# Go generate: A Proposal

## Introduction

The `go build` command automates the construction of Go programs but sometimes preliminary processing is required, processing that `go build` does not support. Motivating examples include:

- yacc: generating `.go` files from yacc grammar (`.y`) files
- protobufs: generating .pb.go files from protocol buffer definition (`.proto`) files
- Unicode: generating tables from UnicodeData.txt
- HTML: embedding .html files into Go source code
- bindata: translating binary files such as JPEGs into byte arrays in Go source

There are other processing steps one can imagine:

- string methods: generating `String() string` methods for types used as enumerated constants
- macros: generating customized implementations given generalized packages, such as `sort.Ints` from `ints`

This proposal offers a design for smooth automation of such processing.

## Non-goal

It is not a goal of this proposal to build a generalized build system like the Unix make(1) utility. We deliberately avoid doing any dependency analysis. The tool does what is asked of it, nothing more.

It is hoped, however, that it may replace many existing uses of make(1) in the Go repo at least.

## Design

There are two basic elements, a new subcommand for the go command, called `go generate`, and directives inside Go source files that control generation.

When `go generate` runs, it scans Go source files looking for those directives, and for each one executes a *generator* that typically creates a new Go source file. The `go generate` tool also sets the build tag "`generate`" so that files may be examined by `go generate` but ignored during build.

The usage is:

```
go generate [-run regexp] [file.go...|packagePath...]
```

(Plus the usual `-x`, `-n`, `-v` and `-tags` options.) If packages are named, each Go source file in each package is scanned for generator directives, and for each directive, the specified generator is run; if files are named, they must be Go source files and generation happens only for directives in those files. Given no arguments, generator processing is applied to the Go source files in the current directory.

The `-run` flag takes a regular expression, analogous to that of the `go test` subcommand, that restricts generation to those directives whose command (see below) matches the regular expression.

Generator directives may appear anywhere in the Go source file and are processed sequentially (no parallelism) in source order as presented to the tool. Each directive is a `//` comment beginning a line, with syntax

```
//go:generate command arg...
```

where *command* is the generator (such as yacc) to be run, corresponding to an executable file that can be run locally; it must either be in the shell path (`gofmt`) or fully qualified (`/usr/you/bin/mytool`) and is run in the package directory.

The arguments are space-separated tokens (or double-quoted strings) passed to the generator as individual arguments when it is run. Shell-like variable expansion is available for any environment variables such as `$HOME`. Also, the special variable `$GOFILE` refers to the name of the file containing the directive. (We may need other special variables such as `$GOPACKAGE`. When the generator is run, these are also provided in the shell environment.) No other special processing, such as globbing, is provided.

No further generators are run if any generator returns an error exit status.

As an example, say we have a package "`my/own/gopher`" that includes a yacc grammar in file `gopher.y`. Inside `main.go` (not `gopher.y`) we place the directive

```
//go:generate yacc -o gopher.go gopher.y
```

(More about what "`yacc`" means in the next section.) Whenever we need to update the generated file, we give the shell command,

```
% go generate my/own/gopher
```

or, if we are already in the source directory,

```
% go generate
```

If we want to make sure that only the yacc generator is run, we execute

```
% go generate -run yacc
```

If we have fixed a bug in yacc and want to update all yacc-generated files in our tree, we can run

```
% go generate -run yacc all
```

The typical cycle for a package author developing software that uses `go generate` is

```
% edit …
% go generate
% go test
```

and once things are settled, the author commits the generated files to the source repository, so that they are available to clients that use `go get`:

```
% git add *.go
% git commit
```

## Commands

The `yacc` program is of course not the standard version, but is accessed from the command line by

```
go tool yacc args…
```

To make it easy to use tools like `yacc` that are not installed in `$PATH`, have complex access methods, or benefit from extra flags or other wrapping, there is a special directive that defines a shorthand for a command. It is a `go:generate` directive followed by the keyword/flag "`-command`" and which generator it defines; the rest of the line is substituted for the command name when the generator is run. Thus to define "`yacc`" as a generator command we access normally by running "`go tool yacc`", we first write the directive

```
	//go:generate -command yacc go tool yacc
```

and then all other generator directives using `yacc` that follow in that file (only) can be written as above:

```
	//go:generate yacc -o gopher.go gopher.y
```

which will be translated to

```
	go tool yacc -o gopher.go gopher.y
```

when run.

## Discussion

This design is unusual but is driven by several motivating principles.

First, `go generate` is intended§ to be run by the author of a package, not the client of it. The author of the package generates the required Go files and includes them in the package; the client does a regular `go get` or `go build`. Generation through `go generate` is *not* part of the build, just a tool for package authors. This avoids complicating the dependency analysis done by Go build.

Second, `go build` should never cause generation to happen automatically by the client of the package. Generators should run only when explicitly requested.

Third, the author of the package should have great freedom in what generator to use (that is a key goal of the proposal), but the client might not have that processor available. As a simple example, if it is a shell script, it will not run on Windows. It is important that automated generation not break clients but be invisible to them, which is another reason it should be run only by the author of the package.

Finally, it must fit well with the existing go command, which means it applies only to Go source files and packages. This is why the directives are in Go files but not, for example, in the `.y` file holding a yacc grammar.

§ One can imagine scenarios where the author wishes the client to run the generator, but in such cases the author must guarantee that the client has the generator available. Regardless, `go get` will not automate the running of the processor, so further installation instructions will need to be provided by the author.

## Examples

Here are some hypothetical worked examples. There are countless more possibilities.

### String methods

We wish to generate a String method for a named constant type. We write a tool, say `strmeth`, that reads a definition for a single constant type and values and prints a complete Go source file containing a method definition for that type.

In our Go source file, `main.go`, we decorate each constant declaration like this (with some blank lines interposed so the generator directive does not appear in the doc comment):

```
//go:generate strmeth Day -o day_string.go $GOFILE

// Day represents the day of the week
type Day int
const (
	Sunday Day = iota
	Monday
	…
)
```

The `strmeth` generator parses the Go source to find the definition of the `Day` type and its constants, and writes out a `String() string` method for that type. For the user, generation of the string method is trivial: just run `go generate`.

### Yacc

As outlined above, we define a custom command

```
//go:generate -command yacc go tool yacc
```

and then anywhere in `main.go` (say) we write

```
//go:generate yacc -o foo.go foo.y
```

### Protocol buffers

The process is the same as with yacc. Inside main.go, we write, for each protocol buffer file we have, a line like

```
//go:generate protoc -go_out=. file.proto
```

Because of the way `protoc` works, we could generate multiple proto definitions into a single `.pb.go` file like this:

```
//go:generate protoc -go_out=. file1.proto file2.proto
```

Since no globbing is provided, one cannot say `*.proto`, but this is intentional, for simplicity and clarity of dependency.

Caveat: The protoc program must be run at the root of the source tree; we would need to provide a -cd option to it or wrap it somehow.

### Binary data

A tool that converts binary files into byte arrays that can be compiled into Go binaries would work similarly. Again, in the Go source we write something like

```
//go:generate bindata -o jpegs.go pic1.jpg pic2.jpg pic3.jpg
```

This is also demonstrates another reason the annotations are in Go source: there is no easy way to inject them into binary files.

### Sort

One could imagine a variant sort implementation that allows one to specify concrete types that have custom sorters, just by automatic rewriting of macro-like sort definition. To do this, we write a `sort.go` file that contains a complete implementation of sort on an explicit but undefined type spelled, say, `TYPE`. In that file we provide a build tag so it is never compiled (`TYPE` is not defined, so it won't compile) but is processed by `go generate`:

```
// +build generate
```

Then we write an generator directive for each type for which we want a custom sort:

```
//go:generate rename TYPE=int
//go:generate rename TYPE=strings
```

or perhaps

```
    //go:generate rename TYPE=int TYPE=strings
```

The rename processor would be a simple wrapping of `gofmt -r`, perhaps written as a shell script.

There are many more possibilities, and it is a goal of this proposal to encourage experimentation with pre-build-time code generation.