Why Golang looks like that

Tomasz Grodzki
Co-Founder &
Developer,
AlphaSOC
tomasz@alphasoc.com
https://github.com/tg

2018-12-10

Long time ago...

Java and C++ were used for writing servers (at least at Google).
They requires too much bookkeeping and repetition.
Some programmers moved towards more dynamic languages (Python).
Efficiency and type safety at stake!

Birth

Born out of frustration.
Efficient compilation, efficient execution, or ease of programming?
First sketches in September 2007 (Robert Griesemer, Rob Pike, Ken Thompson).
Plenty of siblings (Rust, Elixir, Swift).

Language goals

- Easy: compiled, statically typed, but with a dynamic feel.
- Modern: support for networked and multicore computing.
- Fast: at most a few seconds to build a large executable on a single computer.

"Go was intended as a language for writing server programs that would be easy to maintain over time." (golang.org)
"Clean procedural language designed for scalable cloud software." (Rob Pike)

Code design principles

- Reduce clutter and complexity.
- Everything is declared exactly once.
- Syntax is clean and light on keywords.
- No hierarchy: types just are, they don't have to announce their relationships.
- Orthogonality of concepts.
- Orthogonality of features.

Where are my features?

Go does not compete on features.
As of Go 1, the language is fixed.
Features add complexity and hurt readability.
Simple features that interact in simple ways.
Simple means readable and predictable.
Simple means easy to maintain.

Simplicity is complicated

Great talk by Rob Pike: "Simplicity is complicated". (I've borrowed a few slides.)
Go is actually complex, but it *feels* simple.
Interacting elements must mesh seamlessly, without surprises.
Requires a lot of design, implementation work, refinement.
Simplicity is the art of hiding complexity.

A few simple things in Go

- garbage collection
- goroutines
- constants
- interfaces
- packages

Each hides complexity behind a simple facade.

(Simple) Garbage collecting

Super simple to use: - no interface - not event mentioned in the spec
Very difficult to implement well.
Shouldn't affect language goals, e.g. compilation time.

(Simple) Concurrency

Start a routine with a go keyword:

```
go function(args...)
```

Fire and forget.
Zero management:

- no stack size
- no routine ID
- no return

(Simple) Constants

Constants are just numbers.
They don't have specific type.
Integers and floating points have "infinite" precision.
Can be used more freely than variables:

```
x := 2.0          // x is float64
fmt.Println(2*x) // prints "4"

y := 2.0 * time.Second // y is time.Duration (int64)
fmt.Println(30 * y)    // prints "1m0s"

z := 2.5 * time.Second // error: constant 2.5 truncated to integer
```

(Simple) Interfaces

Interface is just a set of methods. No data.

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

Statically and dynamically checked:

```
var r Reader = os.Stdin        // statically checked
var x interface{} = os.Stdin  // statically checked
r = x.(Reader)                 // dynamically checked
```

More general than OOP.
Implement Read/Write and take advantage of `package io`.

(Simple) Packages

```
package http
...
import "net/http"
```

Two keywords – `package`, `import`.
Took a long time to design.
Affects program design, syntax, naming, building, linking, testing, ...
Separation of package path from package name enabled `go get`.

Syntax

Why is the syntax so weird?

Only declaration syntax is different (from C-like languages).
Designed to be easy to analyze.
Can be parsed without symbol table.
Much easier to build tools.

C syntax

In C there is not special syntax for declarations.
You write an expression involving the item being declared:

```
int x;
int f(int a, int b);
int (*fp)(int a, int b);
```

If you write “(*fp)(a, b)“ you’ll call a function that returns int, hence `fp` is a pointer to a function.
Gets complicated quite quickly:

```
int (*fp)(int (*f)(int x, int y), int b)
```

Go syntax

Distinct type syntax in declaration.
Similar to many languages outside of the C family.

```
x int
p *int
a [3]int
```

Reads nicely from left to right.
No direct correspondence between the look and use.

Works well for functions functions functions

Simple example is similar to C:

```
func main(argc int, argv []string) int
```

Reads well: function main takes an int and a slice of strings and returns an int. More complex declarations are also readable:

```
f func(func(int,int) int, int) int
f func(func(int,int) int, int) func(int, int) int
```

Who is weird now?

```
f func(func(int,int) int, int) func(int, int) int
```

vs

```
int (*(*fp)(int (*)(int, int), int))(int, int)
```

Go's declarations read left to right.
C's read in a spiral? See `Clockwise/Spiral Rule.`

Invisible semicolon

Semicolons are in the formal grammar.
Good for parsers, not for people.
Injected automatically, without lookahead.
Lack of lookahead has some side effects: - opening brace of a function cannot appear on a line by itself, - fixed style, - easier for tools and interactive implementation of Go.

References

- Rob Pike - Simplicity is complicated
- golang.org/doc/faq