avaliação da linguagem

Critério	C	Java	Go
Aplicabilidade	Sim	Sim	Sim
Confiabilidade	Não	Sim	Parcial
Aprendizado	Não	Não	Sim
Portabilidade	Não	Sim	Sim
Método de projeto	Estruturado	OO	Estruturado
Evolutibilidade	Não	Sim	Sim
Reusabilidade	Parcial	Sim	Parcial
Integração	Sim	Parcial	Parcial
Custo	Depende da aplicação	Depende da ferramenta	Depende da ferramenta
Escopo	Sim	Sim	Sim
Expressões e comandos	Sim	Sim	Sim
Tipos primitivos e compostos	Sim	Sim	Sim
Persistência	Diversas formas	Biblioteca de funções	JDBC, biblioteca de casses, serialização

Critério	C	Java	Go
Encapsulamento de proteção	não	Sim	Sim
Sistema de tipos	Não	Sim	Sim(fortemente tipada)
Verificação de tipos	Estática	Estática/Dnâmica	Estática/Dinâmica
Polimorfismo	Coerção e sobrecarga	Todos	inclusão
Exceções	Não	Sim	Não
Concorrência	Não	Sim	Sim
Eficiência	Sim	parcial	Sim

REFERÊNCIAS BIBLIOGRÁFICAS

- The Go Blog: <https://blog.golang.org/>
- A Tour of Go: <https://tour.golang.org/>
- The Go FAQ: <https://golang.org/doc/faq>
- Go by Example: <https://dlintw.github.io/gobyexample/public/index.html>