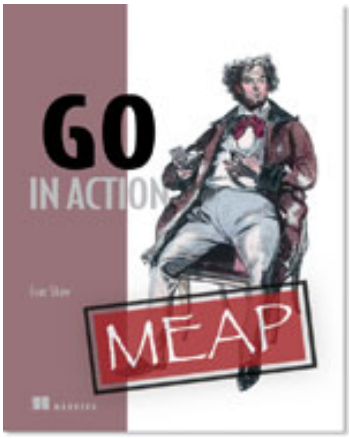
MANNING PUBLICATIONS



Go in Action
By Evan Shaw

Go is a new general purpose programming language that aims to be simple and efficient. It's designed with large applications in mind, but also knows that large applications usually begin life as small applications. In short, it's a language that scales. This green paper, based on Go in Action, discusses at a high level how Go helps programmers build software, especially as it compares to other popular languages.

Why Go?

Over the last several years, many new languages have sprouted up and it's becoming more and more difficult to figure out which ones are worth looking at. Go takes some familiar ideas, cleans them up, and adds some modern features, producing a simple language that's certainly worth considering for your next project.

Simple and concise

Go is designed to be as straightforward as possible. It can be learned quickly and the entire language can easily fit in a person's head. Its features don't interact in strange, unpredictable ways that cause lots of special cases.

In this area, Go's advantages are incremental rather than revolutionary. Individually, they may not seem like much, but together they create a powerful language.

Improved syntax

Go's syntax is cleaner than many other C-like languages. It doesn't require semicolons and needs fewer parentheses in control flow statements.

C declarations are famously difficult to read, especially when the types involved become complex.

```
double *p;
int a[3];
char ** (*foo)(int, int);
```

There are well-known methods of deciphering C type declarations, but the point is that, if there has to be a special technique to read them, something's gone wrong. Go declarations can always be read from left to right.

```
var p *float64
var a [3]int
var foo func(int, int) []string
```

Maps

Go has a built-in associative array type called a map. It's essentially a hash table and it works similarly to a Python dictionary or a Ruby Hash. Here are a couple examples:

```
var indexes map[string]int    #1
squareRoots := map[int]int{   #2
    1: 1,
    4: 2,
    9: 3,
    16: 4,
    25: 5,
}
```

#1 A map of strings to integers
#2 A map of integers to their square roots

Anonymous functions and closures

Sometimes you have a function that only gets used once, often when it's passed to another function. Going off to declare and define this function somewhere else for this single use can be a pain. This is why Go has anonymous functions that can be created right where they're used.

Not only does Go have anonymous functions, but they can capture local variables. Functions that do this are called closures.

Here's a quick example that creates an HTTP handler using an anonymous function.

```
http.HandleFunc("/hello",
    func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, world!")
    }
)
```

Local type inference

In Java, it's not uncommon to see lines of code like this:

```
StringBuilder sb = new StringBuilder();
```

What's with the repetition? Why can't the compiler just figure out that `sb` is a variable of type `StringBuilder` instead of requiring the programmer to write it out twice?

Go fixes this problem through local type inference. The equivalent in Go looks something like this:

```
buf := new(bytes.Buffer)
```

Concurrent

These days, even consumer-grade CPUs often have two or more cores available. However, not many languages are in a good position to take advantage of multiple cores. Some have memory models that don't allow more than one thread to execute at once. Others have no specified memory model at all, which makes writing portable concurrent code an exercise in trial and error.

Go is designed with concurrency in mind. In fact, the Go designers felt concurrency was important enough that they provided easy to use concurrency primitives in the language itself.

Most languages have some kind of threading library available. Each thread runs independently of each other but any mutable values shared between them must be protected by ensuring that only one thread is accessing the value at a time.

This becomes very complicated when there are many shared values or many different types of threads.

Go takes a slightly different approach. Every Go program consists of one or more goroutines. A goroutine is like a thread, but much lighter weight. It's perfectly fine (though perhaps unusual) for an application to be running millions of goroutines.

Sharing mutable values between goroutines directly is discouraged. The more common way to communicate is by using channels to pass values or messages between goroutines. This is a more natural way to think about many concurrent problems.

The go statement

Kicking off a goroutine is as easy as making a function call. Just prefix the call with the word `go`.

```
sum(1, 2, 3)  #1
go sum(1, 2, 3)  #2
```

#1 Call sum
#2 Call sum in a new goroutine

Message passing

Go uses a technique known as message passing to share information between goroutines. A common Go motto is:

> Do not communicate by sharing memory; instead, share memory by communicating.

It's common in many programming languages to share data between threads willy-nilly and then add locks to synchronize. This gets complicated very quickly. Message passing schemes are almost invariably easier to understand and less error prone.

If we wanted to sum numbers in another goroutine by passing messages over a channel, it might look like this:

```
numbers := make(chan int)   #1
go sum(numbers)
for i := 0; i < 10; i++ {   #2
    numbers <- i
}
```

#1 Create a channel.
#2 Pass the numbers 0 through 9.

Safe

Go is a safe language. When we talk about unsafe languages, the usual culprits are C and C++. These languages have a number of corners where undefined behavior lurks. Undefined behavior is always bad news for a programmer, so Go is designed in a way that fixes most of these issues.

Type safety

Many languages are dynamically typed, which can be convenient, but also means that certain errors cannot be caught until the code containing them executes (usually this happens during an important demonstration for your boss). Go is statically typed, which means that type errors are detected when code is compiled.

In addition to being statically typed, Go is strongly typed. This means that types cannot be unsafely coerced into other types that they're not. Numeric values are never converted from one type to another without the programmer giving explicit permission. This is a good thing. Implicit conversions come with complicated rules that are difficult to remember. Anyone who's ever been bitten by a subtle integer overflow bug caused by an implicit conversion can appreciate this feature.

Memory safety

Memory safety means accessing only the memory that belongs to you. Languages that aren't memory safe expose programmers to a variety of possible bugs and security issues.

It's almost impossible to access invalid memory in Go. While it does have pointers, there is no pointer arithmetic. Array accesses have their bounds checked. Garbage collection ensures that there are never dangling pointers and prevents the vast majority of memory leaks.

As nice as it is to have memory safety, there can be benefits to unsafe memory operations. Old school C programmers love to play tricks with C's loose type system to heavily optimize their code. Go can even accommodate this sort of skullduggery, but never by default and only when you, the programmer, are very explicit about it.

Variables are always initialized

In C and C++, variables have an undefined value until they're initialized by the programmer. In Go, all variables are automatically set to a zero value. Forgetting to initialize a variable is not a difficult to diagnose problem like it can be in C or C++ because behavior is consistent on every execution.

Fast

Fast in the context of programming languages usually means that programs written in the language run fast. This is true of Go, but Go can actually be considered fast in several ways.

Runtime

Go is strongly and statically typed, so it's able to compile straight to machine code the same way as C and C++. Programs written in Go do not need to be executed with a virtual machine or interpreter. They start up quickly and continue to run faster than programs written in most other languages.

Compile time

Anyone who's ever worked on a large C or C++ project can tell you about the pains of long compile times. By the time you've compiled your code you've probably become distracted by something else and can't even remember what you're supposed to be doing with your program once it's compiled.

Part of the reason that dynamic languages are popular is because they don't need to be compiled ahead of time. This speeds development because it's easy to iterate quickly. It's only a matter of making a code change and testing it with no long compilation step in between. The downside is that, in broad strokes, interpreted programs are slower than compiled ones and are dynamically typed.

Go is a compiled language, but one of its primary goals is to compile very quickly. Even when compiling large programs, you won't have enough time to space out and start thinking about other things. This helps keep the fast turnarounds of interpreted languages, but also retains the runtime speed and type safety advantages of compiled languages.

Development time

Go's benefits work together to make programming easier. Fast compile times, a rich standard library, and early error detection mean that you'll be able to develop faster. Interface types mean you'll spend less time worrying about how to organize your class hierarchies.

Benefits and limitations of Go

If you've programmed in other languages, you're bound to have certain expectations of what a new language should be able to do. Here I'm going to try to calibrate those expectations and let you know what specific improvements you should notice depending on your background.

Benefits over other static languages

For programmers who are used to static languages, the benefits of Go can be summed up by saying that it feels dynamic. Statically typed languages are often burdened with a great deal of ceremony surrounding everything that happens in a program. This ceremony comes in the form of heavy syntax, type annotations, or complex class hierarchies. Go unencumbers you.

Benefits over dynamic languages

If you're coming from a dynamic language like Python or Ruby, you're probably looking for one thing these languages struggle with: speed. You need look no further. There are a few other parts of Go you may come to love.

As much as possible, Go tries to detect errors when you compile the code rather than when you run it. This means detecting errors sooner. Go puts special effort towards making programmers like you feel comfortable.

Local type inference means fewer type annotations. Fast compile times make sure you never feel slowed down by having to compile your code ahead of time. Interfaces allow a practice similar to the so-called duck typing that's popular in dynamic languages.

A comparison

For a concrete comparison against another language, let's take a look at an echo server written in Java next to one written in Go. The server simply waits for a client connection, reads from the client, and echoes back whatever the client wrote to it. Don't worry, you're not expected to understand this code yet. It's just an attempt to show how Go's little features come together to make programming easier.

First, listing 1 shows the Java version.

Listing 1 An echo server in Java

```
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Echo extends Thread {
    public static void main(String args[]) {
        ServerSocket serverSocket;
        try {
            serverSocket = new ServerSocket(2000);
        } catch (IOException e) {
            System.err.println(e.getMessage());
            return;
        }

        while (true) {
            try {

                Socket clientSocket = serverSocket.accept();   #1
                Echo es = new Echo(clientSocket);

                es.start();                                #2
            } catch (IOException e) {
                System.err.println(e.getMessage());
            }
        }
    }

    private Socket clientSocket;

    public Echo(Socket cs) {
        clientSocket = cs;
    }

    public void run() {
        OutputStream os = null;
        InputStream is = null;
        try {
            byte msg[] = new byte[1024];
            os = clientSocket.getOutputStream();
            is = clientSocket.getInputStream();      #3

            while (true) {
                int n = is.read(msg);
                if (n < 0) {
                    break;
                }
                os.write(msg);
                os .flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
            System.err.println(e.getMessage());
        } finally {
            try {
                if (os != null)
                    os.close();
                if (is != null)
                    is.close();
                clientSocket.close();
            } catch (IOException e) {
                System.err.println(e.getMessage());
            }
        }
    }
}
```

#1 Waits for a client to connect
#2 Starts a new thread

#3 Copies client input to output

Compare that to the Go version in listing 2.

Listing 2 An echo server in Go

```
package main

import (
    “io”
    “log”
    “net”
)

func main() {
    l, err := net.Listen(“tcp”, “:2000”)
    if err != nil {
        log.Fatal(err)
    }
    for {

        conn, err := l.Accept()      #1
        if err != nil {
            log.Fatal(err)
        }

        go func(c net.Conn) {     #2

            n, err := io.Copy(c, c)     #3
            if err != nil {
                log.Println(“error copying:”, err)
            }
            c.Close()
        }(conn)
    }
}
```

#1 Waits for a client to connect
#2 Starts a new goroutine
#3 Copies client input to output

The difference in length between the two examples is apparent, but the Go version is also easier to read. An anonymous function lets us see clearly how the program executes without having to jump around the file. The go statement gives us easy concurrency without having to make a whole class. Go’s library supplies `io.Copy`, which makes the job of copying an input stream to an output stream much easier and also more readable at the same time. The error handling is direct and simple. This is all characteristic of a Go program.

Limitations

While Go is suited for a wide variety of applications, there are some things it doesn’t do well.

METAPROGRAMMING

Metaprogramming is writing code that manipulates other code, or even itself. This is not one of Go’s strong suits. It does not have any type of macro system or generic types or functions. Evaluating code at runtime is not easy and Go is not especially appropriate for creating domain-specific languages.

Go does support limited reflection—that is, examining the structure and methods of a dynamic value at runtime.

REAL-TIME APPLICATIONS

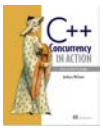
Real-time applications require predictable response times for all operations. Because Go is a garbage-collected language, there are garbage collection pauses that are difficult to predict and can conflict with real-time goals. Research into real-time garbage collection is ongoing and there are solutions available, but not currently for Go. At some point in the future, this story may change and Go might be able to handle real-time requirements.

Summary

Go gives you the best of static and dynamic languages. Its static typing, ahead-of-time compilation model, and control over memory layout keep your code safe and speedy. At the same time, its compilation speed, syntax, and high-level features give you the ability express solutions concisely. It's new, but not revolutionary. It's not meant to be, and hopefully some of the code snippets are already making some sense to you. If not, don't worry. They will soon.

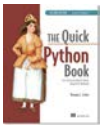
Even if it seems familiar, Go is far from boring. Its designers have gone through great efforts to take the best features and package them in a way that's both sensible and, yes, fun.

Here are some other Manning titles you might be interested in:



C++ Concurrency in Action
Anthony Williams

C# in Depth, Second Edition
Jon Skeet



The Quick Python Book, Second Edition
Vernon L. Ceder

Last updated: August 10, 2012