Seminario de Lenguajes opción Go

Raúl Champredonde

Seminario de Lenguajes opción Go

- Funciones genéricas
- Tipos genéricos

Funciones Genéricos

```go
ints := map[string]int64{
  "first":  34,
  "second": 12,
}
```

```go
floats := map[string]float64{
  "first":  35.98,
  "second": 26.99,
}
```

```go
func SumInts(m map[string]int64) int64 {
  var s int64
  for _, v := range m {
    s += v
  }
  return s
}
```

```go
func SumFloats(m map[string]float64) float64 {
  var s float64
  for _, v := range m {
    s += v
  }
  return s
}
```

Funciones Genéricos

```go
ints := map[string]int64{
  "first":  34,
  "second": 12,
}
```

```go
floats := map[string]float64{
  "first":  35.98,
  "second": 26.99,
}
```

```go
func SumIntsOrFloats[K comparable, V int64 | float64](m map[K]V) V {
  var s V
  for _, v := range m {
    s += v
  }
  return s
}
```

```go
fmt.Printf("Generic Sums: %v and %v\n",
           SumIntsOrFloats(ints),
           SumIntsOrFloats(floats))
```

Funciones Genéricos

```go
func SumIntsOrFloats[K comparable, V int64 | float64](m map[K]V) V {
  var s V
  for _, v := range m {
    s += v
  }
  return s
}
```

Type formal parameters

```go
fmt.Printf("Generic Sums: %v and %v\n",
           SumIntsOrFloats[string, int64](ints),
           SumIntsOrFloats[string, float64](floats))
```

Type actual parameters

Type constraint

Type Parameters

[T any]
[T comparable]
[T int | int16 | int32 | int64 | int8 | float32 | float64]

Tipos Genéricos - Lista

```go
type List[T any] struct {
  first, last *node[T]
}

type node[T any] struct {
  val T
  next *node[T]
}
```

```go
func (l *List[T]) PutOnFront(v T) {
  l.first = &node[T]{v, l.first}
  if l.last == nil {
    l.last = l.first
  }
}

func (l *List[T]) PutOnTail(v T) {
  n := &node[T]{val: v}
  if l.last == nil {
    l.first = n
  } else {
    l.last.next = n
  }
  l.last = n
}
```

```go
func (l *List[T]) GetAll() []T {
  var elems []T
  for e := l.first; e != nil; e = e.next
{
    elems = append(elems, e.val)
  }
  return elems
}
```

```go
func main() {
  list := List[int]{}
  list.PutOnFront(10)
  list.PutOnTail(20)
  list.PutOnFront(30)
  list.PutOnTail(40)
  list.PutOnFront(50)
  list.PutOnTail(60)
  fmt.Println("list:", list.GetAll())
}
// list: [50 30 10 20 40 60]
```

Tipos Genéricos – Árbol binario

```
type Tree[T any] struct
{
  val T
  left, right *Tree[T]
}
```

```
func lt(x, y int) bool
{
  return x <= y
}
```

```
func main() {
  var tree *Tree[int]
  tree = tree.insert(50, lt)
  tree = tree.insert(10, lt)
  tree = tree.insert(90, lt)
  tree = tree.insert(40, lt)
  tree = tree.insert(60, lt)
  tree = tree.insert(30, lt)
  tree = tree.insert(80, lt)
  fmt.Println("Tree:",
tree.GetAll())
}
```

```
func (t *Tree[T]) insert(v T, f func(T, T) bool) *Tree[T]
{
  if t == nil {
    return &Tree[T]{val: v}
  } else {
    if f(v, t.val) {
      t.left = t.left.insert(v, f)
    } else {
      t.right = t.right.insert(v, f)
    }
    return t
  }
}
```

```
  switch {
  case t == nil:
    t = &Tree[T]{val: v}
  case f(v, t.val):
    t.left = t.left.insert(v, f)
  default:
    t.right = t.right.insert(v,
f)
  }
  return t
```

```
func (t *Tree[T]) GetAll() []T {
  var elems []T
  if t != nil {
    elems = append(elems, t.left.GetAll()...)
    elems = append(elems, t.val)
    elems = append(elems,
t.right.GetAll()...)
  }
  return elems
}
```

Interfaces Genéricas

- Investigar

```
type Container[T any] interface {
  Len() int
  Append(T)
  Remove()(T, error)
}
```