Preface

> *"Go is an open source programming language that makes it easy to build simple, reliable, and efficient software."* (From the Go web site at `golang.org`)

Go was conceived in September 2007 by Robert Griesemer, Rob Pike, and Ken Thompson, all at Google, and was announced in November 2009. The goals of the language and its accompanying tools were to be expressive, efficient in both compilation and execution, and effective in writing reliable and robust programs.

Go bears a surface similarity to C and, like C, is a tool for professional programmers, achieving maximum effect with minimum means. But it is much more than an updated version of C. It borrows and adapts good ideas from many other languages, while avoiding features that have led to complexity and unreliable code. Its facilities for concurrency are new and efficient, and its approach to data abstraction and object-oriented programming is unusually flexible. It has automatic memory management or *garbage collection*.

Go is especially well suited for building infrastructure like networked servers, and tools and systems for programmers, but it is truly a general-purpose language and finds use in domains as diverse as graphics, mobile applications, and machine learning. It has become popular as a replacement for untyped scripting languages because it balances expressiveness with safety: Go programs typically run faster than programs written in dynamic languages and suffer far fewer crashes due to unexpected type errors.

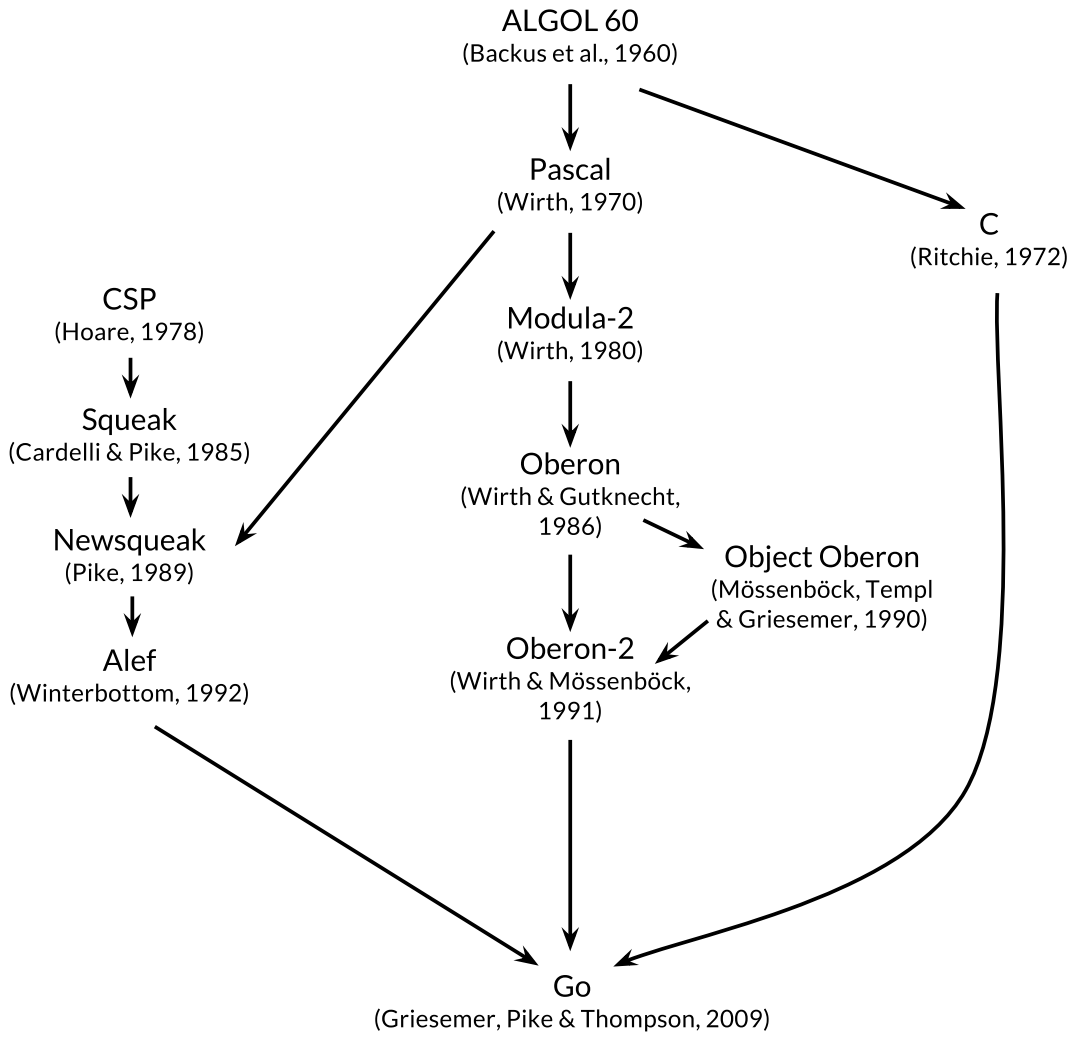
Go is an open-source project, so source code for its compiler, libraries, and tools is freely available to anyone. Contributions to the project come from an active worldwide community. Go runs on Unix-like systems—Linux, FreeBSD, OpenBSD, Mac OS X—and on Plan 9 and Microsoft Windows. Programs written in one of these environments generally work without modification on the others.

This book is meant to help you start using Go effectively right away and to use it well, taking full advantage of Go's language features and standard libraries to write clear, idiomatic, and efficient programs.

The Origins of Go

Like biological species, successful languages beget offspring that incorporate the advantages of their ancestors; interbreeding sometimes leads to surprising strengths; and, very occasionally, a radical new feature arises without precedent. We can learn a lot about why a language is the way it is and what environment it has been adapted for by looking at these influences.

The figure below shows the most important influences of earlier programming languages on the design of Go.



Go is sometimes described as a "C-like language," or as "C for the 21st century." From C, Go inherited its expression syntax, control-flow statements, basic data types, call-by-value parameter passing, pointers, and above all, C's emphasis on programs that compile to efficient machine code and cooperate naturally with the abstractions of current operating systems.

But there are other ancestors in Go's family tree. One major stream of influence comes from languages by Niklaus Wirth, beginning with Pascal. Modula-2 inspired the package concept. Oberon eliminated the distinction between module interface files and module implementation files. Oberon-2 influenced the syntax for packages, imports, and declarations, and Object Oberon provided the syntax for method declarations.

Another lineage among Go's ancestors, and one that makes Go distinctive among recent programming languages, is a sequence of little-known research languages developed at Bell Labs, all inspired by the concept of *communicating sequential processes* (CSP) from Tony Hoare's seminal 1978 paper on the foundations of concurrency. In CSP, a program is a parallel composition of processes that have no shared state; the processes communicate and synchronize using channels. But Hoare's CSP was a formal language for describing the fundamental concepts of concurrency, not a programming language for writing executable programs.

Rob Pike and others began to experiment with CSP implementations as actual languages. The first was called Squeak ("A language for communicating with mice"), which provided a language for handling mouse and keyboard events, with statically created channels. This was followed by Newsqueak, which offered C-like statement and expression syntax and Pascal-like type notation. It was a purely functional language with garbage collection, again aimed at managing keyboard, mouse, and window events. Channels became first-class values, dynamically created and storable in variables.

The Plan 9 operating system carried these ideas forward in a language called Alef. Alef tried to make Newsqueak a viable system programming language, but its omission of garbage collection made concurrency too painful.

Other constructions in Go show the influence of non-ancestral genes here and there; for example `iota` is loosely from APL, and lexical scope with nested functions is from Scheme (and most languages since). Here too we find novel mutations. Go's innovative slices provide dynamic arrays with efficient random access but also permit sophisticated sharing arrangements reminiscent of linked lists. And the `defer` statement is new with Go.

The Go Project

All programming languages reflect the programming philosophy of their creators, which often includes a significant component of reaction to the perceived shortcomings of earlier languages. The Go project was borne of frustration with several software systems at Google that were suffering from an explosion of complexity. (This problem is by no means unique to Google.)

As Rob Pike put it, "complexity is multiplicative": fixing a problem by making one part of the system more complex slowly but surely adds complexity to other parts. With constant pressure to add features and options and configurations, and to ship code quickly, it's easy to neglect simplicity, even though in the long run simplicity is the key to good software.