Golang generics in 1.18

Will be released in Feb 2022

- Will be 100% back compatible
- But have space for future improvements

1. Type parameters for functions and types

```
func min(x, y float64) float64 {
   if x < y {
      return x
   }
   return y
}
```

```
func min[T constraints.Ordered](x, y T) T {
   if x < y {
      return x
   }
   return y
}
```

```
m := min[int](2, 3)
```

- Instantiation
  - Substitute type arguments for type parameters
  - ⚠️ Checks if type arguments implement their constraints
- Can instantiate func or type

```
fmin := min[float64]
```

```
type Tree[T interface{}] struct{
   left, right *Tree[T]
   data        T
}

func (t *Tree[T]) Lookup(x T) *Tree[T]

var stringTree Tree[string]
```

2. Type sets defined by interfaces

- Type constraints
  - Are interfaces

```
type Ordered interface {
   Integer|Float|~string
}
```

~T means the set of all types with underlying type T

```
[S interface{~[]E}, E interface{}]

[S ~[]E, E interface{}]

[S ~[]E, E any]
```

interface{} wrapper is not needed if just one type

any == interface{}

3. Type inference

```
func min[T constraints.Ordered](x, y T) T

var a, b, m float64

m = min[float64](a, b)

m = min(a, b)
```

Some tricky examples

```
func Scale[E constraints.Integer](s []E, c E) []E {
   r := make([]E, len(s))
   for i, v := range s {
      r[i] = v * c
   }
   return r
}
```

This will not work for Point type derived from Integer (type Point []int32). Scale() returns []int32 - not a Point!

```
func Scale[S ~[]E, E constraints.Integer](s S, sc E) S {
   r := make(S, len(s))
   for i, v := range s {
      r[i] = v * c
   }
   return r
}
```

Fixed Scale() func

👍 When to use generics?

- Functions that work on slices, maps, and channels of any element type.
- General purpose data structures.
  - When operating on type parameters, prefer functions to methods.
- When a method looks the same for all types.

👎 When NOT to use generics?

- When just calling a method on the type argument.
- When the implementation of a common method is different for each type.
- When the operation is different for each type, even without a method.