GO PROGRAMMING LANGUAGE WORKSHOP

ARUN DEBRAY
JANUARY 18, 2014

The single most important thing about Go is that it has a really adorable mascot.

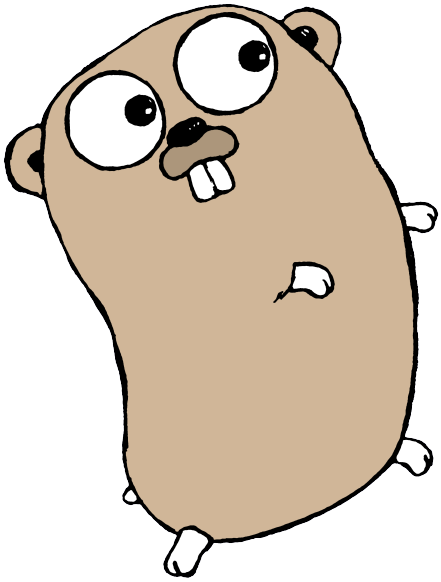

FIGURE 1. The mascot of the Go programming language.

There are lots of different languages, so why another one? Google engineers designed Go as a replacement for C, where we can update it with twenty years' worth of new knowledge. Some languages are beautiful; but Go is designed to be practical. It is designed to scale (e.g. good compiler), but it also scales on other fronts, such as the size of the codebase into $10^5 \sim 10^6$ lines. In fact, the inception for the language was hour-long compile times in C and C++. Go is also designed to scale with the number of developers; for example, there is lots of type safety, etc. But importantly, the language is very small — the complete language-reference is eleven pages, orders of magnitude smaller than that of C++, and it's quite modular.

Interfaces. One can define types as interfaces; the idea here is that "I can do this kind of thing." It's duck typing, so anything that looks like the type and doesn't fail is OK. One doesn't need to declare conformance to interfaces, which is nice when interfacing with third-party code. This can be seen in the following example:

```
type source interface {
    Phrase() string
}

type Rand struct {
    Chars   string
    Length  int
}

func (r *Rand) Phrase() string {
    if r.Chars == "" {
    // Philosophy: the zero value should be useful
        r.Chars = DefaultRandChars
    }
    n := len(r.Chars)
    var b []byte // a slice. This is basically a variable-length array
    for i := 0; i < r.length; i++ {
        b = append(b, r.Chars[rand.Intn(n)])
```

```
    }
    return string(b)
}
```

Another philosophy illustrated by the code is that things should "just work," though there should be a way to optimize it if necessary; thus, we have variable-length slices and fixed-length arrays. The duck typing encourages composition and reuse, so one type can be used for several different applications, which was one of the points of the demo: multiple things could implement `Phrase()`.

```
// Lowercase variables are private and unexported
type truncate struct {
    src Source
    length int
}

// Uppercase variables and functions are exported
// This is a method of the truncate type, with no arguments, returning a string
// truncate satisfies the Phrase interface
func (t *truncate) Phrase() string {
    /* Here, 't' is akin to 'self' or 'this' in other languages.
       It is a Go idiom that the name should have meaning, rather than always
       being the same idea. Ideally, it should also work just the same as a function
       that accepts t as its first argument.
     */
    w := t.src.Phrase()
    if len(w) > t.length {
        w = w[:t.length]
    }
    return w
}

// The language is case-sensitive, so this is different from the type
func Truncate(s Source, length int) Source {
    return &truncate{src: s, length: length} // static initialization, return address
    /* Notice that in C, this would return the address on the stack, which is bad.
       Yet it works in Go, because it was designed to be legal, and to do something
       on the heap because it is an exposed pointer.

       In general, the compiler makes decisions about the stack vs. the heap.
     */
}
```

To elaborate about the stack and heap, the compiler will make placement decisions in order to maximize safety and efficiency, which means there are a lot of magical pointer casts in C that you can't get away with in Go.

Another thing you might have noticed in the code so far is the difference between = and :=. The latter indicates that it is a declaration rather than assigning a value. This is part of Go's philosophy on scaling up to many developers, including forcing all variables and imported libraries to be used.

Interfaces are also really useful for testing. For example, simulating a network is difficult for debugging, but with an interface that's easier to control, one can just write something that "looks like" the input, but without the network unreliability, etc. For example, one could write `type phrase string` and then implement `Phrase()` on it to return just that string. Then, one can test the above functions with this simpler implementation.

Concurrency. Concurrency in most languages is hard to reason about, and tends to generate quite subtle, non-obvious bugs. This is because one has to make sure two different threads don't access the same data at the same time. Go uses the CSP model, which uses some mathematical formalism, and is the first well-known language to do so. The general idea is that there may be several sequential processes running concurrently, but the moments in which they synchronize and interact are carefully placed. For the rest of the time, they should be running completely independently. These user-spaced threads are called goroutines. They're very cheap (so a million are OK, which is *not* true of threads).

Then, one can send data across channels between goroutines in order to momentarily synchronize them. These are like pipes, but strongly typed. The word for channel is `chan`, and `make` is for initializing a built-in type.

```
var pp []*Player
playerc := make(chan *Player)

// Listen for players
for i := 0, i < 2; i++ {
```

```
    // Spin off two goroutines, and continue operating here.
    // func() is the syntax for an anonymous fucntion
    go func() {
        // Listens for inbound connections, get their name, etc.
        // Lots of blocking and network stuff... but totally sequential.
        p, err := AcceptPlayer(listener)
        if err != nil {
            // This is not terribly robust, but it's a demo.
            log.Fatalf("Failed to accept player: %v", err)
        }
        // Send the player object into the player channel, and then stop
        // Go has closures, so it takes playerc with it.
        playerc <- p
    }()
}

// Wait until we have two players
for i := 0; i < 2; i++ {
    // Wait to receive the player from the channel playerc
    p := <-playerc
    // append to the slice
    pp = append(pp, p)
}
```

This style makes it very easy to, for example, fan out lots of requests to servers and wait for them to do something, where the "something" can be made to be completely sequential. Note that the <- operator is sending through a channel, not an assignment.

```
// Send the phrase to all of the players
winnerc := make(chan *Player)

// Throwing away the index value, so that it's purely a 'for-each' loop
// Note that p is scoped only within the for loop, so it has to be redeclared later.
for _, p := range pp {
    /* Anonymous functions are more readable, but when optimizing for performance
       you might want to define it statically. The compiler's pretty good at dealing
       with these sorts of things, though.
     */
    go func(p *Player) {
    // This goroutine leaks; if this weren't a demo, you would want to close it, as
    //       close(winnerc)
        if err := p.Race(w); err != nil {
        // The Race() isn't guaranteed to halt... so you can set a timeout using another channel.
            log.Fatalf("Error from player %v during rage: %v", p, err)
        }
        // err falls out of scope; it was just within the if statement. Block scope options exist.
        winnerc <- p
    /* Passing p into the function helps prevent race conditions; the p
       inside the anonymous function is scoped differently. It's slightly
       confusing, but is a common enough idiom. Sometimes, people have
       arguments (heh) about it, especially in other languages.
     */
    }(p)
}

// The first one whose reply is received wins.
p := <-winnerc
```

If you want an untyped channel, one could define an empty interface, which functions much like a `void*` in C, e.g `make(chan interface{})` (so one can define functions inline). While this comes in handy occasionally, it's generally better to use strongly typed channels.

Another interesting fact is that there are no style arguments, since the toolchain keeps code in one consistent format. Here are some resources:

- `golang.org`, the official website, with a useful low-level tutorial.
- `go-nuts`, a Google group for users of the Go programming language.

The full code for the demo is posted at `https://github.com/josharian/kart`.