Analise técnica - Golang

Linguagens de Programação - UFES@2020/1

Alunos: Gabriel Cipriano, Ivo Pedrini & Pedro Fontes

1. Como instalar e começar a usar?

   O download do instalador pode ser feito em https://golang.org/doc/install.
   Para instalar em Windows ou Mac execute o instalador baixado,para a versão em Linux é necessário adicionar o caminho /usr/local/go/bin às variáveis do ambiente:

   ```
   export PATH=$PATH:/usr/local/go/bin
   ```

   Para checar se a instalação foi realizada com sucesso abra o terminal e execute o comando:

   ```
   go version
   ```

   Uma vez que tenha go instalado, para rodar um arquivo .go basta digitar no terminal:

   ```
   go run arquivo.go
   ```

   Para compilá-lo como um arquivo executável exec:

   ```
   go build -o exec arquivo.go
   ```

2. Quais são os processos de tradução utilizados?

   Códigos em go são compilados.

3. Em que paradigmas se encaixa?

   Go é uma linguagem Multi-paradigmas que se encaixa em Concorrente, Estruturada e Imperativa. Apesar de possuir funcionalidades de programação Funcional e de Orientada a Objetos, ainda se discute se Go pode se encaixar nesses paradigmas.

4. Os nomes são sensíveis à capitalização?

   Sim. Inclusive,, tipos, funções e etc que possuem um nome que inicia com caixa alta são visíveis para outros pacotes, enquanto nomes que iniciam com caixa baixa não são.

5. Quais os caracteres aceitos em um nome?

   Um nome pode ser composto por diferentes letras e números, devendo obrigatoriamente começar com uma letra. Go considera como letra as categorias *Lu, Ll, Lt, Lm,* e *Lo* presentes no Padrão Unicode 8.0, e considera como numeral a categoria *Nd* presente no Padrão Unicode 8.0.

6. Existe alguma restrição de tamanho para nomes?

   Não existe limite.

7. Como é a questão das palavras-chave x palavras reservadas?

   Go trata palavras-chaves da mesma forma que palavras reservadas, não podendo serem utilizadas como identificadores.

8. É possível definir uma variável anônima? Mostre exemplo.

Sim, inclusive utilizamos para descartar o Cabeçalho do arquivo csv de entrada do trabalho. Uma vez que não nos importa a linha retornada por Read(), utilizamos a variável anônima i.e " _ " para descartá-la.

```
func descartaCabecalho(reader *csv.Reader) {
     _, err := reader.Read()
    utils.CheckError(err) }
```

9. A vinculação de tipos (tipagem) é estática ou dinâmica?

A vinculação de tipos de Go é estática, isto é, feita em tempo de compilação.

10. Quais categorias de variável (Sebesta, Seção 5.4.3) apresenta? Mostre exemplos.

Variáveis estáticas:

```
package main
import "fmt"
//Consts são estáticas
const s string = "aqui tem constante"
func main() {
    fmt.Println(s)
    const n int = 20
    fmt.Println(n)
}
```

Variáveis dinâmicas do monte explícitas:

```
package main
import "fmt"

//Dados não-primitivos (como []string são acessados por meio de variáveis de referência
func main() {
   frutas [ ]string
   frutas = append(frutas, "Maçã")
   frutas = append(frutas, "Banana")
}
```

Variáveis dinâmicas da pilha:

```
package main
import "fmt"
//variáveis declaradas em métodos, por exemplo, são dinâmicas da pilha.
func (p *Pessoa)printa() {
     hello := "Hello "
     fmt.Println("%s %s", hello, p.nome)
}
```

11. Permite ocultamento de nomes (variáveis) em blocos aninhados? Mostre exemplo.

Sim.

```go
func main() {
    n := 0
    if true {
        n := 1
        n++
    }
    fmt.Println(n) // 0
}
```

12. Permite definir constantes? Vinculação estática ou dinâmica? Mostre exemplos.
    Sim, vinculação estática.

```go
package main
import "fmt"

const s string = "aqui tem constante"
func main() {
       fmt.Println(s)
       const n = 20
       fmt.Println(n)
}
OUTPUT:
aqui tem constante
20
```

13. Quais os tipos oferecidos? Mostre exemplos de definição de variáveis de cada tipo.
    - Tipo Booleano
        - booleano: `var b bool = true`
    - Tipo Numérico

```go
unsigned:   var x uint         = 4294967295
unsigned8:  var x uint8        = 255
unsigned16: var x uint16       = 65535
unsigned32: var x uint32       = 4294967295
unsigned64: var x uint64       = 18446744073709551615
integer:    var x int          = 2147483647
integer8:   var x int8         = 127
integer16:  var x int16        = 32767
integer32:  var x int32        = 2147483647
integer64:  var x int64        = 9223372036854775807
float32:    var x float32      = 21474.83647
float64:    var x float64      = 922337203.6854775807
complex64:  var x complex64    = 21474 + 83647i
complex128: var x complex128   = 922337203 + 6854775807i
```

```
byte:        var x byte          = 255
rune:        var x rune          = '本'
```

- Tipo String: `var nome string = "Vitor"`
- Tipo Array: `numeros := [5]int{1,2,3,4,5}`
- Tipo Slice: `var frutas []string`
- Tipo Struct:

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

- Tipo Pointer: `var p *int`
- Tipo Function

```
func SimpleFunction() {
    fmt.Println("Hello World")
}
```

- Tipo Interface

```
type   Shape        interface {
    Area()          float64
    Perimeter()     float64
}
```

- Tipo Map: `var  m  map[string]int`
- Tipo Channel: `messages := make(chan string)`

14. Existe o tipo função? São cidadãs de primeira classe? Mostre exemplo.

    Existe. Sim, são de primeira classe.

```
func main() {
      a := func() { fmt.Println("I'm a first class function") }
      a()
      fmt.Printf("%T", a) //%T exibe o tipo da variável
}
Output: "I'm a first class function"
        func
```

15. Possui ponteiros ou referências? Permite aritmética de ponteiros?

    Go possui ponteiros e não permite sua aritmética na biblioteca padrão.

16. Oferece coletor de lixo? Se sim, qual a técnica utilizada?

    Sim, a técnica utilizada é a de mark-and-sweep, que consiste em varrer os objetos marcando cada um que é referenciado de alguma forma com uma flag. Após varrer todos os que não estiverem referenciados são removidos.

17. É possível quebrar seu sistema de tipos (forçar erro de tipo)? Mostre exemplo.

De certa forma, sim. A biblioteca padrão ***unsafe*** possui ponteiros com um comportamento bem parecido de c, e assim como em c, uma atribuição indevida de ponteiro pode quebrar o sistema de tipos de Golang. O pacote padrão ***unsafe*** se chama assim justamente por colocar o sistema de tipos em risco se não for usada com cautela.

```
package main
import "unsafe"
func main() {
	var x int8 = 10
	var y int32
	//y = x //essa atribuição não é permitida pelo sistema de tipos
	y = *(*int32)(unsafe.Pointer(&x))
	println(y)
}
Output: 98588682
```

18. Quais os operadores oferecidos? Mostre exemplo de uso de cada operador.

Soma: `soma := 1 + 2`	Subtração: `sub := 2-1`	Multiplicação: `mult := 2 * 1`

Divisão:
`div := 1.0 / 2.0`
	Módulo:
	`mod:= 1 % 10`
		Incremento:
		`n := 1`
		`n++`

Decremento:
`n := 2`
`n--`
	Comparação:
	`comp := 5 == 5`
	`fmt.Println(comp)`
	`output = true`
		Diferença:
		`dif := 4 != 5`
		`fmt.Println(dif)`
		`output = true`

Maior:
`maior := 5 > 4`
`fmt.Println(maior)`
`output = true`
	Menor:
	`menor := 5 < 5`
	`fmt.Println(menor)`
	`output = false`
		Maior ou igual:
		`maigual := 4 >= 5`
		`fmt.Println(maigual)`
		`output = false`

Menor ou igual:
`meigual := 5 <= 4`
`fmt.Println(meigual)`
`output = false`
	Conjunção:
	`conj:= 5 == 5 && 2 > 1`
	`fmt.Println(conj)`
	`output = true`
		Disjunção:
		`disj:= 5 == 5 || 2 < 1`
		`fmt.Println(disj)`
		`output = true`

Negação:
`verdad := true`
`negacao = !verdad`
`fmt.Println(verdad)`
`output = false`
	Atribuição:
	`nome := "Vitor"`
	`nome = "Ivo"`
		Soma e atribuição:
		`num := 10`
		`num += 10`
		`fmt.Printf("%d", num)`
		`output: 20`

```
Subtração e
atribuição:
num := 10
num -= 10
fmt.Printf("%d",num)
output: 0
```

```
Multiplicação e
atribuição:
num := 10
num *= 10
fmt.Printf("%d", num)
output: 100
```

```
Divisão e atribuição:
num := 10
num /= 2
fmt.Printf("%d", num)
output: 5
```

```
Resto e atribuição:
num := 17
num %= 2
fmt.Printf("%d", num)
output: 1
```

19. Permite sobrecarga de operadores? Mostre exemplo.

    Sim, assim como em C, go embute sobrecarga em seus operadores, mas os programadores só podem usar essa sobrecarga, sem poder implementar novas sobrecargas de operadores.

```go
package main
import "fmt"
func main() {
   a := 5 + 5 //int
   b := 5.5 + 5.5 }//float
```

20. Quais operadores funcionam com avaliação em curto-circuito?

    Quando falamos em operadores lógicos em golang o operando mais a direita é avaliado condicionalmente, então tanto em Conjunção quanto em Disjunção.

21. O operador de atribuição funciona como uma expressão?

    Sim.

22. Quais as estruturas de controle (seleção, iteração) oferecidas? Mostre exemplos.

    If, for e switch.

```go
nome := "Vitor"

if nome == "Vitor" {
     fmt.Println("NOTA  10
🙏")
}else{
  fmt.Println("NOTA 9.8")
}
```

```go
nota := 0
for i := 0; i < 10; i++{
  nota++
}
for nota <= 10 {
  nota++
}
for {
   fmt.Println("Ao  infinito
e além")
}
```

```go
nome := "Vitor"

switch{
case nome == "Vitor":
  fmt.Println("NOTA 10🙏")
default:
fmt.Println("NOTA 9.8")
}
```

23. Quais sentenças de desvio incondicional oferecidas? Mostre exemplos.
- Break
  -
```
OuterLoop:
    for i = 0; i < n; i++ {
        for j = 0; j < m; j++ {
            switch a[i][j] {
                case nil:
                    state = Error
                    break OuterLoop
                case item:
                    state = Found
                    break OuterLoop
            }
        }
    }
```
- Continue
  -
```
RowLoop:
    for y, row := range rows {
        for x, data := range row {
            if data == endOfRow {
                continue RowLoop
            }
            row[x] = data + bias(x, y)
        }
    }
```
- Goto
  -
```
if n%2 == 1 {
    goto L1
}
for n > 0 {
    f()
    n--
L1:
    f()
    n--
}
```

24. Quais os métodos de passagem de parâmetros oferecidos? Mostre exemplos.
    Assim como todas as linguagens da família C, tudo em Go é passado por valor.

25. Permite sobrecarga de subprogramas? Mostre exemplo.

Go não permite sobrecarga de subprogramas.

26. Permite subprogramas genéricos? Mostre exemplo.

Não, mas há um apelo crescente da comunidade para a implementação de tipo genérico dentro da linguagem, e por conta disso a equipe de devs está trabalhando para que num futuro breve essas mudanças estejam implementadas.
source (25/11/2020):
https://go.googlesource.com/proposal/+/refs/heads/master/design/go2draft-type-parameters.md#Generic-types

Assim que uma função genérica em Go se parecerá quando for implementado:

```
// Print prints the elements of any slice.
// Print has a type parameter T and has a single (non-type)
// parameter s which is a slice of that type parameter.
func Print[T any](s []T) {
       //generic code
}
```

27. Como é o suporte para definição de Tipos Abstratos de Dados? Mostre exemplo.

Go suporta a implementação de estruturas. A struct é uma sequência de elementos, chamados de campos, cada qual com seu nome e tipo. Esses nomes podem ser definidos explicitamente ou implicitamente.

```
struct {
    x, y int
    u float32
    A *[]int
    F func()
}
```

28. Permite TADs genéricos/parametrizáveis? Mostre exemplo.

Atualmente, Go não possui suporte abrangente para tipos de dados genéricos. Como foi projetado pensando na simplicidade, na época TADs genéricos foi considerado um acréscimo substancial de complexidade.
O que se aproxima disso em Go é Interfaces, que são usadas para definir um comportamento genérico sem exigir nenhum detalhe de implementação.

Exemplo: Se quisermos ordenar Tipos/Estruturas definidas pelo usuário, podemos utilizar a função Sort da biblioteca padrão sort, mas para isso, o Tipo que queremos comparar deve implementar todos os métodos da interface a seguir:

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

Fizemos isso para ordenar os candidatos por número de votos no trabalho:

```
// Sorting Candidatos by number of votes
type byVotes []Candidato

func (candidatos byVotes) Len() int {
      return len(candidatos)
}
func (candidatos byVotes) Swap(i, j int) {
      candidatos[i], candidatos[j] = candidatos[j], candidatos[i]
}
func (candidatos byVotes) Less(i, j int) bool {
      return candidatos[i].Votos > candidatos[j].Votos
}

//SortByVotes Ordena uma lista de Candidatos por número de votos
func SortByVotes(candidatos []Candidato) []Candidato {
      sort.Sort(byVotes(candidatos))
      return candidatos
}
```

29. Quais as construções de encapsulamento oferecidas? Mostre exemplos.

Go encapsula coisas a nível de pacote. Nomes que começam com letra minúscula são visíveis apenas dentro do pacote. Você pode esconder tudo em um pacote privado e expor apenas tipos, interfaces e funções/métodos específicos.

```
type Fooer interface {
  Foo1()
  Foo2()
  Foo3()
}

type foo struct {
}

func (f foo) Foo1() {
    fmt.Println("Foo1() here")
}

func (f foo) Foo2() {
    fmt.Println("Foo2() here")
}
```

```
func (f foo) Foo3() {
    fmt.Println("Foo3() here")
}

func NewFoo() Fooer {
    return &Foo{}
}
```

30. Quais tipos de polimorfismo suporta? Mostre exemplos.

    Possui Polimorfismo Ad hoc de sobrecarga, como demonstrado na pergunta 19, cujo apesar de estar presente na linguagem não é possível a sua implementação em subprogramas. Além disso também possui o Polimorfismo Inclusão, que como demonstrado no exemplo da pergunta 32 faz com a struct *Gato* possua as características da estrutura *Animal*. Podendo inclusive chamar/sobrescrever métodos da struct *Animal*.

31. Permite herança de tipos? Herança múltipla? Mostre exemplo.

    Go não permite nenhum tipo de herança.

32. Permite sobrescrita de subprogramas? Mostre exemplo.

    Sim, exemplo:

```
type Animal struct{
      nome string
      idade int
}
```

```
type Gato struct{
     Animal
     raça string
 }
```

```
func (a *Animal) DizOi() {
    fmt.Printf("Oi, meu nome é %s, tenho %d anos!", a.nome, a.idade)
}
func (g *Gato) DizOi() {
     fmt.Printf("Oi, meu nome é %s e tenho %d anos e pertenço a %s raça!", a.nome,
a.idade, a.raça)
}
```

33. Permite a definição de subprogramas abstratos? Mostre exemplo.

    Não permite.

34. Oferece mecanismo de controle de exceções? Mostre exemplo.

    Go não oferece um mecanismo de controle de exceções.

35. Possui hierarquia de exceções controlada, como em Java? Qual a raiz?

    Não, ver a resposta 34.

36. Categoriza as exceções em checadas e não-checadas? Como?

Não, ver a resposta 34.

37. Obriga a declaração de exceções lançadas para fora de um subprograma?

Não, ver a resposta 34.

38. Como você avalia a LP usando os critérios do Sebesta (Seção 1.3)?

Simplicidade	Simples.
Ortogonalidade	Muito ortogonal.
Tipos de dados	Amplo suporte.
Projeto de sintaxe	Bem projetado.
Suporte para abstração	Bom suporte.
Expressividade	Expressiva.
Verificação de tipos	Sim, bastante rígida.
Tratamento de exceções	Não possui
Apelidos restritos	Apelidos permitidos.

Podemos ver então com base nas características acima que Go atende plenamente aos critérios de Legibilidade e Facilidade de escrita definidos pelo Sebesta. Go falha em atender plenamente o critério de Confiabilidade por não possuir diretamente um tratamento de exceções, e ser permitido apelidos.

39. Como você avalia a LP usando os critérios do Varejão?

Legibilidade	Boa
Redigibilidade	Boa
Confiabilidade	Razoável. Fortemente tipada, não possui tratamento de exceções mas possui mecanismos de tratamento de erros
Eficiência	Boa. Mais eficiente que Java, menos que C.
Facilidade de aprendizado	Fácil
Ortogonalidade	Muito ortogonal. Poucas exceções aos seus padrões regulares.
Reusabilidade	Razoável. Não possui polimorfismo paramétrico, por exemplo.
Modificabilidade	Boa

Portabilidade	Alta
Implementação	Compilada
Paradigmas	Imperativo (Estruturado, Concorrente)