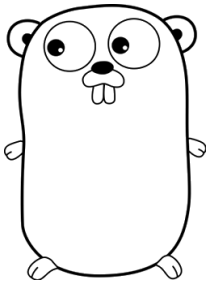

Go Developer's Reference

Typesetter:
Kyle Isom

December 16, 2012

Contents

How to Write Go Code

Introduction

This document demonstrates the development of a simple Go package and introduces the go command, the standard way to fetch, build, and install Go packages and commands.

Code organization

GOPATH and workspaces

One of Go's design goals is to make writing software easier. To that end, the go command doesn't use Makefiles or other configuration files to guide program construction. Instead, it uses the source code to find dependencies and determine build conditions. This means your source code and build scripts are always in sync; they are one and the same.

The one thing you must do is set a `GOPATH` environment variable. `GOPATH` tells the go command (and other related tools) where to find and install the Go packages on your system.

`GOPATH` is a list of paths. It shares the syntax of your system's `PATH` environment variable. A typical `GOPATH` on a Unix system might look like this:

```
GOPATH=/home/user/ext:/home/user/mygo
```

(On a Windows system use semicolons as the path separator instead of colons.)

Each path in the list (in this case `/home/user/ext` or `/home/user/mygo`) specifies the location of a *workspace*. A workspace contains Go source files and their associated package objects, and command executables. It has a prescribed structure of three subdirectories:

- `src` contains Go source files,
- `pkg` contains compiled package objects, and
- `bin` contains executable commands.

Subdirectories of the `src` directory hold independent packages, and all source files (`.go, .c, .h, and .s`) in each subdirectory are elements of that subdirectory's package.

When building a program that imports the package "`widget`" the `go` command looks for `src/pkg/widget` inside the Go root, and then—if the package source isn't found there—it searches for `src/widget` inside each workspace in order.

Multiple workspaces can offer some flexibility and convenience, but for now we'll concern ourselves with only a single workspace.

Let's work through a simple example. First, create a `$HOME/mygo` directory and its `src` subdirectory:

```
$ mkdir -p $HOME/mygo/src # create a place to put source code
```

Next, set it as the `GOPATH`. You should also add the `bin` subdirectory to your `PATH` environment variable so that you can run the commands therein without specifying their full path. To do this, add the following lines to `$HOME/.profile` (or equivalent):

```
export GOPATH=$HOME/mygo
export PATH=$PATH:$HOME/mygo/bin
```

Import paths

The standard packages are given short import paths such as "`fmt`" and "`net/http`" for convenience. For your own projects, it is important to choose a base import path that is unlikely to collide with future additions to the standard library or other external libraries.

The best way to choose an import path is to use the location of your version control repository. For instance, if your source repository is at `example.com` or `code.google.com/p/example`, you should begin your package paths with that URL, as in "`example.com/foo/bar`" or "`code.google.com/p/example/foo/bar`". Using this convention, the `go` command can automatically check out and build the source code by its import path alone.

If you don't intend to install your code in this way, you should at least use a unique prefix like "`widgets/`", as in "`widgets/foo/bar`". A good rule is to use a prefix such as your company or project name, since it is unlikely to be used by another group.

We'll use example/ as our base import path:

```
$ mkdir -p $GOPATH/src/example
```

Package names

The first statement in a Go source file should be

```
package name
```

where *name* is the package's default name for imports. (All files in a package must use the same *name*.)

Go's convention is that the package name is the last element of the import path: the package imported as "`crypto/rot13`" should be named `rot13`. There is no requirement that package names be unique across all packages linked into a single binary, only that the import paths (their full file names) be unique.

Create a new package under `example` called `newmath`:

```
$ cd $GOPATH/src/example
$ mkdir newmath
```

Then create a file named `$GOPATH/src/example/newmath/sqrt.go` containing the following Go code:

```
// Package newmath is a trivial example package.
package newmath

// Sqrt returns an approximation to the square root of x.
func Sqrt(x float64) float64 {
        // This is a terrible implementation.
        // Real code should import "math" and use math.Sqrt.
        z := 0.0
        for i := 0; i < 1000; i++ {
                z -= (z*z - x) / (2 * x)
        }
        return z
}
```

This package is imported by the path name of the directory it's in, starting after the `src` component:

```
import "example/newmath"
```

See Effective Go to learn more about Go's naming conventions.

Building and Installing

The `go` command comprises several subcommands, the most central being `install`. Running `go install` *importpath* builds and installs a package and its dependencies.

To "install a package" means to write the package object or executable command to the `pkg` or `bin` subdirectory of the workspace in which the source resides.

Building a package

To build and install the newmath package, type

```
$ go install example/newmath
```

This command will produce no output if the package and its dependencies are built and installed correctly.

As a convenience, the `go` command will assume the current directory if no import path is specified on the command line. This sequence of commands has the same effect as the one above:

```
$ cd $GOPATH/src/example/newmath
$ go install
```

The resulting workspace directory tree (assuming we're running Linux on a 64-bit system) looks like this:

```
pkg/
    linux_amd64/
        example/
            newmath.a  # package object
src/
    example/
        newmath/
            sqrt.go    # package source
```

Building a command

The `go` command treats code belonging to package main as an executable command and installs the package binary to the `GOPATH`'s `bin` subdirectory.

Add a command named `hello` to the source tree. First create the `example/hello` directory:

```
$ cd $GOPATH/src/example
$ mkdir hello
```

Then create the file `$GOPATH/src/example/hello/hello.go` containing the following Go code.

```
// Hello is a trivial example of a main package.
package main

import (
        "example/newmath"
        "fmt"
)

func main() {
        fmt.Printf("Hello, world.  Sqrt(2) = %v\n", newmath.Sqrt(2))
}
```

Next, run `go install`, which builds and installs the binary to `$GOPATH/bin` (or `$GOBIN`, if set; to simplify presentation, this document assumes `GOBIN` is unset):

```
$ go install example/hello
```

To run the program, invoke it by name as you would any other command:

```
$ $GOPATH/bin/hello
Hello, world.  Sqrt(2) = 1.414213562373095
```

If you added `$HOME/mygo/bin` to your `PATH`, you may omit the path to the executable:

```
$ hello
Hello, world.  Sqrt(2) = 1.414213562373095
```

The workspace directory tree now looks like this:

```
bin/
    hello              # command executable
pkg/
    linux_amd64/
        example/
            newmath.a  # package object
src/
    example/
        hello/
            hello.go   # command source
        newmath/
            sqrt.go    # package source
```

The `go` command also provides a `build` command, which is like `install` except it builds all objects in a temporary directory and does not install them under `pkg` or `bin`. When building a command an executable named after the last element of the import path is written to the current directory. When building a package, `go build` serves merely to test that the package and its dependencies can be built. (The resulting package object is thrown away.)

Testing

Go has a lightweight test framework composed of the `go test` command and the `testing` package.

You write a test by creating a file with a name ending in `_test.go` that contains functions named `TestXXX` with signature `func (t *testing.T)`. The test framework runs each such function; if the function calls a failure function such as `t.Error` or `t.Fail`, the test is considered to have failed.

Add a test to the `newmath` package by creating the file `$GOPATH/src/example/newmath/sqrt_test.go` containing the following Go code.

```
package newmath

import "testing"

func TestSqrt(t *testing.T) {
        const in, out = 4, 2
        if x := Sqrt(in); x != out {
                t.Errorf("Sqrt(%v) = %v, want %v", in, x, out)
        }
}
```

Now run the test with `go test`:

```
$ go test example/newmath
ok      example/newmath 0.165s
```

Run go help test and see the testing package documentation for more detail.

Remote packages

An import path can describe how to obtain the package source code using a revision control system such as Git or Mercurial. The go command uses this property to automatically fetch packages from remote repositories. For instance, the examples described in this document are also kept in a Mercurial repository hosted at Google Code, code.google.com/p/go.example. If you include the repository URL in the package's import path, `go get` will fetch, build, and install it automatically:

```
$ go get code.google.com/p/go.example/hello
$ $GOPATH/bin/hello
Hello, world.  Sqrt(2) = 1.414213562373095
```

If the specified package is not present in a workspace, go get will place it inside the first workspace specified by `GOPATH`. (If the package does already exist, `go get` skips the remote fetch and behaves the same as `go install`.)

After issuing the above `go geti` command, the workspace directory tree should now now look like this:

```
bin/
    hello                 # command executable
pkg/
    linux_amd64/
        code.google.com/p/go.example/
            newmath.a     # package object
        example/
            newmath.a     # package object
src/
     code.google.com/p/go.example/
         hello/
             hello.go      # command source
         newmath/
             sqrt.go       # package source
             sqrt_test.go  # test source
     example/
         hello/
             hello.go      # command source
         newmath/
             sqrt.go       # package source
             sqrt_test.go  # test source
```

The `hello` command hosted at Google Code depends on the `newmath` package within the same repository. The imports in `hello.go` file use the same import path convention, so the `go get` command is able to locate and install the dependent package, too.

```
import "code.google.com/p/go.example/newmath"
```

This convention is the easiest way to make your Go packages available for others to use. The Go Project Dashboard is a list of external Go projects including programs and libraries.

For more information on using remote repositories with the `go` command, see `go help remote`.

Further reading

See Effective Go for tips on writing clear, idiomatic Go code.
Take A Tour of Go to learn the language proper.
Visit the documentation page for a set of in-depth articles about the Go language and its libraries and tools.

Effective Go

Introduction

Go is a new language. Although it borrows ideas from existing languages, it has unusual properties that make effective Go programs different in character from programs written in its relatives. A straightforward translation of a C++ or Java program into Go is unlikely to produce a satisfactory result—Java programs are written in Java, not Go. On the other hand, thinking about the problem from a Go perspective could produce a successful but quite different program. In other words, to write Go well, it's important to understand its properties and idioms. It's also important to know the established conventions for programming in Go, such as naming, formatting, program construction, and so on, so that programs you write will be easy for other Go programmers to understand.

This document gives tips for writing clear, idiomatic Go code. It augments the language specification, the Tour of Go, and How to Write Go Code, all of which you should read first.

Examples

The Go package sources are intended to serve not only as the core library but also as examples of how to use the language. If you have a question about how to approach a problem or how something might be implemented, they can provide answers, ideas and background.

Formatting

Formatting issues are the most contentious but the least consequential. People can adapt to different formatting styles but it's better if they don't have to, and less time is devoted to the topic if everyone adheres to the same style. The problem is how to approach this Utopia without a long prescriptive style guide.

With Go we take an unusual approach and let the machine take care of most formatting issues. The `gofmt` program (also available as `go fmt`, which operates at the package level rather than source file level) reads a Go program and emits the source in a standard style of indentation and vertical alignment, retaining and if necessary reformatting comments. If you want to know how to handle some new layout situation, run `gofmt`; if the answer doesn't seem right, rearrange your program (or file a bug about `gofmt`), don't work around it.

As an example, there's no need to spend time lining up the comments on the fields of a structure. `Gofmt` will do that for you. Given the declaration

```
type T struct {
    name string // name of the object
    value int // its value
}
```

`gofmt` will line up the columns:

```
type T struct {
    name    string // name of the object
    value   int    // its value
}
```

All Go code in the standard packages has been formatted with `gofmt`.

Some formatting details remain. Very briefly,

Indentation We use tabs for indentation and `gofmt` emits them by default. Use spaces only if you must.

Line length Go has no line length limit. Don't worry about overflowing a punched card. If a line feels too long, wrap it and indent with an extra tab.

Parentheses Go needs fewer parentheses: control structures (`if`, `for`, `switch`) do not have parentheses in their syntax. Also, the operator precedence hierarchy is shorter and clearer, so

```
x<<8 + y<<16
```

means what the spacing implies.

Commentary

Go provides C-style `/* */` block comments and C++-style `//` line comments. Line comments are the norm; block comments appear mostly as package comments and are also useful to disable large swaths of code.

The program—and web server—`godoc` processes Go source files to extract documentation about the contents of the package. Comments that appear before top-level declarations, with no intervening newlines, are extracted along with the declaration to serve as explanatory text for the item. The nature and style of these comments determines the quality of the documentation `godoc` produces.

Every package should have a *package comment*, a block comment preceding the package clause. For multi-file packages, the package comment only needs to be present in one file, and any one will do. The package comment should introduce the package and provide information relevant to the package as a whole. It will appear first on the `godoc` page and should set up the detailed documentation that follows.

```
/*
    Package regexp implements a simple library for
    regular expressions.

    The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
        '(' regexp ')'
*/
package regexp
```

If the package is simple, the package comment can be brief.

```
// Package path implements utility routines for
// manipulating slash-separated filename paths.
```

Comments do not need extra formatting such as banners of stars. The generated output may not even be presented in a fixed-width font, so don't depend on spacing for alignment—`godoc`, like `gofmt`, takes care of that. The comments are uninterpreted plain text, so HTML and other annotations such as `_this_` will reproduce *verbatim* and should not be used. Depending on the context, `godoc` might not even reformat comments, so make sure they look good straight up: use correct spelling, punctuation, and sentence structure, fold long lines, and so on.

Inside a package, any comment immediately preceding a top-level declaration serves as a *doc comment* for that declaration. Every exported (capitalized) name in a program should have a doc comment.

Doc comments work best as complete sentences, which allow a wide variety of automated presentations. The first sentence should be a one-sentence summary that starts with the name being declared.

```
// Compile parses a regular expression and returns, if successful, a Regexp
// object that can be used to match against text.
func Compile(str string) (regexp *Regexp, err error) {
```

Go's declaration syntax allows grouping of declarations. A single doc comment can introduce a group of related constants or variables. Since the whole declaration is presented, such a comment can often be perfunctory.

```
// Error codes returned by failures to parse an expression.
var (
    ErrInternal      = errors.New("regexp: internal error")
    ErrUnmatchedLpar = errors.New("regexp: unmatched '('")
    ErrUnmatchedRpar = errors.New("regexp: unmatched ')'")
    ...
)
```

Even for private names, grouping can also indicate relationships between items, such as the fact that a set of variables is protected by a mutex.

```
var (
    countLock   sync.Mutex
    inputCount  uint32
    outputCount uint32
    errorCount  uint32
)
```

Names

Names are as important in Go as in any other language. In some cases they even have semantic effect: for instance, the visibility of a name outside a package is determined by whether its first character is upper case. It's therefore worth spending a little time talking about naming conventions in Go programs.

Package names

When a package is imported, the package name becomes an accessor for the contents. After

```
import "bytes"
```

the importing package can talk about `bytes.Buffer`. It's helpful if everyone using the package can use the same name to refer to its contents, which implies that the package name should be good: short, concise, evocative. By convention, packages are given lower case, single-word names; there should be no need for underscores or mixedCaps. Err on the side of brevity, since everyone using your package will be typing that name. And don't worry about collisions *a priori*. The package name is only the default name for imports; it need not be unique across all source code, and in the rare case of a collision the importing package can choose a different name to use locally. In any case, confusion is rare because the file name in the import determines just which package is being used.

Another convention is that the package name is the base name of its source directory; the package in `src/pkg/encoding/base64` is imported as `"encoding/base64"` but has name `base64`, not `encoding_base64` and not `encodingBase64`.

The importer of a package will use the name to refer to its contents (the `import .` notation is intended mostly for tests and other unusual situations and should be avoided unless necessary), so exported names in the package can use that fact to avoid stutter. For instance, the buffered reader type in the `bufio` package is called `Reader`, not `BufReader`, because users see it as `bufio.Reader`, which is a clear, concise name. Moreover, because imported

entities are always addressed with their package name, `bufio.Reader` does not conflict with `io.Reader`. Similarly, the function to make new instances of `ring.Ring`—which is the definition of a *constructor* in Go—would normally be called `NewRing`, but since `Ring` is the only type exported by the package, and since the package is called `ring`, it's called just `New`, which clients of the package see as `ring.New`. Use the package structure to help you choose good names.

Another short example is `once.Do`; `once.Do(setup)` reads well and would not be improved by writing `once.DoOrWaitUntilDon` Long names don't automatically make things more readable. If the name represents something intricate or subtle, it's usually better to write a helpful doc comment than to attempt to put all the information into the name.

Getters

Go doesn't provide automatic support for getters and setters. There's nothing wrong with providing getters and setters yourself, and it's often appropriate to do so, but it's neither idiomatic nor necessary to put `Get` into the getter's name. If you have a field called `owner` (lower case, unexported), the getter method should be called `Owner` (upper case, exported), not `GetOwner`. The use of upper-case names for export provides the hook to discriminate the field from the method. A setter function, if needed, will likely be called `SetOwner`. Both names read well in practice:

```
owner := obj.Owner()
if owner != user {
    obj.SetOwner(user)
}
```

Interface names

By convention, one-method interfaces are named by the method name plus the -er suffix: `Reader`, `Writer`, `Formatter` etc.

There are a number of such names and it's productive to honor them and the function names they capture. `Read`, `Write`, `Close`, `Flush`, `String` and so on have canonical signatures and meanings. To avoid confusion, don't give your method one of those names unless it has the same signature and meaning. Conversely, if your type implements a method with the same meaning as a method on a well-known type, give it the same name and signature; call your string-converter method `String` not `ToString`.

MixedCaps

Finally, the convention in Go is to use `MixedCaps` or `mixedCaps` rather than underscores to write multiword names.

Semicolons

Like C, Go's formal grammar uses semicolons to terminate statements; unlike C, those semicolons do not appear in the source. Instead the lexer uses a simple rule to insert semicolons automatically as it scans, so the input text is mostly free of them.

The rule is this. If the last token before a newline is an identifier (which includes words like `int` and `float64`), a basic literal such as a number or string constant, or one of the tokens

```
break continue fallthrough return ++ -- ) }
```

the lexer always inserts a semicolon after the token. This could be summarized as, "if the newline comes after a token that could end a statement, insert a semicolon".

A semicolon can also be omitted immediately before a closing brace, so a statement such as

```
    go func() { for { dst <- <-src } }()
```

needs no semicolons. Idiomatic Go programs have semicolons only in places such as `for` loop clauses, to separate the initializer, condition, and continuation elements. They are also necessary to separate multiple statements on a line, should you write code that way.

One caveat. You should never put the opening brace of a control structure (`if`, `for`, `switch`, or `select`) on the next line. If you do, a semicolon will be inserted before the brace, which could cause unwanted effects. Write them like this

```
if i < f() {
    g()
}
```

not like this

```
if i < f()  // wrong!
{           // wrong!
    g()
}
```

Control structures

The control structures of Go are related to those of C but differ in important ways. There is no `do` or `while` loop, only a slightly generalized `for`; `switch` is more flexible; `if` and `switch` accept an optional initialization statement like that of `for`; and there are new control structures including a type switch and a multiway communications multiplexer, `select`. The syntax is also slightly different: there are no parentheses and the bodies must always be brace-delimited.

If

In Go a simple `if` looks like this:

```
if x > 0 {
    return y
}
```

Mandatory braces encourage writing simple `if` statements on multiple lines. It's good style to do so anyway, especially when the body contains a control statement such as a `return` or `break`.

Since `if` and `switch` accept an initialization statement, it's common to see one used to set up a local variable.

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

In the Go libraries, you'll find that when an `if` statement doesn't flow into the next statement—that is, the body ends in `break`, `continue`, `goto`, or `return`—the unnecessary `else` is omitted.

```
f, err := os.Open(name)
if err != nil {
    return err
}
codeUsing(f)
```

This is an example of a common situation where code must guard against a sequence of error conditions. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in `return` statements, the resulting code needs no `else` statements.

```
f, err := os.Open(name)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
codeUsing(f, d)
```

Redeclaration

An aside: The last example in the previous section demonstrates a detail of how the `:=` short declaration form works. The declaration that calls `os.Open` reads,

```
f, err := os.Open(name)
```

This statement declares two variables, `f` and `err`. A few lines later, the call to `f.Stat` reads,

```
d, err := f.Stat()
```

which looks as if it declares `d` and `err`. Notice, though, that `err` appears in both statements. This duplication is legal: `err` is declared by the first statement, but only *re-assigned* in the second. This means that the call to `f.Stat` uses the existing `err` variable declared above, and just gives it a new value.

In a `:=` declaration a variable `v` may appear even if it has already been declared, provided:

- this declaration is in the same scope as the existing declaration of `v` (if `v` is already declared in an outer scope, the declaration will create a new variable),
- the corresponding value in the initialization is assignable to `v`, and
- there is at least one other variable in the declaration that is being declared anew.

This unusual property is pure pragmatism, making it easy to use a single `err` value, for example, in a long `if-else` chain. You'll see it used often.

For

The Go `for` loop is similar to—but not the same as—C's. It unifies `for` and `while` and there is no `do-while`. There are three forms, only one of which has semicolons.

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

Short declarations make it easy to declare the index variable right in the loop.

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

If you're looping over an array, slice, string, or map, or reading from a channel, a `range` clause can manage the loop.

```
for key, value := range oldMap {
    newMap[key] = value
}
```

If you only need the first item in the range (the key or index), drop the second:

```
for key := range m {
    if expired(key) {
        delete(m, key)
    }
}
```

If you only need the second item in the range (the value), use the *blank identifier*, an underscore, to discard the first:

```
sum := 0
for _, value := range array {
    sum += value
}
```

For strings, the `range` does more work for you, breaking out individual Unicode characters by parsing the UTF-8. Erroneous encodings consume one byte and produce the replacement rune U+FFFD. The loop

```
for pos, char := range "  " {
    fmt.Printf("character %c starts at byte position %d\n", char, pos)
}
```

prints

```
character   starts at byte position 0
character   starts at byte position 3
character   starts at byte position 6
```

Finally, Go has no comma operator and `++` and `--` are statements not expressions. Thus if you want to run multiple variables in a `for` you should use parallel assignment.

```
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

Switch

Go's `switch` is more general than C's. The expressions need not be constants or even integers, the cases are evaluated top to bottom until a match is found, and if the `switch` has no expression it switches on `true`. It's therefore possible—and idiomatic—to write an `if-else-if-else` chain as a `switch`.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
```

```
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

There is no automatic fall through, but cases can be presented in comma-separated lists.

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+', '%':
        return true
    }
    return false
}
```

Here's a comparison routine for byte arrays that uses two `switch` statements:

```
// Compare returns an integer comparing the two byte arrays,
// lexicographically.
// The result will be 0 if a == b, -1 if a < b, and +1 if a > b
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    switch {
    case len(a) < len(b):
        return -1
    case len(a) > len(b):
        return 1
    }
    return 0
}
```

A switch can also be used to discover the dynamic type of an interface variable. Such a *type switch* uses the syntax of a type assertion with the keyword `type` inside the parentheses. If the switch declares a variable in the expression, the variable will have the corresponding type in each clause.

```
switch t := interfaceValue.(type) {
default:
    fmt.Printf("unexpected type %T", t)  // %T prints type
case bool:
    fmt.Printf("boolean %t\n", t)
case int:
    fmt.Printf("integer %d\n", t)
case *bool:
    fmt.Printf("pointer to boolean %t\n", *t)
case *int:
    fmt.Printf("pointer to integer %d\n", *t)
}
```

Functions

Multiple return values

One of Go's unusual features is that functions and methods can return multiple values. This form can be used to improve on a couple of clumsy idioms in C programs: in-band error returns (such as `-1` for `EOF`) and modifying an argument.

In C, a write error is signaled by a negative count with the error code secreted away in a volatile location. In Go, `Write` can return a count *and* an error: "Yes, you wrote some bytes but not all of them because you filled the device". The signature of `File.Write` in package `os` is:

```
func (file *File) Write(b []byte) (n int, err error)
```

and as the documentation says, it returns the number of bytes written and a non-nil `error` when `n != len(b)`. This is a common style; see the section on error handling for more examples.

A similar approach obviates the need to pass a pointer to a return value to simulate a reference parameter. Here's a simple-minded function to grab a number from a position in a byte array, returning the number and the next position.

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i])-'0'
    }
    return x, i
}
```

You could use it to scan the numbers in an input array `a` like this:

```
    for i := 0; i < len(a); {
        x, i = nextInt(a, i)
        fmt.Println(x)
    }
```

Named result parameters

The return or result "parameters" of a Go function can be given names and used as regular variables, just like the incoming parameters. When named, they are initialized to the zero values for their types when the function begins; if the function executes a `return` statement with no arguments, the current values of the result parameters are used as the returned values.

The names are not mandatory but they can make code shorter and clearer: they're documentation. If we name the results of `nextInt` it becomes obvious which returned `int` is which.

```
func nextInt(b []byte, pos int) (value, nextPos int) {
```

Because named results are initialized and tied to an unadorned return, they can simplify as well as clarify. Here's a version of `io.ReadFull` that uses them well:

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:]
    }
    return
}
```

Defer

Go's `defer` statement schedules a function call (the *deferred* function) to be run immediately before the function executing the `defer` returns. It's an unusual but effective way to deal with situations such as resources that must be released regardless of which path a function takes to return. The canonical examples are unlocking a mutex or closing a file.

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close will run when we're finished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discussed later.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here.
}
```

Deferring a call to a function such as `Close` has two advantages. First, it guarantees that you will never forget to close the file, a mistake that's easy to make if you later edit the function to add a new return path. Second, it means that the close sits near the open, which is much clearer than placing it at the end of the function.

The arguments to the deferred function (which include the receiver if the function is a method) are evaluated when the *defer* executes, not when the *call* executes. Besides avoiding worries about variables changing values as the function executes, this means that a single deferred call site can defer multiple function executions. Here's a silly example.

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

Deferred functions are executed in LIFO order, so this code will cause `4 3 2 1 0` to be printed when the function returns. A more plausible example is a simple way to trace function execution through the program. We could write a couple of simple tracing routines like this:

```
func trace(s string)   { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

// Use them like this:
func a() {
    trace("a")
    defer untrace("a")
    // do something....
}
```

We can do better by exploiting the fact that arguments to deferred functions are evaluated when the `defer` executes. The tracing routine can set up the argument to the untracing routine. This example:

```
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}

func un(s string) {
    fmt.Println("leaving:", s)
}

func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}

func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

prints

```
entering: b
in b
entering: a
in a
leaving: a
leaving: b
```

For programmers accustomed to block-level resource management from other languages, `defer` may seem peculiar, but its most interesting and powerful applications come precisely from the fact that it's not block-based but function-based. In the section on `panic` and `recover` we'll see another example of its possibilities.

Data

Allocation with `new`

Go has two allocation primitives, the built-in functions `new` and `make`. They do different things and apply to different types, which can be confusing, but the rules are simple. Let's talk about `new` first. It's a built-in function that allocates memory, but unlike its namesakes in some other languages it does not *initialize* the memory, it only *zeros* it. That is, `new(T)` allocates zeroed storage for a new item of type `T` and returns its address, a value of type `*T`. In Go terminology, it returns a pointer to a newly allocated zero value of type `T`.

Since the memory returned by `new` is zeroed, it's helpful to arrange when designing your data structures that the zero value of each type can be used without further initialization. This means a user of the data structure can create one with `new` and get right to work. For example, the documentation for `bytes.Buffer` states that "the zero value for `Buffer` is an empty buffer ready to use." Similarly, `sync.Mutex` does not have an explicit constructor or `Init` method. Instead, the zero value for a `sync.Mutex` is defined to be an unlocked mutex.

The zero-value-is-useful property works transitively. Consider this type declaration.

```
type SyncedBuffer struct {
    lock    sync.Mutex
```

```
    buffer  bytes.Buffer
}
```

Values of type `SyncedBuffer` are also ready to use immediately upon allocation or just declaration. In the next snippet, both `p` and `v` will work correctly without further arrangement.

```
p := new(SyncedBuffer)  // type *SyncedBuffer
var v SyncedBuffer      // type  SyncedBuffer
```

Constructors and composite literals

Sometimes the zero value isn't good enough and an initializing constructor is necessary, as in this example derived from package `os`.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

There's a lot of boiler plate in there. We can simplify it using a *composite literal*, which is an expression that creates a new instance each time it is evaluated.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}
    return &f
}
```

Note that, unlike in C, it's perfectly OK to return the address of a local variable; the storage associated with the variable survives after the function returns. In fact, taking the address of a composite literal allocates a fresh instance each time it is evaluated, so we can combine these last two lines.

```
    return &File{fd, name, nil, 0}
```

The fields of a composite literal are laid out in order and must all be present. However, by labeling the elements explicitly as *field*`:`*value* pairs, the initializers can appear in any order, with the missing ones left as their respective zero values. Thus we could say

```
    return &File{fd: fd, name: name}
```

As a limiting case, if a composite literal contains no fields at all, it creates a zero value for the type. The expressions `new(File)` and `&File{}` are equivalent.

Composite literals can also be created for arrays, slices, and maps, with the field labels being indices or map keys as appropriate. In these examples, the initializations work regardless of the values of `Enone`, `Eio`, and `Einval`, as long as they are distinct.

```
a := [...]string   {Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
s := []string      {Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
m := map[int]string{Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
```

Allocation with make

Back to allocation. The built-in function `make(T, args)` serves a purpose different from `new(T)`. It creates slices, maps, and channels only, and it returns an *initialized* (not *zeroed*) value of type `T` (not `*T`). The reason for the distinction is that these three types are, under the covers, references to data structures that must be initialized before use. A slice, for example, is a three-item descriptor containing a pointer to the data (inside an array), the length, and the capacity, and until those items are initialized, the slice is `nil`. For slices, maps, and channels, `make` initializes the internal data structure and prepares the value for use. For instance,

```
make([]int, 10, 100)
```

allocates an array of 100 ints and then creates a slice structure with length 10 and a capacity of 100 pointing at the first 10 elements of the array. (When making a slice, the capacity can be omitted; see the section on slices for more information.) In contrast, `new([]int)` returns a pointer to a newly allocated, zeroed slice structure, that is, a pointer to a `nil` slice value.

These examples illustrate the difference between `new` and `make`.

```
var p *[]int = new([]int)       // allocates slice structure; *p == nil; rarely useful
var v  []int = make([]int, 100) // the slice v now refers to a new array of 100 ints

// Unnecessarily complex:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Idiomatic:
v := make([]int, 100)
```

Remember that `make` applies only to maps, slices and channels and does not return a pointer. To obtain an explicit pointer allocate with `new`.

Arrays

Arrays are useful when planning the detailed layout of memory and sometimes can help avoid allocation, but primarily they are a building block for slices, the subject of the next section. To lay the foundation for that topic, here are a few words about arrays.

There are major differences between the ways arrays work in Go and C. In Go,

- Arrays are values. Assigning one array to another copies all the elements.
- In particular, if you pass an array to a function, it will receive a *copy* of the array, not a pointer to it.
- The size of an array is part of its type. The types `[10]int` and `[20]int` are distinct.

The value property can be useful but also expensive; if you want C-like behavior and efficiency, you can pass a pointer to the array.

```
func Sum(a *[3]float64) (sum float64) {
    for _, v := range *a {
        sum += v
    }
    return
}

array := [...]float64{7.0, 8.5, 9.1}
x := Sum(&array)  // Note the explicit address-of operator
```

But even this style isn't idiomatic Go. Slices are.

Slices

Slices wrap arrays to give a more general, powerful, and convenient interface to sequences of data. Except for items with explicit dimension such as transformation matrices, most array programming in Go is done with slices rather than simple arrays.

Slices are *reference types*, which means that if you assign one slice to another, both refer to the same underlying array. For instance, if a function takes a slice argument, changes it makes to the elements of the slice will be visible to the caller, analogous to passing a pointer to the underlying array. A `Read` function can therefore accept a slice argument rather than a pointer and a count; the length within the slice sets an upper limit of how much data to read. Here is the signature of the `Read` method of the `File` type in package `os`:

```
func (file *File) Read(buf []byte) (n int, err error)
```

The method returns the number of bytes read and an error value, if any. To read into the first 32 bytes of a larger buffer `b`, *slice* (here used as a verb) the buffer.

```
    n, err := f.Read(buf[0:32])
```

Such slicing is common and efficient. In fact, leaving efficiency aside for the moment, the following snippet would also read the first 32 bytes of the buffer.

```
    var n int
    var err error
    for i := 0; i < 32; i++ {
        nbytes, e := f.Read(buf[i:i+1])  // Read one byte.
        if nbytes == 0 || e != nil {
            err = e
            break
        }
        n += nbytes
    }
```

The length of a slice may be changed as long as it still fits within the limits of the underlying array; just assign it to a slice of itself. The *capacity* of a slice, accessible by the built-in function `cap`, reports the maximum length the slice may assume. Here is a function to append data to a slice. If the data exceeds the capacity, the slice is reallocated. The resulting slice is returned. The function uses the fact that `len` and `cap` are legal when applied to the `nil` slice, and return 0.

```
func Append(slice, data[]byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // reallocate
        // Allocate double what's needed, for future growth.
        newSlice := make([]byte, (l+len(data))*2)
        // The copy function is predeclared and works for any slice type.
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    for i, c := range data {
        slice[l+i] = c
    }
    return slice
}
```

We must return the slice afterwards because, although `Append` can modify the elements of `slice`, the slice itself (the run-time data structure holding the pointer, length, and capacity) is passed by value.

The idea of appending to a slice is so useful it's captured by the `append` built-in function. To understand that function's design, though, we need a little more information, so we'll return to it later.

Maps

Maps are a convenient and powerful built-in data structure to associate values of different types. The key can be of any type for which the equality operator is defined, such as integers, floating point and complex numbers, strings, pointers, interfaces (as long as the dynamic type supports equality), structs and arrays. Slices cannot be used as map keys, because equality is not defined on them. Like slices, maps are a reference type. If you pass a map to a function that changes the contents of the map, the changes will be visible in the caller.

Maps can be constructed using the usual composite literal syntax with colon-separated key-value pairs, so it's easy to build them during initialization.

```
var timeZone = map[string] int {
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
}
```

Assigning and fetching map values looks syntactically just like doing the same for arrays except that the index doesn't need to be an integer.

```
offset := timeZone["EST"]
```

An attempt to fetch a map value with a key that is not present in the map will return the zero value for the type of the entries in the map. For instance, if the map contains integers, looking up a non-existent key will return `0`. A set can be implemented as a map with value type `bool`. Set the map entry to `true` to put the value in the set, and then test it by simple indexing.

```
attended := map[string] bool {
    "Ann": true,
    "Joe": true,
    ...
}

if attended[person] { // will be false if person is not in the map
    fmt.Println(person, "was at the meeting")
}
```

Sometimes you need to distinguish a missing entry from a zero value. Is there an entry for `"UTC"` or is that zero value because it's not in the map at all? You can discriminate with a form of multiple assignment.

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

For obvious reasons this is called the "comma ok" idiom. In this example, if `tz` is present, `seconds` will be set appropriately and `ok` will be true; if not, `seconds` will be set to zero and `ok` will be false. Here's a function that puts it together with a nice error report:

```
func offset(tz string) int {
    if seconds, ok := timeZone[tz]; ok {
        return seconds
    }
    log.Println("unknown time zone:", tz)
    return 0
}
```

To test for presence in the map without worrying about the actual value, you can use the blank identifier (_). The blank identifier can be assigned or declared with any value of any type, with the value discarded harmlessly. For testing just presence in a map, use the blank identifier in place of the usual variable for the value.

```
_, present := timeZone[tz]
```

To delete a map entry, use the `delete` built-in function, whose arguments are the map and the key to be deleted. It's safe to do this this even if the key is already absent from the map.

```
delete(timeZone, "PDT")  // Now on Standard Time
```

Printing

Formatted printing in Go uses a style similar to C's `printf` family but is richer and more general. The functions live in the `fmt` package and have capitalized names: `fmt.Printf`, `fmt.Fprintf`, `fmt.Sprintf` and so on. The string functions (`Sprintf` etc.) return a string rather than filling in a provided buffer.

You don't need to provide a format string. For each of `Printf`, `Fprintf` and `Sprintf` there is another pair of functions, for instance `Print` and `Println`. These functions do not take a format string but instead generate a default format for each argument. The `Println` versions also insert a blank between arguments and append a newline to the output while the `Print` versions add blanks only if the operand on neither side is a string. In this example each line produces the same output.

```
fmt.Printf("Hello %d\n", 23)
fmt.Fprint(os.Stdout, "Hello ", 23, "\n")
fmt.Println("Hello", 23)
fmt.Println(fmt.Sprint("Hello ", 23))
```

As mentioned in the Tour, `fmt.Fprint` and friends take as a first argument any object that implements the `io.Writer` interface; the variables `os.Stdout` and `os.Stderr` are familiar instances.

Here things start to diverge from C. First, the numeric formats such as `%d` do not take flags for signedness or size; instead, the printing routines use the type of the argument to decide these properties.

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

prints

```
18446744073709551615 ffffffffffffffff; -1 -1
```

If you just want the default conversion, such as decimal for integers, you can use the catchall format `%v` (for "value"); the result is exactly what `Print` and `Println` would produce. Moreover, that format can print *any* value, even arrays, structs, and maps. Here is a print statement for the time zone map defined in the previous section.

```
fmt.Printf("%v\n", timeZone)  // or just fmt.Println(timeZone)
```

which gives output

```
map[CST:-21600 PST:-28800 EST:-18000 UTC:0 MST:-25200]
```

For maps the keys may be output in any order, of course. When printing a struct, the modified format `%+v` annotates the fields of the structure with their names, and for any value the alternate format `%#v` prints the value in full Go syntax.

```
type T struct {
    a int
    b float64
    c string
}
t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", timeZone)
```

prints

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string] int{"CST":-21600, "PST":-28800, "EST":-18000, "UTC":0, "MST":-25200}
```

(Note the ampersands.) That quoted string format is also available through `%q` when applied to a value of type `string` or `[]byte`; the alternate format `%#q` will use backquotes instead if possible. Also, `%x` works on strings and arrays of bytes as well as on integers, generating a long hexadecimal string, and with a space in the format (`% x`) it puts spaces between the bytes.

Another handy format is `%T`, which prints the *type* of a value.

```
fmt.Printf("%T\n", timeZone)
```

prints

```
map[string] int
```

If you want to control the default format for a custom type, all that's required is to define a method with the signature `String() string` on the type. For our simple type `T`, that might look like this.

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

to print in the format

```
7/-2.35/"abc\tdef"
```

(If you need to print *values* of type `T` as well as pointers to `T`, the receiver for `String` must be of value type; this example used a pointer because that's more efficient and idiomatic for struct types. See the section below on pointers vs. value receivers for more information.)

Our `String` method is able to call `Sprintf` because the print routines are fully reentrant and can be used recursively. We can even go one step further and pass a print routine's arguments directly to another such routine. The signature of `Printf` uses the type `...interface{}` for its final argument to specify that an arbitrary number of parameters (of arbitrary type) can appear after the format.

```
func Printf(format string, v ...interface{}) (n int, err error) {
```

Within the function `Printf`, `v` acts like a variable of type `[]interface{}` but if it is passed to another variadic function, it acts like a regular list of arguments. Here is the implementation of the function `log.Println` we used above. It passes its arguments directly to `fmt.Sprintln` for the actual formatting.

```
// Println prints to the standard logger in the manner of fmt.Println.
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))  // Output takes parameters (int, string)
}
```

We write `...` after `v` in the nested call to `Sprintln` to tell the compiler to treat `v` as a list of arguments; otherwise it would just pass `v` as a single slice argument.

There's even more to printing than we've covered here. See the `godoc` documentation for package `fmt` for the details.

By the way, a `...` parameter can be of a specific type, for instance `...int` for a min function that chooses the least of a list of integers:

```
func Min(a ...int) int {
    min := int(^uint(0) >> 1)  // largest int
    for _, i := range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

Append

Now we have the missing piece we needed to explain the design of the `append` built-in function. The signature of `append` is different from our custom `Append` function above. Schematically, it's like this:

```
func append(slice []T, elements...T) []T
```

where T is a placeholder for any given type. You can't actually write a function in Go where the type `T` is determined by the caller. That's why `append` is built in: it needs support from the compiler.

What `append` does is append the elements to the end of the slice and return the result. The result needs to be returned because, as with our hand-written `Append`, the underlying array may change. This simple example

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

prints `[1 2 3 4 5 6]`. So `append` works a little like `Printf`, collecting an arbitrary number of arguments.

But what if we wanted to do what our `Append` does and append a slice to a slice? Easy: use `...` at the call site, just as we did in the call to `Output` above. This snippet produces identical output to the one above.

```
x := []int{1,2,3}
y := []int{4,5,6}
x = append(x, y...)
fmt.Println(x)
```

Without that `...`, it wouldn't compile because the types would be wrong; `y` is not of type `int`.

Initialization

Although it doesn't look superficially very different from initialization in C or C++, initialization in Go is more powerful. Complex structures can be built during initialization and the ordering issues between initialized objects in different packages are handled correctly.

Constants

Constants in Go are just that—constant. They are created at compile time, even when defined as locals in functions, and can only be numbers, strings or booleans. Because of the compile-time restriction, the expressions that define them must be constant expressions, evaluatable by the compiler. For instance, `1<<3` is a constant expression, while `math.Sin(math.Pi/4)` is not because the function call to `math.Sin` needs to happen at run time.

In Go, enumerated constants are created using the `iota` enumerator. Since `iota` can be part of an expression and expressions can be implicitly repeated, it is easy to build intricate sets of values.

{{code "/doc/progs/eff_bytesize.go" '/^type ByteSize/' '/^\)/'}}

The ability to attach a method such as `String` to a type makes it possible for such values to format themselves automatically for printing, even as part of a general type.

{{code "/doc/progs/eff_bytesize.go" '/^func.*ByteSize.*String/' '/^}/'}}

The expression `YB` prints as `1.00YB`, while `ByteSize(1e13)` prints as `9.09TB`.

Note that it's fine to call `Sprintf` and friends in the implementation of `String` methods, but beware of recurring into the `String` method through the nested `Sprintf` call using a string format (`%s`, `%q`, `%v`, `%x` or `%X`). The `ByteSize` implementation of `String` is safe because it calls `Sprintf` with `%f`.

Variables

Variables can be initialized just like constants but the initializer can be a general expression computed at run time.

```
var (
    HOME = os.Getenv("HOME")
    USER = os.Getenv("USER")
    GOROOT = os.Getenv("GOROOT")
)
```

The init function

Finally, each source file can define its own niladic `init` function to set up whatever state is required. (Actually each file can have multiple `init` functions.) And finally means finally: `init` is called after all the variable declarations in the package have evaluated their initializers, and those are evaluated only after all the imported packages have been initialized.

Besides initializations that cannot be expressed as declarations, a common use of `init` functions is to verify or repair correctness of the program state before real execution begins.

```
func init() {
    if USER == "" {
        log.Fatal("$USER not set")
    }
    if HOME == "" {
        HOME = "/usr/" + USER
    }
    if GOROOT == "" {
        GOROOT = HOME + "/go"
    }
    // GOROOT may be overridden by --goroot flag on command line.
    flag.StringVar(&GOROOT, "goroot", GOROOT, "Go root directory")
}
```

Methods

Pointers vs. Values

Methods can be defined for any named type that is not a pointer or an interface; the receiver does not have to be a struct.

In the discussion of slices above, we wrote an `Append` function. We can define it as a method on slices instead. To do this, we first declare a named type to which we can bind the method, and then make the receiver for the method a value of that type.

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // Body exactly the same as above
}
```

This still requires the method to return the updated slice. We can eliminate that clumsiness by redefining the method to take a *pointer* to a `ByteSlice` as its receiver, so the method can overwrite the caller's slice.

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // Body as above, without the return.
    *p = slice
}
```

In fact, we can do even better. If we modify our function so it looks like a standard `Write` method, like this,

```
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    // Again as above.
    *p = slice
    return len(data), nil
}
```

then the type `*ByteSlice` satisfies the standard interface `io.Writer`, which is handy. For instance, we can print into one.

```
    var b ByteSlice
    fmt.Fprintf(&b, "This hour has %d days\n", 7)
```

We pass the address of a `ByteSlice` because only `*ByteSlice` satisfies `io.Writer`. The rule about pointers vs. values for receivers is that value methods can be invoked on pointers and values, but pointer methods can only be invoked on pointers. This is because pointer methods can modify the receiver; invoking them on a copy of the value would cause those modifications to be discarded.

By the way, the idea of using `Write` on a slice of bytes is implemented by `bytes.Buffer`.

Interfaces and other types

Interfaces

Interfaces in Go provide a way to specify the behavior of an object: if something can do *this*, then it can be used *here*. We've seen a couple of simple examples already; custom printers can be implemented by a `String` method while `Fprintf` can generate output to anything with a `Write` method. Interfaces with only one or two methods are common in Go code, and are usually given a name derived from the method, such as `io.Writer` for something that implements `Write`.

A type can implement multiple interfaces. For instance, a collection can be sorted by the routines in package `sort` if it implements `sort.Interface`, which contains `Len()`, `Less(i, j int) bool`, and `Swap(i, j int)`, and it could also have a custom formatter. In this contrived example `Sequence` satisfies both.

{{code "/doc/progs/eff_sequence.go" `/^type/` "$"}}

Conversions

The `String` method of `Sequence` is recreating the work that `Sprint` already does for slices. We can share the effort if we convert the `Sequence` to a plain `[]int` before calling `Sprint`.

```
func (s Sequence) String() string {
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

The conversion causes `s` to be treated as an ordinary slice and therefore receive the default formatting. Without the conversion, `Sprint` would find the `String` method of `Sequence` and recur indefinitely. Because the two types (`Sequence` and `[]int`) are the same if we ignore the type name, it's legal to convert between them. The conversion doesn't create a new value, it just temporarily acts as though the existing value has a new type. (There are other legal conversions, such as from integer to floating point, that do create a new value.)

It's an idiom in Go programs to convert the type of an expression to access a different set of methods. As an example, we could use the existing type `sort.IntSlice` to reduce the entire example to this:

```
type Sequence []int

// Method for printing - sorts the elements before printing
func (s Sequence) String() string {
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

Now, instead of having `Sequence` implement multiple interfaces (sorting and printing), we're using the ability of a data item to be converted to multiple types (`Sequence`, `sort.IntSlice` and `[]int`), each of which does some part of the job. That's more unusual in practice but can be effective.

Generality

If a type exists only to implement an interface and has no exported methods beyond that interface, there is no need to export the type itself. Exporting just the interface makes it clear that it's the behavior that matters, not the implementation, and that other implementations with different properties can mirror the behavior of the original type. It also avoids the need to repeat the documentation on every instance of a common method.

In such cases, the constructor should return an interface value rather than the implementing type. As an example, in the hash libraries both `crc32.NewIEEE` and `adler32.New` return the interface type `hash.Hash32`. Substituting the CRC-32 algorithm for Adler-32 in a Go program requires only changing the constructor call; the rest of the code is unaffected by the change of algorithm.

A similar approach allows the streaming cipher algorithms in the various `crypto` packages to be separated from the block ciphers they chain together. The `Block` interface in the `crypto/cipher` package specifies the behavior of a block cipher, which provides encryption of a single block of data. Then, by analogy with the `bufio` package, cipher packages that implement this interface can be used to construct streaming ciphers, represented by the `Stream` interface, without knowing the details of the block encryption.

The `crypto/cipher` interfaces look like this:

```
type Block interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}

type Stream interface {
    XORKeyStream(dst, src []byte)
}
```

Here's the definition of the counter mode (CTR) stream, which turns a block cipher into a streaming cipher; notice that the block cipher's details are abstracted away:

```
// NewCTR returns a Stream that encrypts/decrypts using the given Block in
// counter mode. The length of iv must be the same as the Block's block size.
func NewCTR(block Block, iv []byte) Stream
```

`NewCTR` applies not just to one specific encryption algorithm and data source but to any implementation of the `Block` interface and any `Stream`. Because they return interface values, replacing CTR encryption with other encryption modes is a localized change. The constructor calls must be edited, but because the surrounding code must treat the result only as a `Stream`, it won't notice the difference.

Interfaces and methods

Since almost anything can have methods attached, almost anything can satisfy an interface. One illustrative example is in the `http` package, which defines the `Handler` interface. Any object that implements `Handler` can serve HTTP requests.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

`ResponseWriter` is itself an interface that provides access to the methods needed to return the response to the client. Those methods include the standard `Write` method, so an `http.ResponseWriter` can be used wherever an `io.Writer` can be used. `Request` is a struct containing a parsed representation of the request from the client.

For brevity, let's ignore POSTs and assume HTTP requests are always GETs; that simplification does not affect the way the handlers are set up. Here's a trivial but complete implementation of a handler to count the number of times the page is visited.

```
// Simple counter server.
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ctr.n++
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}
```

(Keeping with our theme, note how `Fprintf` can print to an `http.ResponseWriter`.) For reference, here's how to attach such a server to a node on the URL tree.

```
import "net/http"
...
ctr := new(Counter)
http.Handle("/counter", ctr)
```

But why make `Counter` a struct? An integer is all that's needed. (The receiver needs to be a pointer so the increment is visible to the caller.)

```
// Simpler counter server.
type Counter int

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}
```

What if your program has some internal state that needs to be notified that a page has been visited? Tie a channel to the web page.

```
// A channel that sends a notification on each visit.
// (Probably want the channel to be buffered.)
type Chan chan *http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ch <- req
    fmt.Fprint(w, "notification sent")
}
```

Finally, let's say we wanted to present on `/args` the arguments used when invoking the server binary. It's easy to write a function to print the arguments.

```
func ArgServer() {
    for _, s := range os.Args {
        fmt.Println(s)
    }
}
```

How do we turn that into an HTTP server? We could make `ArgServer` a method of some type whose value we ignore, but there's a cleaner way. Since we can define a method for any type except pointers and interfaces, we can write a method for a function. The `http` package contains this code:

```
// The HandlerFunc type is an adapter to allow the use of
// ordinary functions as HTTP handlers.  If f is a function
// with the appropriate signature, HandlerFunc(f) is a
// Handler object that calls f.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(c, req).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, req *Request) {
    f(w, req)
}
```

`HandlerFunc` is a type with a method, `ServeHTTP`, so values of that type can serve HTTP requests. Look at the implementation of the method: the receiver is a function, `f`, and the method calls `f`. That may seem odd but it's not that different from, say, the receiver being a channel and the method sending on the channel.

To make `ArgServer` into an HTTP server, we first modify it to have the right signature.

```
// Argument server.
func ArgServer(w http.ResponseWriter, req *http.Request) {
    for _, s := range os.Args {
        fmt.Fprintln(w, s)
    }
}
```

`ArgServer` now has same signature as `HandlerFunc`, so it can be converted to that type to access its methods, just as we converted `Sequence` to `IntSlice` to access `IntSlice.Sort`. The code to set it up is concise:

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

When someone visits the page `/args`, the handler installed at that page has value `ArgServer` and type `HandlerFunc`. The HTTP server will invoke the method `ServeHTTP` of that type, with `ArgServer` as the receiver, which will in turn call `ArgServer` (via the invocation `f(c, req)` inside `HandlerFunc.ServeHTTP`). The arguments will then be displayed.

In this section we have made an HTTP server from a struct, an integer, a channel, and a function, all because interfaces are just sets of methods, which can be defined for (almost) any type.

Embedding

Go does not provide the typical, type-driven notion of subclassing, but it does have the ability to "borrow" pieces of an implementation by *embedding* types within a struct or interface.

Interface embedding is very simple. We've mentioned the `io.Reader` and `io.Writer` interfaces before; here are their definitions.

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

type Writer interface {
    Write(p []byte) (n int, err error)
}
```

The `io` package also exports several other interfaces that specify objects that can implement several such methods. For instance, there is `io.ReadWriter`, an interface containing both `Read` and `Write`. We could specify `io.ReadWriter` by listing the two methods explicitly, but it's easier and more evocative to embed the two interfaces to form the new one, like this:

```
// ReadWriter is the interface that combines the Reader and Writer interfaces.
type ReadWriter interface {
    Reader
    Writer
}
```

This says just what it looks like: A `ReadWriter` can do what a `Reader` does *and* what a `Writer` does; it is a union of the embedded interfaces (which must be disjoint sets of methods). Only interfaces can be embedded within interfaces.

The same basic idea applies to structs, but with more far-reaching implications. The `bufio` package has two struct types, `bufio.Reader` and `bufio.Writer`, each of which of course implements the analogous interfaces from package `io`. And `bufio` also implements a buffered reader/writer, which it does by combining a reader and a writer into one struct using embedding: it lists the types within the struct but does not give them field names.

```
// ReadWriter stores pointers to a Reader and a Writer.
// It implements io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

The embedded elements are pointers to structs and of course must be initialized to point to valid structs before they can be used. The `ReadWriter` struct could be written as

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

but then to promote the methods of the fields and to satisfy the `io` interfaces, we would also need to provide forwarding methods, like this:

```
func (rw *ReadWriter) Read(p []byte) (n int, err error) {
    return rw.reader.Read(p)
}
```

By embedding the structs directly, we avoid this bookkeeping. The methods of embedded types come along for free, which means that `bufio.ReadWriter` not only has the methods of `bufio.Reader` and `bufio.Writer`, it also satisfies all three interfaces: `io.Reader`, `io.Writer`, and `io.ReadWriter`.

There's an important way in which embedding differs from subclassing. When we embed a type, the methods of that type become methods of the outer type, but when they are invoked the receiver of the method is the inner type, not the outer one. In our example, when the `Read` method of a `bufio.ReadWriter` is invoked, it has exactly the same effect as the forwarding method written out above; the receiver is the `reader` field of the `ReadWriter`, not the `ReadWriter` itself.

Embedding can also be a simple convenience. This example shows an embedded field alongside a regular, named field.

```
type Job struct {
    Command string
    *log.Logger
}
```

The `Job` type now has the `Log`, `Logf` and other methods of `*log.Logger`. We could have given the `Logger` a field name, of course, but it's not necessary to do so. And now, once initialized, we can log to the `Job`:

```
job.Log("starting now...")
```

The `Logger` is a regular field of the struct and we can initialize it in the usual way with a constructor,

```
func NewJob(command string, logger *log.Logger) *Job {
    return &Job{command, logger}
}
```

or with a composite literal,

```
job := &Job{command, log.New(os.Stderr, "Job: ", log.Ldate)}
```

If we need to refer to an embedded field directly, the type name of the field, ignoring the package qualifier, serves as a field name. If we needed to access the `*log.Logger` of a `Job` variable `job`, we would write `job.Logger`. This would be useful if we wanted to refine the methods of `Logger`.

```
func (job *Job) Logf(format string, args ...interface{}) {
    job.Logger.Logf("%q: %s", job.Command, fmt.Sprintf(format, args...))
}
```

Embedding types introduces the problem of name conflicts but the rules to resolve them are simple. First, a field or method `X` hides any other item `X` in a more deeply nested part of the type. If `log.Logger` contained a field or method called `Command`, the `Command` field of `Job` would dominate it.

Second, if the same name appears at the same nesting level, it is usually an error; it would be erroneous to embed `log.Logger` if the `Job` struct contained another field or method called `Logger`. However, if the duplicate name is never mentioned in the program outside the type definition, it is OK. This qualification provides some protection against changes made to types embedded from outside; there is no problem if a field is added that conflicts with another field in another subtype if neither field is ever used.

Concurrency

Share by communicating

Concurrent programming is a large topic and there is space only for some Go-specific highlights here.

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one goroutine has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:

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

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.

One way to think about this model is to consider a typical single-threaded program running on one CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchronization. Unix pipelines, for example, fit this model perfectly. Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be seen as a type-safe generalization of Unix pipes.

Goroutines

They're called *goroutines* because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations. A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.

Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and management.

Prefix a function or method call with the `go` keyword to run the call in a new goroutine. When the call completes, the goroutine exits, silently. (The effect is similar to the Unix shell's `&` notation for running a command in the background.)

```
go list.Sort()  // run list.Sort concurrently; don't wait for it.
```

A function literal can be handy in a goroutine invocation.

```
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()  // Note the parentheses - must call the function.
}
```

In Go, function literals are closures: the implementation makes sure the variables referred to by the function survive as long as they are active.

These examples aren't too practical because the functions have no way of signaling completion. For that, we need channels.

Channels

Like maps, channels are a reference type and are allocated with `make`. If an optional integer parameter is provided, it sets the buffer size for the channel. The default is zero, for an unbuffered or synchronous channel.

```
ci := make(chan int)            // unbuffered channel of integers
cj := make(chan int, 0)         // unbuffered channel of integers
cs := make(chan *os.File, 100)  // buffered channel of pointers to Files
```

Channels combine communication—the exchange of a value—with synchronization—guaranteeing that two calculations (goroutines) are in a known state.

There are lots of nice idioms using channels. Here's one to get us started. In the previous section we launched a sort in the background. A channel can allow the launching goroutine to wait for the sort to complete.

```
c := make(chan int)  // Allocate a channel.
// Start the sort in a goroutine; when it completes, signal on the channel.
go func() {
    list.Sort()
```

```
    c <- 1  // Send a signal; value does not matter.
}()
doSomethingForAWhile()
<-c   // Wait for sort to finish; discard sent value.
```

Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value.

A buffered channel can be used like a semaphore, for instance to limit throughput. In this example, incoming requests are passed to `handle`, which sends a value into the channel, processes the request, and then receives a value from the channel. The capacity of the channel buffer limits the number of simultaneous calls to `process`.

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1    // Wait for active queue to drain.
    process(r)  // May take a long time.
    <-sem       // Done; enable next request to run.
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // Don't wait for handle to finish.
    }
}
```

Here's the same idea implemented by starting a fixed number of `handle` goroutines all reading from the request channel. The number of goroutines limits the number of simultaneous calls to `process`. This `Serve` function also accepts a channel on which it will be told to exit; after launching the goroutines it blocks receiving from that channel.

```
func handle(queue chan *Request) {
    for r := range queue {
        process(r)
    }
}

func Serve(clientRequests chan *Request, quit chan bool) {
    // Start handlers
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // Wait to be told to exit.
}
```

Channels of channels

One of the most important properties of Go is that a channel is a first-class value that can be allocated and passed around like any other. A common use of this property is to implement safe, parallel demultiplexing.

In the example in the previous section, `handle` was an idealized handler for a request but we didn't define the type it was handling. If that type includes a channel on which to reply, each client can provide its own path for the answer. Here's a schematic definition of type `Request`.

```
type Request struct {
    args        []int
    f           func([]int) int
```

```
    resultChan  chan int
}
```

The client provides a function and its arguments, as well as a channel inside the request object on which to receive the answer.

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Send request
clientRequests <- request
// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)
```

On the server side, the handler function is the only thing that changes.

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

There's clearly a lot more to do to make it realistic, but this code is a framework for a rate-limited, parallel, non-blocking RPC system, and there's not a mutex in sight.

Parallelization

Another application of these ideas is to parallelize a calculation across multiple CPU cores. If the calculation can be broken into separate pieces that can execute independently, it can be parallelized, with a channel to signal when each piece completes.

Let's say we have an expensive operation to perform on a vector of items, and that the value of the operation on each item is independent, as in this idealized example.

```
type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to v[n-1].
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1    // signal that this piece is done
}
```

We launch the pieces independently in a loop, one per CPU. They can complete in any order but it doesn't matter; we just count the completion signals by draining the channel after launching all the goroutines.

```
const NCPU = 4  // number of CPU cores

func (v Vector) DoAll(u Vector) {
    c := make(chan int, NCPU)  // Buffering optional but sensible.
    for i := 0; i < NCPU; i++ {
```

```
        go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < NCPU; i++ {
        <-c    // wait for one task to complete
    }
    // All done.
}
```

The current implementation of the Go runtime will not parallelize this code by default. It dedicates only a single core to user-level processing. An arbitrary number of goroutines can be blocked in system calls, but by default only one can be executing user-level code at any time. It should be smarter and one day it will be smarter, but until it is if you want CPU parallelism you must tell the run-time how many goroutines you want executing code simultaneously. There are two related ways to do this. Either run your job with environment variable `GOMAXPROCS` set to the number of cores to use or import the `runtime` package and call `runtime.GOMAXPROCS(NCPU)`. A helpful value might be `runtime.NumCPU()`, which reports the number of logical CPUs on the local machine. Again, this requirement is expected to be retired as the scheduling and run-time improve.

A leaky buffer

The tools of concurrent programming can even make non-concurrent ideas easier to express. Here's an example abstracted from an RPC package. The client goroutine loops receiving data from some source, perhaps a network. To avoid allocating and freeing buffers, it keeps a free list, and uses a buffered channel to represent it. If the channel is empty, a new buffer gets allocated. Once the message buffer is ready, it's sent to the server on `serverChan`.

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        var b *Buffer
        // Grab a buffer if available; allocate if not.
        select {
        case b = <-freeList:
            // Got one; nothing more to do.
        default:
            // None free, so allocate a new one.
            b = new(Buffer)
        }
        load(b)              // Read next message from the net.
        serverChan <- b      // Send to server.
    }
}
```

The server loop receives each message from the client, processes it, and returns the buffer to the free list.

```
func server() {
    for {
        b := <-serverChan    // Wait for work.
        process(b)
        // Reuse buffer if there's room.
        select {
        case freeList <- b:
            // Buffer on free list; nothing more to do.
        default:
            // Free list full, just carry on.
        }
```

```
    }
}
```

The client attempts to retrieve a buffer from `freeList`; if none is available, it allocates a fresh one. The server's send to `freeList` puts `b` back on the free list unless the list is full, in which case the buffer is dropped on the floor to be reclaimed by the garbage collector. (The `default` clauses in the `select` statements execute when no other case is ready, meaning that the `selects` never block.) This implementation builds a leaky bucket free list in just a few lines, relying on the buffered channel and the garbage collector for bookkeeping.

Errors

Library routines must often return some sort of error indication to the caller. As mentioned earlier, Go's multivalue return makes it easy to return a detailed error description alongside the normal return value. By convention, errors have type `error`, a simple built-in interface.

```
type error interface {
    Error() string
}
```

A library writer is free to implement this interface with a richer model under the covers, making it possible not only to see the error but also to provide some context. For example, `os.Open` returns an `os.PathError`.

```
// PathError records an error and the operation and
// file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Err error    // Returned by the system call.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

`PathError`'s `Error` generates a string like this:

```
open /etc/passwx: no such file or directory
```

Such an error, which includes the problematic file name, the operation, and the operating system error it triggered, is useful even if printed far from the call that caused it; it is much more informative than the plain "no such file or directory".

When feasible, error strings should identify their origin, such as by having a prefix naming the package that generated the error. For example, in package `image`, the string representation for a decoding error due to an unknown format is "image: unknown format".

Callers that care about the precise error details can use a type switch or a type assertion to look for specific errors and extract details. For `PathErrors` this might include examining the internal `Err` field for recoverable failures.

```
for try := 0; try < 2; try++ {
    file, err = os.Create(filename)
    if err == nil {
        return
    }
    if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC {
        deleteTempFiles()  // Recover some space.
        continue
```

```
    }
    return
}
```

The second `if` statement here is idiomatic Go. The type assertion `err.(*os.PathError)` is checked with the "comma ok" idiom (mentioned earlier in the context of examining maps). If the type assertion fails, `ok` will be false, and `e` will be `nil`. If it succeeds, `ok` will be true, which means the error was of type `*os.PathError`, and then so is `e`, which we can examine for more information about the error.

Panic

The usual way to report an error to a caller is to return an `error` as an extra return value. The canonical `Read` method is a well-known instance; it returns a byte count and an `error`. But what if the error is unrecoverable? Sometimes the program simply cannot continue.

For this purpose, there is a built-in function `panic` that in effect creates a run-time error that will stop the program (but see the next section). The function takes a single argument of arbitrary type—often a string—to be printed as the program dies. It's also a way to indicate that something impossible has happened, such as exiting an infinite loop. In fact, the compiler recognizes a `panic` at the end of a function and suppresses the usual check for a `return` statement.

```
// A toy implementation of cube root using Newton's method.
func CubeRoot(x float64) float64 {
    z := x/3   // Arbitrary initial value
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // A million iterations has not converged; something is wrong.
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

This is only an example but real library functions should avoid `panic`. If the problem can be masked or worked around, it's always better to let things continue to run rather than taking down the whole program. One possible counterexample is during initialization: if the library truly cannot set itself up, it might be reasonable to panic, so to speak.

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

Recover

When `panic` is called, including implicitly for run-time errors such as indexing an array out of bounds or failing a type assertion, it immediately stops execution of the current function and begins unwinding the stack of the goroutine, running any deferred functions along the way. If that unwinding reaches the top of the goroutine's stack, the program dies. However, it is possible to use the built-in function `recover` to regain control of the goroutine and resume normal execution.

A call to `recover` stops the unwinding and returns the argument passed to `panic`. Because the only code that runs while unwinding is inside deferred functions, `recover` is only useful inside deferred functions.

One application of `recover` is to shut down a failing goroutine inside a server without killing the other executing goroutines.

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("work failed:", err)
        }
    }()
    do(work)
}
```

In this example, if `do(work)` panics, the result will be logged and the goroutine will exit cleanly without disturbing the others. There's no need to do anything else in the deferred closure; calling `recover` handles the condition completely.

Because `recover` always returns `nil` unless called directly from a deferred function, deferred code can call library routines that themselves use `panic` and `recover` without failing. As an example, the deferred function in `safelyDo` might call a logging function before calling `recover`, and that logging code would run unaffected by the panicking state.

With our recovery pattern in place, the `do` function (and anything it calls) can get out of any bad situation cleanly by calling `panic`. We can use that idea to simplify error handling in complex software. Let's look at an idealized excerpt from the `regexp` package, which reports parsing errors by calling `panic` with a local error type. Here's the definition of `Error`, an `error` method, and the `Compile` function.

```
// Error is the type of a parse error; it satisfies the error interface.
type Error string
func (e Error) Error() string {
    return string(e)
}

// error is a method of *Regexp that reports parsing errors by
// panicking with an Error.
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compile returns a parsed representation of the regular expression.
func Compile(str string) (regexp *Regexp, err error) {
    regexp = new(Regexp)
    // doParse will panic if there is a parse error.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil    // Clear return value.
            err = e.(Error) // Will re-panic if not a parse error.
        }
    }()
    return regexp.doParse(str), nil
}
```

If `doParse` panics, the recovery block will set the return value to `nil`—deferred functions can modify named return values. It then will then check, in the assignment to `err`, that the problem was a parse error by asserting

that it has the local type `Error`. If it does not, the type assertion will fail, causing a run-time error that continues the stack unwinding as though nothing had interrupted it. This check means that if something unexpected happens, such as an array index out of bounds, the code will fail even though we are using `panic` and `recover` to handle user-triggered errors.

With error handling in place, the `error` method makes it easy to report parse errors without worrying about unwinding the parse stack by hand.

Useful though this pattern is, it should be used only within a package. `Parse` turns its internal `panic` calls into `error` values; it does not expose `panics` to its client. That is a good rule to follow.

By the way, this re-panic idiom changes the panic value if an actual error occurs. However, both the original and new failures will be presented in the crash report, so the root cause of the problem will still be visible. Thus this simple re-panic approach is usually sufficient—it's a crash after all—but if you want to display only the original value, you can write a little more code to filter unexpected problems and re-panic with the original error. That's left as an exercise for the reader.

A web server

Let's finish with a complete Go program, a web server. This one is actually a kind of web re-server. Google provides a service at http://chart.apis.google.com that does automatic formatting of data into charts and graphs. It's hard to use interactively, though, because you need to put the data into the URL as a query. The program here provides a nicer interface to one form of data: given a short piece of text, it calls on the chart server to produce a QR code, a matrix of boxes that encode the text. That image can be grabbed with your cell phone's camera and interpreted as, for instance, a URL, saving you typing the URL into the phone's tiny keyboard.

Here's the complete program. An explanation follows.

{{code "/doc/progs/eff_qr.go"}}

The pieces up to `main` should be easy to follow. The one flag sets a default HTTP port for our server. The template variable `templ` is where the fun happens. It builds an HTML template that will be executed by the server to display the page; more about that in a moment.

The `main` function parses the flags and, using the mechanism we talked about above, binds the function `QR` to the root path for the server. Then `http.ListenAndServe` is called to start the server; it blocks while the server runs.

`QR` just receives the request, which contains form data, and executes the template on the data in the form value named `s`.

The template package `html/template` is powerful; this program just touches on its capabilities. In essence, it rewrites a piece of HTML text on the fly by substituting elements derived from data items passed to `templ.Execute`, in this case the form value. Within the template text (`templateStr`), double-brace-delimited pieces denote template actions. The piece from `{{html "{{if .}}"}}` to `{{html "{{end}}"}}` executes only if the value of the current data item, called `.` (dot), is non-empty. That is, when the string is empty, this piece of the template is suppressed.

The two snippets `{{html "{{.}}"}}` say to show the data presented to the template—the query string—on the web page. The HTML template package automatically provides appropriate escaping so the text is safe to display.

The rest of the template string is just the HTML to show when the page loads. If this is too quick an explanation, see the documentation for the template package for a more thorough discussion.

And there you have it: a useful web server in a few lines of code plus some data-driven HTML text. Go is powerful enough to make a lot happen in a few lines.

Error Handling and Go

If you have written any Go code you have probably encountered the built-in `error` type. Go code uses `error` values to indicate an abnormal state. For example, the `os.Open` function returns a non-nil `error` value when it fails to open a file.

```
func Open(name string) (file *File, err error)
```

The following code uses `os.Open` to open a file. If an error occurs it calls `log.Fatal` to print the error message and stop.

```
        f, err := os.Open("filename.ext")
        if err != nil {
                log.Fatal(err)
        }
        // do something with the open *File f
```

You can get a lot done in Go knowing just this about the `error` type, but in this article we'll take a closer look at `error` and discuss some good practices for error handling in Go.

The error type

The `error` type is an interface type. An `error` variable represents any value that can describe itself as a string. Here is the interface's declaration:

```
type error interface {
        Error() string
}
```

The `error` type, as with all built in types, is predeclared in the universe block.

The most commonly-used `error` implementation is the errors package's unexported `errorString` type.

```
// errorString is a trivial implementation of error.
type errorString struct {
        s string
}

func (e *errorString) Error() string {
        return e.s
}
```

You can construct one of these values with the `errors.New` function. It takes a string that it converts to an `errors.errorString` and returns as an `error` value.

```
// New returns an error that formats as the given text.
func New(text string) error {
        return &errorString{text}
}
```

Here's how you might use `errors.New`:

```
func Sqrt(f float64) (float64, error) {
        if f < 0 {
                return 0, errors.New("math: square root of negative number")
        }
        // implementation
}
```

A caller passing a negative argument to `Sqrt` receives a non-nil `error` value (whose concrete representation is an `errors.errorString` value). The caller can access the error string ("math: square root of…") by calling the `error`'s `Error` method, or by just printing it:

```
        f, err := Sqrt(-1)
        if err != nil {
                fmt.Println(err)
        }
```

The fmt package formats an `error` value by calling its `Error() string` method.

It is the error implementation's responsibility to summarize the context. The error returned by `os.Open` formats as "open /etc/passwd: permission denied," not just "permission denied." The error returned by our `Sqrt` is missing information about the invalid argument.

To add that information, a useful function is the `fmt` package's `Errorf`. It formats a string according to `Printf`'s rules and returns it as an `error` created by `errors.New`.

```
        if f < 0 {
                return 0, fmt.Errorf("math: square root of negative number %g", f)
        }
```

In many cases `fmt.Errorf` is good enough, but since `error` is an interface, you can use arbitrary data structures as error values, to allow callers to inspect the details of the error.

For instance, our hypothetical callers might want to recover the invalid argument passed to `Sqrt`. We can enable that by defining a new error implementation instead of using `errors.errorString`:

```
type NegativeSqrtError float64

func (f NegativeSqrtError) Error() string {
        return fmt.Sprintf("math: square root of negative number %g", float64(f))
}
```

A sophisticated caller can then use a type assertion to check for a `NegativeSqrtError` and handle it specially, while callers that just pass the error to `fmt.Println` or `log.Fatal` will see no change in behavior.

As another example, the json package specifies a `SyntaxError` type that the `json.Decode` function returns when it encounters a syntax error parsing a JSON blob.

```
type SyntaxError struct {
        msg    string // description of error
        Offset int64  // error occurred after reading Offset bytes
}

func (e *SyntaxError) Error() string { return e.msg }
```

The `Offset` field isn't even shown in the default formatting of the error, but callers can use it to add file and line information to their error messages:

```
        if err := dec.Decode(&val); err != nil {
                if serr, ok := err.(*json.SyntaxError); ok {
                        line, col := findLine(f, serr.Offset)
                        return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, err)
                }
                return err
        }
```

(This is a slightly simplified version of some actual code from the Camlistore project.)

The `error` interface requires only a `Error` method; specific error implementations might have additional methods. For instance, the net package returns errors of type `error`, following the usual convention, but some of the error implementations have additional methods defined by the `net.Error` interface:

```
package net

type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

Client code can test for a `net.Error` with a type assertion and then distinguish transient network errors from permanent ones. For instance, a web crawler might sleep and retry when it encounters a temporary error and give up otherwise.

```
        if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
                time.Sleep(1e9)
                continue
        }
        if err != nil {
                log.Fatal(err)
        }
```

Simplifying repetitive error handling

In Go, error handling is important. The language's design and conventions encourage you to explicitly check for errors where they occur (as distinct from the convention in other languages of throwing exceptions and sometimes catching them). In some cases this makes Go code verbose, but fortunately there are some techniques you can use to minimize repetitive error handling.

Consider an App Engine application with an HTTP handler that retrieves a record from the datastore and formats it with a template.

```
func init() {
        http.HandleFunc("/view", viewRecord)
}

func viewRecord(w http.ResponseWriter, r *http.Request) {
        c := appengine.NewContext(r)
        key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
        record := new(Record)
        if err := datastore.Get(c, key, record); err != nil {
                http.Error(w, err.Error(), 500)
                return
        }
        if err := viewTemplate.Execute(w, record); err != nil {
```

```
                http.Error(w, err.Error(), 500)
        }
}
```

This function handles errors returned by the `datastore.Get` function and `viewTemplate`'s `Execute` method. In both cases, it presents a simple error message to the user with the HTTP status code 500 ("Internal Server Error"). This looks like a manageable amount of code, but add some more HTTP handlers and you quickly end up with many copies of identical error handling code.

To reduce the repetition we can define our own HTTP `appHandler` type that includes an `error` return value:

```
type appHandler func(http.ResponseWriter, *http.Request) error
```

Then we can change our `viewRecord` function to return errors:

```
func viewRecord(w http.ResponseWriter, r *http.Request) error {
        c := appengine.NewContext(r)
        key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
        record := new(Record)
        if err := datastore.Get(c, key, record); err != nil {
                return err
        }
        return viewTemplate.Execute(w, record)
}
```

This is simpler than the original version, but the http package doesn't understand functions that return `error`. To fix this we can implement the `http.Handler` interface's `ServeHTTP` method on `appHandler`:

```
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        if err := fn(w, r); err != nil {
                http.Error(w, err.Error(), 500)
        }
}
```

The `ServeHTTP` method calls the `appHandler` function and displays the returned error (if any) to the user. Notice that the method's receiver, `fn`, is a function. (Go can do that!) The method invokes the function by calling the receiver in the expression `fn(w, r)`.

Now when registering `viewRecord` with the http package we use the `Handle` function (instead of `HandleFunc`) as `appHandler` is an `http.Handler` (not an `http.HandlerFunc`).

```
func init() {
        http.Handle("/view", appHandler(viewRecord))
}
```

With this basic error handling infrastructure in place, we can make it more user friendly. Rather than just displaying the error string, it would be better to give the user a simple error message with an appropriate HTTP status code, while logging the full error to the App Engine developer console for debugging purposes.

To do this we create an `appError` struct containing an `error` and some other fields:

```
type appError struct {
        Error   error
        Message string
        Code    int
}
```

Next we modify the appHandler type to return `*appError` values:

```
type appHandler func(http.ResponseWriter, *http.Request) *appError
```

(It's usually a mistake to pass back the concrete type of an error rather than `error`, for reasons discussed in the Go FAQ, but it's the right thing to do here because `ServeHTTP` is the only place that sees the value and uses its contents.)

And make `appHandler`'s `ServeHTTP` method display the `appError`'s `Message` to the user with the correct HTTP status `Code` and log the full `Error` to the developer console:

```
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        if e := fn(w, r); e != nil { // e is *appError, not os.Error.
                c := appengine.NewContext(r)
                c.Errorf("%v", e.Error)
                http.Error(w, e.Message, e.Code)
        }
}
```

Finally, we update `viewRecord` to the new function signature and have it return more context when it encounters an error:

```
func viewRecord(w http.ResponseWriter, r *http.Request) *appError {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return &appError{err, "Record not found", 404}
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        return &appError{err, "Can't display record", 500}
    }
    return nil
}
```

This version of `viewRecord` is the same length as the original, but now each of those lines has specific meaning and we are providing a friendlier user experience.

It doesn't end there; we can further improve the error handling in our application. Some ideas:

- give the error handler a pretty HTML template,
- make debugging easier by writing the stack trace to the HTTP response when the user is an administrator,
- write a constructor function for `appError` that stores the stack trace for easier debugging,
- recover from panics inside the `appHandler`, logging the error to the console as "Critical," while telling the user "a serious error has occurred." This is a nice touch to avoid exposing the user to inscrutable error messages caused by programming errors. See the Defer, Panic, and Recover article for more details.

Conclusion

Proper error handling is an essential requirement of good software. By employing the techniques described in this post you should be able to write more reliable and succinct Go code.

Defer, Panic and Recover

Go has the usual mechanisms for control flow: if, for, switch, goto. It also has the go statement to run code in a separate goroutine. Here I'd like to discuss some of the less common ones: defer, panic, and recover.

A **defer statement** pushes a function call onto a list. The list of saved calls is executed after the surrounding function returns. Defer is commonly used to simplify functions that perform various clean-up actions.

For example, let's look at a function that opens two files and copies the contents of one file to the other:

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }

    written, err = io.Copy(dst, src)
    dst.Close()
    src.Close()
    return
}
```

This works, but there is a bug. If the call to os.Create fails, the function will return without closing the source file. This can be easily remedied by putting a call to src.Close before the second return statement, but if the function were more complex the problem might not be so easily noticed and resolved. By introducing defer statements we can ensure that the files are always closed:

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}
```

Defer statements allow us to think about closing each file right after opening it, guaranteeing that, regardless of the number of return statements in the function, the files *will* be closed.

The behavior of defer statements is straightforward and predictable. There are three simple rules:

1. *A deferred function's arguments are evaluated when the defer statement is evaluated.*

In this example, the expression "i" is evaluated when the Println call is deferred. The deferred call will print "0" after the function returns.

```
func a() {
    i := 0
    defer fmt.Println(i)
    i++
    return
}
```

2. *Deferred function calls are executed in Last In First Out order*after*the surrounding function returns.*

This function prints "3210":

```
func b() {
    for i := 0; i < 4; i++ {
        defer fmt.Print(i)
    }
}
```

3. *Deferred functions may read and assign to the returning function's named return values.*

In this example, a deferred function increments the return value i *after* the surrounding function returns. Thus, this function returns 2:

```
func c() (i int) {
    defer func() { i++ }()
    return 1
}
```

This is convenient for modifying the error return value of a function; we will see an example of this shortly.

Panic is a built-in function that stops the ordinary flow of control and begins *panicking*. When the function F calls panic, execution of F stops, any deferred functions in F are executed normally, and then F returns to its caller. To the caller, F then behaves like a call to panic. The process continues up the stack until all functions in the current goroutine have returned, at which point the program crashes. Panics can be initiated by invoking panic directly. They can also be caused by runtime errors, such as out-of-bounds array accesses.

Recover is a built-in function that regains control of a panicking goroutine. Recover is only useful inside deferred functions. During normal execution, a call to recover will return nil and have no other effect. If the current goroutine is panicking, a call to recover will capture the value given to panic and resume normal execution.

Here's an example program that demonstrates the mechanics of panic and defer:

```
package main

import "fmt"
import "io" // OMIT
import "os" // OMIT

func main() {
        f()
        fmt.Println("Returned normally from f.")
}

func f() {
        defer func() {
                if r := recover(); r != nil {
                        fmt.Println("Recovered in f", r)
                }
```

```
        }()
        fmt.Println("Calling g.")
        g(0)
        fmt.Println("Returned normally from g.")
}

func g(i int) {
        if i > 3 {
                fmt.Println("Panicking!")
                panic(fmt.Sprintf("%v", i))
        }
        defer fmt.Println("Defer in g", i)
        fmt.Println("Printing in g", i)
        g(i + 1)
}

// STOP OMIT

// Revised version.
func CopyFile(dstName, srcName string) (written int64, err error) {
        src, err := os.Open(srcName)
        if err != nil {
                return
        }
        defer src.Close()

        dst, err := os.Create(dstName)
        if err != nil {
                return
        }
        defer dst.Close()

        return io.Copy(dst, src)
}
```

The function g takes the int i, and panics if i is greater than 3, or else it calls itself with the argument i+1. The function f defers a function that calls recover and prints the recovered value (if it is non-nil). Try to picture what the output of this program might be before reading on.

The program will output:

```
Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
Recovered in f 4
Returned normally from f.
```

If we remove the deferred function from f the panic is not recovered and reaches the top of the goroutine's call stack, terminating the program. This modified program will output:

```
Calling g.
Printing in g 0
```

```
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
panic: 4

panic PC=0x2a9cd8
[stack trace omitted]
```

For a real-world example of **panic** and **recover**, see the json package from the Go standard library. It decodes JSON-encoded data with a set of recursive functions. When malformed JSON is encountered, the parser calls panic to unwind the stack to the top-level function call, which recovers from the panic and returns an appropriate error value (see the 'error' and 'unmarshal' methods of the decodeState type in decode.go).

The convention in the Go libraries is that even when a package uses panic internally, its external API still presents explicit error return values.

Other uses of **defer** (beyond the file.Close example given earlier) include releasing a mutex:

```
mu.Lock()
defer mu.Unlock()
```

printing a footer:

```
printHeader()
defer printFooter()
```

and more.

In summary, the defer statement (with or without panic and recover) provides an unusual and powerful mechanism for control flow. It can be used to model a number of features implemented by special-purpose structures in other programming languages. Try it out.

Go Concurrency Patterns

Timing out, moving on

Concurrent programming has its own idioms. A good example is timeouts. Although Go's channels do not support them directly, they are easy to implement. Say we want to receive from the channel `ch`, but want to wait at most one second for the value to arrive. We would start by creating a signalling channel and launching a goroutine that sleeps before sending on the channel:

```
timeout := make(chan bool, 1)
go func() {
        time.Sleep(1 * time.Second)
        timeout <- true
}()
```

We can then use a `select` statement to receive from either `ch` or `timeout`. If nothing arrives on `ch` after one second, the timeout case is selected and the attempt to read from `ch` is abandoned.

```
select {
case <-ch:
        // a read from ch has occurred
case <-timeout:
        // the read from ch has timed out
}
```

The `timeout` channel is buffered with space for 1 value, allowing the timeout goroutine to send to the channel and then exit. The goroutine doesn't know (or care) whether the value is received. This means the goroutine won't hang around forever if the `ch` receive happens before the timeout is reached. The `timeout` channel will eventually be deallocated by the garbage collector.

(In this example we used `time.Sleep` to demonstrate the mechanics of goroutines and channels. In real programs you should use `time.After`, a function that returns a channel and sends on that channel after the specified duration.)

Let's look at another variation of this pattern. In this example we have a program that reads from multiple replicated databases simultaneously. The program needs only one of the answers, and it should accept the answer that arrives first.

The function `Query` takes a slice of database connections and a `query` string. It queries each of the databases in parallel and returns the first response it receives:

```
func Query(conns []Conn, query string) Result {
        ch := make(chan Result, 1)
        for _, conn := range conns {
                go func(c Conn) {
                        select {
                        case ch <- c.DoQuery(query):
                        default:
                        }
                }(conn)
        }
```

```
        return <-ch
}
```

In this example, the closure does a non-blocking send, which it achieves by using the send operation in `select` statement with a `default` case. If the send cannot go through immediately the default case will be selected. Making the send non-blocking guarantees that none of the goroutines launched in the loop will hang around. However, if the result arrives before the main function has made it to the receive, the send could fail since no one is ready.

This problem is a textbook example of what is known as a race condition, but the fix is trivial. We just make sure to buffer the channel `ch` (by adding the buffer length as the second argument to make), guaranteeing that the first send has a place to put the value. This ensures the send will always succeed, and the first value to arrive will be retrieved regardless of the order of execution.

These two examples demonstrate the simplicity with which Go can express complex interactions between goroutines.

Gobs of Data

To transmit a data structure across a network or to store it in a file, it must be encoded and then decoded again. There are many encodings available, of course: JSON, XML, Google's protocol buffers, and more. And now there's another, provided by Go's gob package.

Why define a new encoding? It's a lot of work and redundant at that. Why not just use one of the existing formats? Well, for one thing, we do! Go has packages supporting all the encodings just mentioned (the protocol buffer package is in a separate repository but it's one of the most frequently downloaded). And for many purposes, including communicating with tools and systems written in other languages, they're the right choice.

But for a Go-specific environment, such as communicating between two servers written in Go, there's an opportunity to build something much easier to use and possibly more efficient.

Gobs work with the language in a way that an externally-defined, language-independent encoding cannot. At the same time, there are lessons to be learned from the existing systems.

Goals

The gob package was designed with a number of goals in mind.

First, and most obvious, it had to be very easy to use. First, because Go has reflection, there is no need for a separate interface definition language or "protocol compiler". The data structure itself is all the package should need to figure out how to encode and decode it. On the other hand, this approach means that gobs will never work as well with other languages, but that's OK: gobs are unashamedly Go-centric.

Efficiency is also important. Textual representations, exemplified by XML and JSON, are too slow to put at the center of an efficient communications network. A binary encoding is necessary.

Gob streams must be self-describing. Each gob stream, read from the beginning, contains sufficient information that the entire stream can be parsed by an agent that knows nothing a priori about its contents. This property means that you will always be able to decode a gob stream stored in a file, even long after you've forgotten what data it represents.

There were also some things to learn from our experiences with Google protocol buffers.

Protocol buffer misfeatures

Protocol buffers had a major effect on the design of gobs, but have three features that were deliberately avoided. (Leaving aside the property that protocol buffers aren't self-describing: if you don't know the data definition used to encode a protocol buffer, you might not be able to parse it.)

First, protocol buffers only work on the data type we call a struct in Go. You can't encode an integer or array at the top level, only a struct with fields inside it. That seems a pointless restriction, at least in Go. If all you want to send is an array of integers, why should you have to put it into a struct first?

Next, a protocol buffer definition may specify that fields `T.x` and `T.y` are required to be present whenever a value of type `T` is encoded or decoded. Although such required fields may seem like a good idea, they are costly to implement because the codec must maintain a separate data structure while encoding and decoding, to be able to report when required fields are missing. They're also a maintenance problem. Over time, one may want to modify the data definition to remove a required field, but that may cause existing clients of the data to crash. It's better not to have them in the encoding at all. (Protocol buffers also have optional fields. But if we don't have required fields, all fields are optional and that's that. There will be more to say about optional fields a little later.)

The third protocol buffer misfeature is default values. If a protocol buffer omits the value for a "defaulted" field, then the decoded structure behaves as if the field were set to that value. This idea works nicely when you have getter and setter methods to control access to the field, but is harder to handle cleanly when the container is just a plain

idiomatic struct. Required fields are also tricky to implement: where does one define the default values, what types do they have (is text UTF-8? uninterpreted bytes? how many bits in a float?) and despite the apparent simplicity, there were a number of complications in their design and implementation for protocol buffers. We decided to leave them out of gobs and fall back to Go's trivial but effective defaulting rule: unless you set something otherwise, it has the "zero value" for that type - and it doesn't need to be transmitted.

So gobs end up looking like a sort of generalized, simplified protocol buffer. How do they work?

Values

The encoded gob data isn't about `int8`s and `uint16`s. Instead, somewhat analogous to constants in Go, its integer values are abstract, sizeless numbers, either signed or unsigned. When you encode an `int8`, its value is transmitted as an unsized, variable-length integer. When you encode an `int64`, its value is also transmitted as an unsized, variable-length integer. (Signed and unsigned are treated distinctly, but the same unsized-ness applies to unsigned values too.) If both have the value 7, the bits sent on the wire will be identical. When the receiver decodes that value, it puts it into the receiver's variable, which may be of arbitrary integer type. Thus an encoder may send a 7 that came from an `int8`, but the receiver may store it in an `int64`. This is fine: the value is an integer and as a long as it fits, everything works. (If it doesn't fit, an error results.) This decoupling from the size of the variable gives some flexibility to the encoding: we can expand the type of the integer variable as the software evolves, but still be able to decode old data.

This flexibility also applies to pointers. Before transmission, all pointers are flattened. Values of type `int8`, `*int8`, `**int8`, `****int8`, etc. are all transmitted as an integer value, which may then be stored in `int` of any size, or `*int`, or `******int`, etc. Again, this allows for flexibility.

Flexibility also happens because, when decoding a struct, only those fields that are sent by the encoder are stored in the destination. Given the value

```
type T struct { X, Y, Z int } // Only exported fields are encoded and decoded.
var t = T{X: 7, Y: 0, Z: 8}
```

the encoding of `t` sends only the 7 and 8. Because it's zero, the value of `Y` isn't even sent; there's no need to send a zero value.

The receiver could instead decode the value into this structure:

```
type U struct{ X, Y *int8 } // Note: pointers to int8s
var u U
```

and acquire a value of `u` with only `X` set (to the address of an `int8` variable set to 7); the `Z` field is ignored - where would you put it? When decoding structs, fields are matched by name and compatible type, and only fields that exist in both are affected. This simple approach finesses the "optional field" problem: as the type `T` evolves by adding fields, out of date receivers will still function with the part of the type they recognize. Thus gobs provide the important result of optional fields - extensibility - without any additional mechanism or notation.

From integers we can build all the other types: bytes, strings, arrays, slices, maps, even floats. Floating-point values are represented by their IEEE 754 floating-point bit pattern, stored as an integer, which works fine as long as you know their type, which we always do. By the way, that integer is sent in byte-reversed order because common values of floating-point numbers, such as small integers, have a lot of zeros at the low end that we can avoid transmitting.

One nice feature of gobs that Go makes possible is that they allow you to define your own encoding by having your type satisfy the GobEncoder and GobDecoder interfaces, in a manner analogous to the JSON package's Marshaler and Unmarshaler and also to the Stringer interface from package fmt. This facility makes it possible to represent special features, enforce constraints, or hide secrets when you transmit data. See the documentation for details.

Types on the wire

The first time you send a given type, the gob package includes in the data stream a description of that type. In fact, what happens is that the encoder is used to encode, in the standard gob encoding format, an internal struct that describes the type and gives it a unique number. (Basic types, plus the layout of the type description structure, are predefined by the software for bootstrapping.) After the type is described, it can be referenced by its type number.

Thus when we send our first type `T`, the gob encoder sends a description of `T` and tags it with a type number, say 127. All values, including the first, are then prefixed by that number, so a stream of `T` values looks like:

```
("define type id" 127, definition of type T)(127, T value)(127, T value), ...
```

These type numbers make it possible to describe recursive types and send values of those types. Thus gobs can encode types such as trees:

```
type Node struct {
        Value       int
        Left, Right *Node
}
```

(It's an exercise for the reader to discover how the zero-defaulting rule makes this work, even though gobs don't represent pointers.)

With the type information, a gob stream is fully self-describing except for the set of bootstrap types, which is a well-defined starting point.

Compiling a machine

The first time you encode a value of a given type, the gob package builds a little interpreted machine specific to that data type. It uses reflection on the type to construct that machine, but once the machine is built it does not depend on reflection. The machine uses package unsafe and some trickery to convert the data into the encoded bytes at high speed. It could use reflection and avoid unsafe, but would be significantly slower. (A similar high-speed approach is taken by the protocol buffer support for Go, whose design was influenced by the implementation of gobs.) Subsequent values of the same type use the already-compiled machine, so they can be encoded right away.

Decoding is similar but harder. When you decode a value, the gob package holds a byte slice representing a value of a given encoder-defined type to decode, plus a Go value into which to decode it. The gob package builds a machine for that pair: the gob type sent on the wire crossed with the Go type provided for decoding. Once that decoding machine is built, though, it's again a reflectionless engine that uses unsafe methods to get maximum speed.

Use

There's a lot going on under the hood, but the result is an efficient, easy-to-use encoding system for transmitting data. Here's a complete example showing differing encoded and decoded types. Note how easy it is to send and receive values; all you need to do is present values and variables to the gob package and it does all the work.

```
package main

import (
        "bytes"
        "encoding/gob"
        "fmt"
        "log"
)

type P struct {
        X, Y, Z int
        Name    string
}

type Q struct {
        X, Y *int32
        Name string
}
```

```
func main() {
        // Initialize the encoder and decoder.  Normally enc and dec would be
        // bound to network connections and the encoder and decoder would
        // run in different processes.
        var network bytes.Buffer        // Stand-in for a network connection
        enc := gob.NewEncoder(&network) // Will write to network.
        dec := gob.NewDecoder(&network) // Will read from network.
        // Encode (send) the value.
        err := enc.Encode(P{3, 4, 5, "Pythagoras"})
        if err != nil {
                log.Fatal("encode error:", err)
        }
        // Decode (receive) the value.
        var q Q
        err = dec.Decode(&q)
        if err != nil {
                log.Fatal("decode error:", err)
        }
        fmt.Printf("%q: {%d,%d}\n", q.Name, *q.X, *q.Y)
}
```

You can compile and run this example code in the Go Playground.

The rpc package builds on gobs to turn this encode/decode automation into transport for method calls across the network. That's a subject for another article.

Details

The gob package documentation, especially the file doc.go, expands on many of the details described here and includes a full worked example showing how the encoding represents data. If you are interested in the innards of the gob implementation, that's a good place to start.

Go Language Specification

Introduction

This is a reference manual for the Go programming language. For more information and other documents, see http://golang.org.

Go is a general-purpose language designed with systems programming in mind. It is strongly typed and garbage-collected and has explicit support for concurrent programming. Programs are constructed from *packages*, whose properties allow efficient management of dependencies. The existing implementations use a traditional compile/link model to generate executable binaries.

The grammar is compact and regular, allowing for easy analysis by automatic tools such as integrated development environments.

Notation

The syntax is specified using Extended Backus-Naur Form (EBNF):

```
Production  = production_name "=" [ Expression ] "." .
Expression  = Alternative { "|" Alternative } .
Alternative = Term { Term } .
Term        = production_name | token [ "…" token ] | Group | Option | Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .
```

Productions are expressions constructed from terms and the following operators, in increasing precedence:

```
|   alternation
()  grouping
[]  option (0 or 1 times)
{}  repetition (0 to n times)
```

Lower-case production names are used to identify lexical tokens. Non-terminals are in CamelCase. Lexical tokens are enclosed in double quotes "" or back quotes ``.

The form `a … b` represents the set of characters from `a` through `b` as alternatives. The horizontal ellipsis … is also used elsewhere in the spec to informally denote various enumerations or code snippets that are not further specified. The character … (as opposed to the three characters `...`) is not a token of the Go language.

Source code representation

Source code is Unicode text encoded in UTF-8. The text is not canonicalized, so a single accented code point is distinct from the same character constructed from combining an accent and a letter; those are treated as two code points. For simplicity, this document will use the unqualified term *character* to refer to a Unicode code point in the source text.

Each code point is distinct; for instance, upper and lower case letters are different characters.

Implementation restriction: For compatibility with other tools, a compiler may disallow the NUL character (U+0000) in the source text.

Characters

The following terms are used to denote specific Unicode character classes:

```
newline        = /* the Unicode code point U+000A */ .
unicode_char   = /* an arbitrary Unicode code point except newline */ .
unicode_letter = /* a Unicode code point classified as "Letter" */ .
unicode_digit  = /* a Unicode code point classified as "Decimal Digit" */ .
```

In The Unicode Standard 6.0, Section 4.5 “General Category” defines a set of character categories. Go treats those characters in category Lu, Ll, Lt, Lm, or Lo as Unicode letters, and those in category Nd as Unicode digits.

Letters and digits

The underscore character _ (U+005F) is considered a letter.

```
letter        = unicode_letter | "_" .
decimal_digit = "0" … "9" .
octal_digit   = "0" … "7" .
hex_digit     = "0" … "9" | "A" … "F" | "a" … "f" .
```

Lexical elements

Comments

There are two forms of comments:

1. *Line comments* start with the character sequence `//` and stop at the end of the line. A line comment acts like a newline.
2. *General comments* start with the character sequence `/*` and continue through the character sequence `*/`. A general comment containing one or more newlines acts like a newline, otherwise it acts like a space.

Comments do not nest.

Tokens

Tokens form the vocabulary of the Go language. There are four classes: *identifiers*, *keywords*, *operators and delimiters*, and *literals*. *White space*, formed from spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token. Also, a newline or end of file may trigger the insertion of a semicolon. While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

Semicolons

The formal grammar uses semicolons `";"` as terminators in a number of productions. Go programs may omit most of these semicolons using the following two rules:

1. When the input is broken into tokens, a semicolon is automatically inserted into the token stream at the end of a non-blank line if the line's final token is
   - an identifier
   - an integer, floating-point, imaginary, rune, or string literal
   - one of the keywords `break`, `continue`, `fallthrough`, or `return`
   - one of the operators and delimiters `++`, `--`, `)`, `]`, or `}`
2. To allow complex statements to occupy a single line, a semicolon may be omitted before a closing `")"` or `"}"`.

To reflect idiomatic use, code examples in this document elide semicolons using these rules.

Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier = letter { letter | unicode_digit } .
```

```
a
_x9
ThisVariableIsExported
```

Some identifiers are predeclared.

Keywords

The following keywords are reserved and may not be used as identifiers.

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Operators and Delimiters

The following character sequences represent operators, delimiters, and other special tokens:

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=
```

Integer literals

An integer literal is a sequence of digits representing an integer constant. An optional prefix sets a non-decimal base: `0` for octal, `0x` or `0X` for hexadecimal. In hexadecimal literals, letters `a-f` and `A-F` represent values 10 through 15.

```
int_lit     = decimal_lit | octal_lit | hex_lit .
decimal_lit = ( "1" … "9" ) { decimal_digit } .
octal_lit   = "0" { octal_digit } .
hex_lit     = "0" ( "x" | "X" ) hex_digit { hex_digit } .
```

```
42
0600
0xBadFace
170141183460469231731687303715884105727
```

Floating-point literals

A floating-point literal is a decimal representation of a floating-point constant. It has an integer part, a decimal point, a fractional part, and an exponent part. The integer and fractional part comprise decimal digits; the exponent part is an `e` or `E` followed by an optionally signed decimal exponent. One of the integer part or the fractional part may be elided; one of the decimal point or the exponent may be elided.

```
float_lit = decimals "." [ decimals ] [ exponent ] |
            decimals exponent |
            "." decimals [ exponent ] .
decimals  = decimal_digit { decimal_digit } .
exponent  = ( "e" | "E" ) [ "+" | "-" ] decimals .
```

```
0.
72.40
072.40  // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
```

Imaginary literals

An imaginary literal is a decimal representation of the imaginary part of a complex constant. It consists of a floating-point literal or decimal integer followed by the lower-case letter `i`.

```
imaginary_lit = (decimals | float_lit) "i" .
```

```
0i
011i  // == 11i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i
```

Rune literals

A rune literal represents a rune constant, an integer value identifying a Unicode code point. A rune literal is expressed as one or more characters enclosed in single quotes. Within the quotes, any character may appear except single quote and newline. A single quoted character represents the Unicode value of the character itself, while multi-character sequences beginning with a backslash encode values in various formats.

The simplest form represents the single character within the quotes; since Go source text is Unicode characters encoded in UTF-8, multiple UTF-8-encoded bytes may represent a single integer value. For instance, the literal `'a'` holds a single byte representing a literal `a`, Unicode U+0061, value `0x61`, while `'ä'` holds two bytes (`0xc3 0xa4`) representing a literal `a`-dieresis, U+00E4, value `0xe4`.

Several backslash escapes allow arbitrary values to be encoded as ASCII text. There are four ways to represent the integer value as a numeric constant: `\x` followed by exactly two hexadecimal digits; `\u` followed by exactly four hexadecimal digits; `\U` followed by exactly eight hexadecimal digits, and a plain backslash `\` followed by exactly three octal digits. In each case the value of the literal is the value represented by the digits in the corresponding base.

Although these representations all result in an integer, they have different valid ranges. Octal escapes must represent a value between 0 and 255 inclusive. Hexadecimal escapes satisfy this condition by construction. The escapes `\u` and `\U` represent Unicode code points so within them some values are illegal, in particular those above `0x10FFFF` and surrogate halves.

After a backslash, certain single-character escapes represent special values:

```
\a   U+0007 alert or bell
\b   U+0008 backspace
```

```
\f   U+000C form feed
\n   U+000A line feed or newline
\r   U+000D carriage return
\t   U+0009 horizontal tab
\v   U+000b vertical tab
\\   U+005c backslash
\'   U+0027 single quote  (valid escape only within rune literals)
\"   U+0022 double quote  (valid escape only within string literals)
```

All other sequences starting with a backslash are illegal inside rune literals.

```
char_lit         = "'" ( unicode_value | byte_value ) "'" .
unicode_value    = unicode_char | little_u_value | big_u_value | escaped_char .
byte_value       = octal_byte_value | hex_byte_value .
octal_byte_value = `\` octal_digit octal_digit octal_digit .
hex_byte_value   = `\` "x" hex_digit hex_digit .
little_u_value   = `\` "u" hex_digit hex_digit hex_digit hex_digit .
big_u_value      = `\` "U" hex_digit hex_digit hex_digit hex_digit
                           hex_digit hex_digit hex_digit hex_digit .
escaped_char     = `\` ( "a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"` ) .
```

```
'a'
'ä'
' '
'\t'
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
'aa'         // illegal: too many characters
'\xa'        // illegal: too few hexadecimal digits
'\0'         // illegal: too few octal digits
'\uDFFF'     // illegal: surrogate half
'\U00110000' // illegal: invalid Unicode code point
```

String literals

A string literal represents a string constant obtained from concatenating a sequence of characters. There are two forms: raw string literals and interpreted string literals.

Raw string literals are character sequences between back quotes ``. Within the quotes, any character is legal except back quote. The value of a raw string literal is the string composed of the uninterpreted (implicitly UTF-8-encoded) characters between the quotes; in particular, backslashes have no special meaning and the string may contain newlines. Carriage returns inside raw string literals are discarded from the raw string value.

Interpreted string literals are character sequences between double quotes "". The text between the quotes, which may not contain newlines, forms the value of the literal, with backslash escapes interpreted as they are in rune literals (except that \' is illegal and \" is legal), with the same restrictions. The three-digit octal (*nnn*) and two-digit hexadecimal (\x*nn*) escapes represent individual *bytes* of the resulting string; all other escapes represent the (possibly multi-byte) UTF-8 encoding of individual *characters*. Thus inside a string literal \377 and \xFF represent a single byte of value 0xFF=255, while ÿ, \u00FF, \U000000FF and \xc3\xbf represent the two bytes 0xc3 0xbf of the UTF-8 encoding of character U+00FF.

```
string_lit             = raw_string_lit | interpreted_string_lit .
raw_string_lit         = "`" { unicode_char | newline } "`" .
interpreted_string_lit = `"` { unicode_value | byte_value } `"` .
```

```
`abc`  // same as "abc"
`\n
\n`    // same as "\\n\n\\n"
"\n"
""
"Hello, world!\n"
"  "
"\u65e5 \U00008a9e"
"\xff\u00FF"
"\uD800"       // illegal: surrogate half
"\U00110000"   // illegal: invalid Unicode code point
```

These examples all represent the same string:

```
"  "                                 // UTF-8 input text
`  `                                 // UTF-8 input text as a raw literal
"\u65e5\u672c\u8a9e"                    // the explicit Unicode code points
"\U000065e5\U0000672c\U00008a9e"        // the explicit Unicode code points
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"  // the explicit UTF-8 bytes
```

If the source code represents a character as two code points, such as a combining form involving an accent and a letter, the result will be an error if placed in a rune literal (it is not a single code point), and will appear as two code points if placed in a string literal.

Constants

There are *boolean constants*, *rune constants*, *integer constants*, *floating-point constants*, *complex constants*, and *string constants*. Character, integer, floating-point, and complex constants are collectively called *numeric constants*.

A constant value is represented by a rune, integer, floating-point, imaginary, or string literal, an identifier denoting a constant, a constant expression, a conversion with a result that is a constant, or the result value of some built-in functions such as `unsafe.Sizeof` applied to any value, `cap` or `len` applied to some expressions, `real` and `imag` applied to a complex constant and `complex` applied to numeric constants. The boolean truth values are represented by the predeclared constants `true` and `false`. The predeclared identifier iota denotes an integer constant.

In general, complex constants are a form of constant expression and are discussed in that section.

Numeric constants represent values of arbitrary precision and do not overflow.

Constants may be typed or untyped. Literal constants, `true`, `false`, `iota`, and certain constant expressions containing only untyped constant operands are untyped.

A constant may be given a type explicitly by a constant declaration or conversion, or implicitly when used in a variable declaration or an assignment or as an operand in an expression. It is an error if the constant value cannot be represented as a value of the respective type. For instance, `3.0` can be given any integer or any floating-point type, while `2147483648.0` (equal to `1<<31`) can be given the types `float32`, `float64`, or `uint32` but not `int32` or `string`.

There are no constants denoting the IEEE-754 infinity and not-a-number values, but the `math` package's Inf, NaN, IsInf, and IsNaN functions return and test for those values at run time.

Implementation restriction: Although numeric constants have arbitrary precision in the language, a compiler may implement them using an internal representation with limited precision. That said, every implementation must:

- Represent integer constants with at least 256 bits.
- Represent floating-point constants, including the parts of a complex constant, with a mantissa of at least 256 bits and a signed exponent of at least 32 bits.
- Give an error if unable to represent an integer constant precisely.
- Give an error if unable to represent a floating-point or complex constant due to overflow.

- Round to the nearest representable constant if unable to represent a floating-point or complex constant due to limits on precision.

These requirements apply both to literal constants and to the result of evaluating constant expressions.

Types

A type determines the set of values and operations specific to values of that type. A type may be specified by a (possibly qualified) *type name* a *type literal*, which composes a new type from previously declared types.

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeLit   = ArrayType | StructType | PointerType | FunctionType | InterfaceType |
        SliceType | MapType | ChannelType .
```

Named instances of the boolean, numeric, and string types are predeclared. *Composite types*—array, struct, pointer, function, interface, slice, map, and channel types—may be constructed using type literals.

The *static type* (or just *type*) of a variable is the type defined by its declaration. Variables of interface type also have a distinct *dynamic type*, which is the actual type of the value stored in the variable at run-time. The dynamic type may vary during execution but is always assignable to the static type of the interface variable. For non-interface types, the dynamic type is always the static type.

Each type `T` has an *underlying type*: If `T` is a predeclared type or a type literal, the corresponding underlying type is `T` itself. Otherwise, `T`'s underlying type is the underlying type of the type to which `T` refers in its type declaration.

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

The underlying type of `string`, `T1`, and `T2` is `string`. The underlying type of `[]T1`, `T3`, and `T4` is `[]T1`.

Method sets

A type may have a *method set* associated with it. The method set of an interface type is its interface. The method set of any other type `T` consists of all methods with receiver type `T`. The method set of the corresponding pointer type `*T` is the set of all methods with receiver `*T` or `T` (that is, it also contains the method set of `T`). Further rules apply to structs containing anonymous fields, as described in the section on struct types. Any other type has an empty method set. In a method set, each method must have a unique method name.

The method set of a type determines the interfaces that the type implements and the methods that can be called using a receiver of that type.

Boolean types

A *boolean type* represents the set of Boolean truth values denoted by the predeclared constants `true` and `false`. The predeclared boolean type is `bool`.

Numeric types

A *numeric type* represents sets of integer or floating-point values. The predeclared architecture-independent numeric types are:

```
uint8       the set of all unsigned  8-bit integers (0 to 255)
uint16      the set of all unsigned 16-bit integers (0 to 65535)
uint32      the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      the set of all unsigned 64-bit integers (0 to 18446744073709551615)
```

```
int8        the set of all signed  8-bit integers (-128 to 127)
int16       the set of all signed 16-bit integers (-32768 to 32767)
int32       the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64       the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

float32     the set of all IEEE-754 32-bit floating-point numbers
float64     the set of all IEEE-754 64-bit floating-point numbers

complex64   the set of all complex numbers with float32 real and imaginary parts
complex128  the set of all complex numbers with float64 real and imaginary parts

byte        alias for uint8
rune        alias for int32
```

The value of an n-bit integer is n bits wide and represented using two's complement arithmetic.

There is also a set of predeclared numeric types with implementation-specific sizes:

```
uint     either 32 or 64 bits
int      same size as uint
uintptr  an unsigned integer large enough to store the uninterpreted bits of a pointer value
```

To avoid portability issues all numeric types are distinct except `byte`, which is an alias for `uint8`, and `rune`, which is an alias for `int32`. Conversions are required when different numeric types are mixed in an expression or assignment. For instance, `int32` and `int` are not the same type even though they may have the same size on a particular architecture.

String types

A *string type* represents the set of string values. Strings behave like slices of bytes but are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is `string`.

The elements of strings have type `byte` and may be accessed using the usual indexing operations. It is illegal to take the address of such an element; if `s[i]` is the ith byte of a string, `&s[i]` is invalid. The length of string `s` can be discovered using the built-in function `len`. The length is a compile-time constant if `s` is a string literal.

Array types

An array is a numbered sequence of elements of a single type, called the element type. The number of elements is called the length and is never negative.

```
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
```

The length is part of the array's type and must be a constant expression that evaluates to a non-negative integer value. The length of array `a` can be discovered using the built-in function `len(a)`. The elements can be indexed by integer indices 0 through `len(a)-1`. Array types are always one-dimensional but may be composed to form multi-dimensional types.

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64  // same as [2]([2]([2]float64))
```

Slice types

A slice is a reference to a contiguous segment of an array and contains a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The value of an uninitialized slice is `nil`.

```
SliceType = "[" "]" ElementType .
```

Like arrays, slices are indexable and have a length. The length of a slice `s` can be discovered by the built-in function `len(s)`; unlike with arrays it may change during execution. The elements can be addressed by integer indices 0 through `len(s)-1`. The slice index of a given element may be less than the index of the same element in the underlying array.

A slice, once initialized, is always associated with an underlying array that holds its elements. A slice therefore shares storage with its array and with other slices of the same array; by contrast, distinct arrays always represent distinct storage.

The array underlying a slice may extend past the end of the slice. The *capacity* is a measure of that extent: it is the sum of the length of the slice and the length of the array beyond the slice; a slice of length up to that capacity can be created by 'slicing' a new one from the original slice. The capacity of a slice `a` can be discovered using the built-in function `cap(a)`.

A new, initialized slice value for a given element type `T` is made using the built-in function `make`, which takes a slice type and parameters specifying the length and optionally the capacity:

```
make([]T, length)
make([]T, length, capacity)
```

A call to `make` allocates a new, hidden array to which the returned slice value refers. That is, executing

```
make([]T, length, capacity)
```

produces the same slice as allocating an array and slicing it, so these two examples result in the same slice:

```
make([]int, 50, 100)
new([100]int)[0:50]
```

Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. With arrays of arrays, the inner arrays are, by construction, always the same length; however with slices of slices (or arrays of slices), the lengths may vary dynamically. Moreover, the inner slices must be allocated individually (with `make`).

Struct types

A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly (AnonymousField). Within a struct, non-blank field names must be unique.

```
StructType     = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl      = (IdentifierList Type | AnonymousField) [ Tag ] .
AnonymousField = [ "*" ] TypeName .
Tag            = string_lit .
```

```
// An empty struct.
struct {}

// A struct with 6 fields.
struct {
    x, y int
    u float32
```

```
    _ float32  // padding
    A *[]int
    F func()
}
```

A field declared with a type but no explicit field name is an *anonymous field*, also called an *embedded* field or an embedding of the type in the struct. An embedded type must be specified as a type name `T` or as a pointer to a non-interface type name `*T`, and `T` itself may not be a pointer type. The unqualified type name acts as the field name.

```
// A struct with four anonymous fields of type T1, *T2, P.T3 and *P.T4
struct {
    T1        // field name is T1
    *T2       // field name is T2
    P.T3      // field name is T3
    *P.T4     // field name is T4
    x, y int  // field names are x and y
}
```

The following declaration is illegal because field names must be unique in a struct type:

```
struct {
    T     // conflicts with anonymous field *T and *P.T
    *T    // conflicts with anonymous field T and *P.T
    *P.T  // conflicts with anonymous field T and *T
}
```

A field or method `f` of an anonymous field in a struct `x` is called *promoted* if `x.f` is a legal selector that denotes that field or method `f`.

Promoted fields act like ordinary fields of a struct except that they cannot be used as field names in composite literals of the struct.

Given a struct type `S` and a type named `T`, promoted methods are included in the method set of the struct as follows:

- If `S` contains an anonymous field `T`, the method sets of `S` and `*S` both include promoted methods with receiver `T`. The method set of `*S` also includes promoted methods with receiver `*T`.
- If `S` contains an anonymous field `*T`, the method sets of `S` and `*S` both include promoted methods with receiver `T` or `*T`.

A field declaration may be followed by an optional string literal *tag*, which becomes an attribute for all the fields in the corresponding field declaration. The tags are made visible through a reflection interface but are otherwise ignored.

```
// A struct corresponding to the TimeStamp protocol buffer.
// The tag strings define the protocol buffer field numbers.
struct {
    microsec  uint64 "field 1"
    serverIP6 uint64 "field 2"
    process   string "field 3"
}
```

Pointer types

A pointer type denotes the set of all pointers to variables of a given type, called the *base type* of the pointer. The value of an uninitialized pointer is `nil`.

```
PointerType = "*" BaseType .
BaseType = Type .
```

```
*Point
*[4]int
```

Function types

A function type denotes the set of all functions with the same parameter and result types. The value of an uninitialized variable of function type is `nil`.

```
FunctionType   = "func" Signature .
Signature      = Parameters [ Result ] .
Result         = Parameters | Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
ParameterList  = ParameterDecl { "," ParameterDecl } .
ParameterDecl  = [ IdentifierList ] [ "..." ] Type .
```

Within a list of parameters or results, the names (IdentifierList) must either all be present or all be absent. If present, each name stands for one item (parameter or result) of the specified type; if absent, each type stands for one item of that type. Parameter and result lists are always parenthesized except that if there is exactly one unnamed result it may be written as an unparenthesized type.

The final parameter in a function signature may have a type prefixed with `...`. A function with such a parameter is called *variadic* and may be invoked with zero or more arguments for that parameter.

```
func()
func(x int) int
func(a, _ int, z float32) bool
func(a, b int, z float32) (bool)
func(prefix string, values ...int)
func(a, b int, z float64, opt ...interface{}) (success bool)
func(int, int, float64) (float64, *[]int)
func(n int) func(p *T)
```

Interface types

An interface type specifies a method set called its *interface*. A variable of interface type can store a value of any type with a method set that is any superset of the interface. Such a type is said to *implement the interface*. The value of an uninitialized variable of interface type is `nil`.

```
InterfaceType      = "interface" "{" { MethodSpec ";" } "}" .
MethodSpec         = MethodName Signature | InterfaceTypeName .
MethodName         = identifier .
InterfaceTypeName  = TypeName .
```

As with all method sets, in an interface type, each method must have a unique name.

```
// A simple File interface
interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
    Close()
}
```

More than one type may implement an interface. For instance, if two types `S1` and `S2` have the method set

```
func (p T) Read(b Buffer) bool { return … }
func (p T) Write(b Buffer) bool { return … }
func (p T) Close() { … }
```

(where `T` stands for either `S1` or `S2`) then the `File` interface is implemented by both `S1` and `S2`, regardless of what other methods `S1` and `S2` may have or share.

A type implements any interface comprising any subset of its methods and may therefore implement several distinct interfaces. For instance, all types implement the *empty interface*:

```
interface{}
```

Similarly, consider this interface specification, which appears within a type declaration to define an interface called `Lock`:

```
type Lock interface {
    Lock()
    Unlock()
}
```

If `S1` and `S2` also implement

```
func (p T) Lock() { … }
func (p T) Unlock() { … }
```

they implement the `Lock` interface as well as the `File` interface.

An interface may use an interface type name `T` in place of a method specification. The effect, called embedding an interface, is equivalent to enumerating the methods of `T` explicitly in the interface.

```
type ReadWrite interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
}

type File interface {
    ReadWrite  // same as enumerating the methods in ReadWrite
    Lock       // same as enumerating the methods in Lock
    Close()
}
```

An interface type `T` may not embed itself or any interface type that embeds `T`, recursively.

```
// illegal: Bad cannot embed itself
type Bad interface {
    Bad
}

// illegal: Bad1 cannot embed itself using Bad2
type Bad1 interface {
    Bad2
}
type Bad2 interface {
    Bad1
}
```

Map types

A map is an unordered group of elements of one type, called the element type, indexed by a set of unique *keys* of another type, called the key type. The value of an uninitialized map is `nil`.

```
MapType     = "map" "[" KeyType "]" ElementType .
KeyType     = Type .
```

The comparison operators `==` and `!=` must be fully defined for operands of the key type; thus the key type must not be a function, map, or slice. If the key type is an interface type, these comparison operators must be defined for the dynamic key values; failure will cause a run-time panic.

```
map[string]int
map[*T]struct{ x, y float64 }
map[string]interface{}
```

The number of map elements is called its length. For a map `m`, it can be discovered using the built-in function `len(m)` and may change during execution. Elements may be added during execution using assignments and retrieved with index expressions; they may be removed with the `delete` built-in function.

A new, empty map value is made using the built-in function `make`, which takes the map type and an optional capacity hint as arguments:

```
make(map[string]int)
make(map[string]int, 100)
```

The initial capacity does not bound its size: maps grow to accommodate the number of items stored in them, with the exception of `nil` maps. A `nil` map is equivalent to an empty map except that no elements may be added.

Channel types

A channel provides a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type. The value of an uninitialized channel is `nil`.

```
ChannelType = ( "chan" [ "<-" ] | "<-" "chan" ) ElementType .
```

The `<-` operator specifies the channel *direction*, *send* or *receive*. If no direction is given, the channel is *bi-directional*. A channel may be constrained only to send or only to receive by conversion or assignment.

```
chan T          // can be used to send and receive values of type T
chan<- float64  // can only be used to send float64s
<-chan int      // can only be used to receive ints
```

The `<-` operator associates with the leftmost `chan` possible:

```
chan<- chan int    // same as chan<- (chan int)
chan<- <-chan int  // same as chan<- (<-chan int)
<-chan <-chan int  // same as <-chan (<-chan int)
chan (<-chan int)
```

A new, initialized channel value can be made using the built-in function `make`, which takes the channel type and an optional capacity as arguments:

```
make(chan int, 100)
```

The capacity, in number of elements, sets the size of the buffer in the channel. If the capacity is greater than zero, the channel is asynchronous: communication operations succeed without blocking if the buffer is not full (sends) or not empty (receives), and elements are received in the order they are sent. If the capacity is zero or absent, the communication succeeds only when both a sender and receiver are ready. A `nil` channel is never ready for communication.

A channel may be closed with the built-in function `close`; the multi-valued assignment form of the receive operator tests whether a channel has been closed.

Properties of types and values

Type identity

Two types are either *identical* or *different*.

Two named types are identical if their type names originate in the same TypeSpec. A named and an unnamed type are always different. Two unnamed types are identical if the corresponding type literals are identical, that is, if they have the same literal structure and corresponding components have identical types. In detail:

- Two array types are identical if they have identical element types and the same array length.
- Two slice types are identical if they have identical element types.
- Two struct types are identical if they have the same sequence of fields, and if corresponding fields have the same names, and identical types, and identical tags. Two anonymous fields are considered to have the same name. Lower-case field names from different packages are always different.
- Two pointer types are identical if they have identical base types.
- Two function types are identical if they have the same number of parameters and result values, corresponding parameter and result types are identical, and either both functions are variadic or neither is. Parameter and result names are not required to match.
- Two interface types are identical if they have the same set of methods with the same names and identical function types. Lower-case method names from different packages are always different. The order of the methods is irrelevant.
- Two map types are identical if they have identical key and value types.
- Two channel types are identical if they have identical value types and the same direction.

Given the declarations

```
type (
    T0 []string
    T1 []string
    T2 struct{ a, b int }
    T3 struct{ a, c int }
    T4 func(int, float64) *T0
    T5 func(x int, y float64) *[]string
)
```

these types are identical:

```
T0 and T0
[]int and []int
struct{ a, b *T5 } and struct{ a, b *T5 }
func(x int, y float64) *[]string and func(int, float64) (result *[]string)
```

`T0` and `T1` are different because they are named types with distinct declarations; `func(int, float64) *T0` and `func(x int, y float64) *[]string` are different because `T0` is different from `[]string`.

Assignability

A value `x` is *assignable* to a variable of type `T` ("`x` is assignable to `T`") in any of these cases:

- `x`'s type is identical to `T`.
- `x`'s type `V` and `T` have identical underlying types and at least one of `V` or `T` is not a named type.
- `T` is an interface type and `x` implements `T`.

- `x` is a bidirectional channel value, `T` is a channel type, `x`'s type `V` and `T` have identical element types, and at least one of `V` or `T` is not a named type.
- `x` is the predeclared identifier `nil` and `T` is a pointer, function, slice, map, channel, or interface type.
- `x` is an untyped constant representable by a value of type `T`.

Any value may be assigned to the blank identifier.

Blocks

A *block* is a sequence of declarations and statements within matching brace brackets.

```
Block = "{" { Statement ";" } "}" .
```

In addition to explicit blocks in the source code, there are implicit blocks:

1. The *universe block* encompasses all Go source text.
2. Each package has a *package* block containing all Go source text for that package.
3. Each file has a *file block* containing all Go source text in that file.
4. Each `if`, `for`, and `switch` statement is considered to be in its own implicit block.
5. Each clause in a `switch` or `select` statement acts as an implicit block.

Blocks nest and influence scoping.

Declarations and scope

A declaration binds a non-blank identifier to a constant, type, variable, function, or package. Every identifier in a program must be declared. No identifier may be declared twice in the same block, and no identifier may be declared in both the file and package block.

```
Declaration   = ConstDecl | TypeDecl | VarDecl .
TopLevelDecl  = Declaration | FunctionDecl | MethodDecl .
```

The *scope* of a declared identifier is the extent of source text in which the identifier denotes the specified constant, type, variable, function, or package.

Go is lexically scoped using blocks:

1. The scope of a predeclared identifier is the universe block.
2. The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at top level (outside any function) is the package block.
3. The scope of an imported package identifier is the file block of the file containing the import declaration.
4. The scope of an identifier denoting a function parameter or result variable is the function body.
5. The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec (ShortVarDecl for short variable declarations) and ends at the end of the innermost containing block.
6. The scope of a type identifier declared inside a function begins at the identifier in the TypeSpec and ends at the end of the innermost containing block.

An identifier declared in a block may be redeclared in an inner block. While the identifier of the inner declaration is in scope, it denotes the entity declared by the inner declaration.

The package clause is not a declaration; the package name does not appear in any scope. Its purpose is to identify the files belonging to the same package and to specify the default package name for import declarations.

Label scopes

Labels are declared by labeled statements and are used in the `break`, `continue`, and `goto` statements (`break` statements, `continue` statements, `goto` statements). It is illegal to define a label that is never used. In contrast to other identifiers, labels are not block scoped and do not conflict with identifiers that are not labels. The scope of a label is the body of the function in which it is declared and excludes the body of any nested function.

Blank identifier

The *blank identifier*, represented by the underscore character `_`, may be used in a declaration like any other identifier but the declaration does not introduce a new binding.

Predeclared identifiers

The following identifiers are implicitly declared in the universe block:

```
Types:
    bool byte complex64 complex128 error float32 float64
    int int8 int16 int32 int64 rune string
    uint uint8 uint16 uint32 uint64 uintptr

Constants:
    true false iota

Zero value:
    nil

Functions:
    append cap close complex copy delete imag len
    make new panic print println real recover
```

Exported identifiers

An identifier may be *exported* to permit access to it from another package. An identifier is exported if both:

1. the first character of the identifier's name is a Unicode upper case letter (Unicode class "Lu"); and
2. the identifier is declared in the package block or it is a field name or method name.

All other identifiers are not exported.

Uniqueness of identifiers

Given a set of identifiers, an identifier is called *unique* if it is *different* from every other in the set. Two identifiers are different if they are spelled differently, or if they appear in different packages and are not exported. Otherwise, they are the same.

Constant declarations

A constant declaration binds a list of identifiers (the names of the constants) to the values of a list of constant expressions. The number of identifiers must be equal to the number of expressions, and the nth identifier on the left is bound to the value of the nth expression on the right.

```
ConstDecl      = "const" ( ConstSpec | "(" { ConstSpec ";" } ")" ) .
ConstSpec      = IdentifierList [ [ Type ] "=" ExpressionList ] .

IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .
```

If the type is present, all constants take the type specified, and the expressions must be assignable to that type. If the type is omitted, the constants take the individual types of the corresponding expressions. If the expression values are untyped constants, the declared constants remain untyped and the constant identifiers denote the constant values. For instance, if the expression is a floating-point literal, the constant identifier denotes a floating-point constant, even if the literal's fractional part is zero.

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0         // untyped floating-point constant
const (
    size int64 = 1024
    eof        = -1  // untyped integer constant
)
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo", untyped integer and string constants
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

Within a parenthesized `const` declaration list the expression list may be omitted from any but the first declaration. Such an empty list is equivalent to the textual substitution of the first preceding non-empty expression list and its type if any. Omitting the list of expressions is therefore equivalent to repeating the previous list. The number of identifiers must be equal to the number of expressions in the previous list. Together with the `iota constant generator` this mechanism permits light-weight declaration of sequential values:

```
const (
    Sunday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Partyday
    numberOfDays  // this constant is not exported
)
```

Iota

Within a constant declaration, the predeclared identifier `iota` represents successive untyped integer constants. It is reset to 0 whenever the reserved word `const` appears in the source and increments after each ConstSpec. It can be used to construct a set of related constants:

```
const (  // iota is reset to 0
    c0 = iota  // c0 == 0
    c1 = iota  // c1 == 1
    c2 = iota  // c2 == 2
)

const (
    a = 1 << iota  // a == 1 (iota has been reset)
    b = 1 << iota  // b == 2
    c = 1 << iota  // c == 4
)

const (
    u         = iota * 42  // u == 0     (untyped integer constant)
    v float64 = iota * 42  // v == 42.0  (float64 constant)
    w         = iota * 42  // w == 84    (untyped integer constant)
)
```

```
const x = iota  // x == 0 (iota has been reset)
const y = iota  // y == 0 (iota has been reset)
```

Within an ExpressionList, the value of each `iota` is the same because it is only incremented after each ConstSpec:

```
const (
    bit0, mask0 = 1 << iota, 1<<iota - 1  // bit0 == 1, mask0 == 0
    bit1, mask1                           // bit1 == 2, mask1 == 1
    _, _                                  // skips iota == 2
    bit3, mask3                           // bit3 == 8, mask3 == 7
)
```

This last example exploits the implicit repetition of the last non-empty expression list.

Type declarations

A type declaration binds an identifier, the *type name*, to a new type that has the same underlying type as an existing type. The new type is different from the existing type.

```
TypeDecl     = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec     = identifier Type .
```

```
type IntArray [16]int

type (
    Point struct{ x, y float64 }
    Polar Point
)

type TreeNode struct {
    left, right *TreeNode
    value *Comparable
}

type Block interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}
```

The declared type does not inherit any methods bound to the existing type, but the method set of an interface type or of elements of a composite type remains unchanged:

```
// A Mutex is a data type with two methods, Lock and Unlock.
type Mutex struct         { /* Mutex fields */ }
func (m *Mutex) Lock()    { /* Lock implementation */ }
func (m *Mutex) Unlock()  { /* Unlock implementation */ }

// NewMutex has the same composition as Mutex but its method set is empty.
type NewMutex Mutex

// The method set of the base type of PtrMutex remains unchanged,
// but the method set of PtrMutex is empty.
type PtrMutex *Mutex

// The method set of *PrintableMutex contains the methods
```

```
// Lock and Unlock bound to its anonymous field Mutex.
type PrintableMutex struct {
    Mutex
}

// MyBlock is an interface type that has the same method set as Block.
type MyBlock Block
```

A type declaration may be used to define a different boolean, numeric, or string type and attach methods to it:

```
type TimeZone int

const (
    EST TimeZone = -(5 + iota)
    CST
    MST
    PST
)

func (tz TimeZone) String() string {
    return fmt.Sprintf("GMT+%dh", tz)
}
```

Variable declarations

A variable declaration creates a variable, binds an identifier to it and gives it a type and optionally an initial value.

```
VarDecl     = "var" ( VarSpec | "(" { VarSpec ";" } ")" ) .
VarSpec     = IdentifierList ( Type [ "=" ExpressionList ] | "=" ExpressionList ) .
```

```
var i int
var U, V, W float64
var k = 0
var x, y float32 = -1, -2
var (
    i       int
    u, v, s = 2.0, 3.0, "bar"
)
var re, im = complexSqrt(-1)
var _, found = entries[name]  // map lookup; only interested in "found"
```

If a list of expressions is given, the variables are initialized by assigning the expressions to the variables in order; all expressions must be consumed and all variables initialized from them. Otherwise, each variable is initialized to its zero value.

If the type is present, each variable is given that type. Otherwise, the types are deduced from the assignment of the expression list.

If the type is absent and the corresponding expression evaluates to an untyped constant, the type of the declared variable is as described in Assignments.

Implementation restriction: A compiler may make it illegal to declare a variable inside a function body if the variable is never used.

Short variable declarations

A *short variable declaration* uses the syntax:

```
ShortVarDecl = IdentifierList ":=" ExpressionList .
```

It is a shorthand for a regular variable declaration with initializer expressions but no types:

```
"var" IdentifierList = ExpressionList .
```

```
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w := os.Pipe(fd)  // os.Pipe() returns two values
_, y, _ := coord(p)  // coord() returns three values; only interested in y coordinate
```

Unlike regular variable declarations, a short variable declaration may redeclare variables provided they were originally declared in the same block with the same type, and at least one of the non-blank variables is new. As a consequence, redeclaration can only appear in a multi-variable short declaration. Redeclaration does not introduce a new variable; it just assigns a new value to the original.

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // redeclares offset
```

Short variable declarations may appear only inside functions. In some contexts such as the initializers for `if`, `for`, or `switch` statements, they can be used to declare local temporary variables.

Function declarations

A function declaration binds an identifier, the *function name*, to a function.

```
FunctionDecl = "func" FunctionName Signature [ Body ] .
FunctionName = identifier .
Body         = Block .
```

A function declaration may omit the body. Such a declaration provides the signature for a function implemented outside Go, such as an assembly routine.

```
func min(x int, y int) int {
    if x < y {
        return x
    }
    return y
}

func flushICache(begin, end uintptr)  // implemented externally
```

Method declarations

A method is a function with a *receiver*. A method declaration binds an identifier, the *method name*, to a method. It also associates the method with the receiver's *base type*.

```
MethodDecl   = "func" Receiver MethodName Signature [ Body ] .
Receiver     = "(" [ identifier ] [ "*" ] BaseTypeName ")" .
BaseTypeName = identifier .
```

The receiver type must be of the form `T` or `*T` where `T` is a type name. The type denoted by `T` is called the receiver *base type*; it must not be a pointer or interface type and it must be declared in the same package as the method. The method is said to be *bound* to the base type and the method name is visible only within selectors for that type.

For a base type, the non-blank names of methods bound to it must be unique. If the base type is a struct type, the non-blank method and field names must be distinct.

Given type `Point`, the declarations

```
func (p *Point) Length() float64 {
    return math.Sqrt(p.x * p.x + p.y * p.y)
}

func (p *Point) Scale(factor float64) {
    p.x *= factor
    p.y *= factor
}
```

bind the methods `Length` and `Scale`, with receiver type `*Point`, to the base type `Point`.

If the receiver's value is not referenced inside the body of the method, its identifier may be omitted in the declaration. The same applies in general to parameters of functions and methods.

The type of a method is the type of a function with the receiver as first argument. For instance, the method `Scale` has type

```
func(p *Point, factor float64)
```

However, a function declared this way is not a method.

Expressions

An expression specifies the computation of a value by applying operators and functions to operands.

Operands

Operands denote the elementary values in an expression. An operand may be a literal, a (possibly qualified) identifier denoting a constant, variable, or function, a method expression yielding a function, or a parenthesized expression.

```
Operand    = Literal | OperandName | MethodExpr | "(" Expression ")" .
Literal    = BasicLit | CompositeLit | FunctionLit .
BasicLit   = int_lit | float_lit | imaginary_lit | char_lit | string_lit .
OperandName = identifier | QualifiedIdent.
```

Qualified identifiers

A qualified identifier is an identifier qualified with a package name prefix. Both the package name and the identifier must not be blank.

```
QualifiedIdent = PackageName "." identifier .
```

A qualified identifier accesses an identifier in a different package, which must be imported. The identifier must be exported and declared in the package block of that package.

```
math.Sin    // denotes the Sin function in package math
```

Composite literals

Composite literals construct values for structs, arrays, slices, and maps and create a new value each time they are evaluated. They consist of the type of the value followed by a brace-bound list of composite elements. An element may be a single expression or a key-value pair.

```
CompositeLit  = LiteralType LiteralValue .
LiteralType   = StructType | ArrayType | "[" "..." "]" ElementType |
                SliceType | MapType | TypeName .
```

```
LiteralValue  = "{" [ ElementList [ "," ] ] "}" .
ElementList   = Element { "," Element } .
Element       = [ Key ":" ] Value .
Key           = FieldName | ElementIndex .
FieldName     = identifier .
ElementIndex  = Expression .
Value         = Expression | LiteralValue .
```

The LiteralType must be a struct, array, slice, or map type (the grammar enforces this constraint except when the type is given as a TypeName). The types of the expressions must be assignable to the respective field, element, and key types of the LiteralType; there is no additional conversion. The key is interpreted as a field name for struct literals, an index expression for array and slice literals, and a key for map literals. For map literals, all elements must have a key. It is an error to specify multiple elements with the same field name or constant key value.

For struct literals the following rules apply:

- A key must be a field name declared in the LiteralType.
- A literal that does not contain any keys must list an element for each struct field in the order in which the fields are declared.
- If any element has a key, every element must have a key.
- A literal that contains keys does not need to have an element for each struct field. Omitted fields get the zero value for that field.
- A literal may omit the element list; such a literal evaluates to the zero value for its type.
- It is an error to specify an element for a non-exported field of a struct belonging to a different package.

Given the declarations

```
type Point3D struct { x, y, z float64 }
type Line struct { p, q Point3D }
```

one may write

```
origin := Point3D{}                            // zero value for Point3D
line := Line{origin, Point3D{y: -4, z: 12.3}}  // zero value for line.q.x
```

For array and slice literals the following rules apply:

- Each element has an associated integer index marking its position in the array.
- An element with a key uses the key as its index; the key must be a constant integer expression.
- An element without a key uses the previous element's index plus one. If the first element has no key, its index is zero.

Taking the address of a composite literal generates a pointer to a unique instance of the literal's value.

```
var pointer *Point3D = &Point3D{y: 1000}
```

The length of an array literal is the length specified in the LiteralType. If fewer elements than the length are provided in the literal, the missing elements are set to the zero value for the array element type. It is an error to provide elements with index values outside the index range of the array. The notation `...` specifies an array length equal to the maximum element index plus one.

```
buffer := [10]string{}             // len(buffer) == 10
intSet := [6]int{1, 2, 3, 5}       // len(intSet) == 6
days := [...]string{"Sat", "Sun"}  // len(days) == 2
```

A slice literal describes the entire underlying array literal. Thus, the length and capacity of a slice literal are the maximum element index plus one. A slice literal has the form

```
[]T{x1, x2, … xn}
```

and is a shortcut for a slice operation applied to an array:

```
tmp := [n]T{x1, x2, … xn}
tmp[0 : n]
```

Within a composite literal of array, slice, or map type `T`, elements that are themselves composite literals may elide the respective literal type if it is identical to the element type of `T`. Similarly, elements that are addresses of composite literals may elide the `&T` when the element type is `*T`.

```
[...]Point{{1.5, -3.5}, {0, 0}}   // same as [...]Point{Point{1.5, -3.5}, Point{0, 0}}
[][]int{{1, 2, 3}, {4, 5}}        // same as [][]int{[]int{1, 2, 3}, []int{4, 5}}

[...]*Point{{1.5, -3.5}, {0, 0}}  // same as [...]*Point{&Point{1.5, -3.5}, &Point{0, 0}}
```

A parsing ambiguity arises when a composite literal using the TypeName form of the LiteralType appears between the keyword and the opening brace of the block of an "if", "for", or "switch" statement, because the braces surrounding the expressions in the literal are confused with those introducing the block of statements. To resolve the ambiguity in this rare case, the composite literal must appear within parentheses.

```
if x == (T{a,b,c}[i]) { … }
if (x == T{a,b,c}[i]) { … }
```

Examples of valid array, slice, and map literals:

```
// list of prime numbers
primes := []int{2, 3, 5, 7, 9, 2147483647}

// vowels[ch] is true if ch is a vowel
vowels := [128]bool{'a': true, 'e': true, 'i': true, 'o': true, 'u': true, 'y': true}

// the array [10]float32{-1, 0, 0, 0, -0.1, -0.1, 0, 0, 0, -1}
filter := [10]float32{-1, 4: -0.1, -0.1, 9: -1}

// frequencies in Hz for equal-tempered scale (A4 = 440Hz)
noteFrequency := map[string]float32{
    "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
    "G0": 24.50, "A0": 27.50, "B0": 30.87,
}
```

Function literals

A function literal represents an anonymous function. It consists of a specification of the function type and a function body.

```
FunctionLit = FunctionType Body .
```

```
func(a, b int, z float64) bool { return a*b < int(z) }
```

A function literal can be assigned to a variable or invoked directly.

```
f := func(x, y int) int { return x + y }
func(ch chan int) { ch <- ACK }(replyChan)
```

Function literals are *closures*: they may refer to variables defined in a surrounding function. Those variables are then shared between the surrounding function and the function literal, and they survive as long as they are accessible.

Primary expressions

Primary expressions are the operands for unary and binary expressions.

```
PrimaryExpr =
    Operand |
    Conversion |
    BuiltinCall |
    PrimaryExpr Selector |
    PrimaryExpr Index |
    PrimaryExpr Slice |
    PrimaryExpr TypeAssertion |
    PrimaryExpr Call .

Selector       = "." identifier .
Index          = "[" Expression "]" .
Slice          = "[" [ Expression ] ":" [ Expression ] "]" .
TypeAssertion  = "." "(" Type ")" .
Call           = "(" [ ArgumentList [ "," ] ] ")" .
ArgumentList   = ExpressionList [ "..." ] .
```

```
x
2
(s + ".txt")
f(3.1415, true)
Point{1, 2}
m["foo"]
s[i : j + 1]
obj.color
f.p[i].x()
```

Selectors

For a primary expression `x` that is not a package name, the *selector expression*

```
x.f
```

denotes the field or method `f` of the value `x` (or sometimes `*x`; see below). The identifier `f` is called the (field or method) *selector*; it must not be the blank identifier. The type of the selector expression is the type of `f`. If `x` is a package name, see the section on qualified identifiers.

A selector `f` may denote a field or method `f` of a type `T`, or it may refer to a field or method `f` of a nested anonymous field of `T`. The number of anonymous fields traversed to reach `f` is called its *depth* in `T`. The depth of a field or method `f` declared in `T` is zero. The depth of a field or method `f` declared in an anonymous field `A` in `T` is the depth of `f` in `A` plus one.

The following rules apply to selectors:

1. For a value `x` of type `T` or `*T` where `T` is not an interface type, `x.f` denotes the field or method at the shallowest depth in `T` where there is such an `f`. If there is not exactly one `f` with shallowest depth, the selector expression is illegal.

2. For a variable `x` of type `I` where `I` is an interface type, `x.f` denotes the actual method with name `f` of the value assigned to `x`. If there is no method with name `f` in the method set of `I`, the selector expression is illegal.

3. In all other cases, `x.f` is illegal.

4. If `x` is of pointer or interface type and has the value `nil`, assigning to, evaluating, or calling `x.f` causes a run-time panic.

Selectors automatically dereference pointers to structs. If `x` is a pointer to a struct, `x.y` is shorthand for `(*x).y`; if the field `y` is also a pointer to a struct, `x.y.z` is shorthand for `(*(*x).y).z`, and so on. If `x` contains an anonymous field of type `*A`, where `A` is also a struct type, `x.f` is a shortcut for `(*x.A).f`.

For example, given the declarations:

```
type T0 struct {
    x int
}

func (recv *T0) M0()

type T1 struct {
    y int
}

func (recv T1) M1()

type T2 struct {
    z int
    T1
    *T0
}

func (recv *T2) M2()

var p *T2  // with p != nil and p.T0 != nil
```

one may write:

```
p.z   // (*p).z
p.y   // ((*p).T1).y
p.x   // (*(*p).T0).x

p.M2  // (*p).M2
p.M1  // ((*p).T1).M1
p.M0  // ((*p).T0).M0
```

Indexes

A primary expression of the form

```
a[x]
```

denotes the element of the array, slice, string or map `a` indexed by `x`. The value `x` is called the *index* or *map key*, respectively. The following rules apply:

For `a` of type `A` or `*A` where `A` is an array type, or for `a` of type `S` where `S` is a slice type:

- `x` must be an integer value and `0 <= x < len(a)`
- `a[x]` is the array element at index `x` and the type of `a[x]` is the element type of `A`
- if `a` is `nil` or if the index `x` is out of range, a run-time panic occurs

For `a` of type `T` where `T` is a string type:

- `x` must be an integer value and `0 <= x < len(a)`
- `a[x]` is the byte at index `x` and the type of `a[x]` is `byte`

- `a[x]` may not be assigned to
- if the index `x` is out of range, a run-time panic occurs

For `a` of type `M` where `M` is a map type:

- `x`'s type must be assignable to the key type of `M`
- if the map contains an entry with key `x`, `a[x]` is the map value with key `x` and the type of `a[x]` is the value type of `M`
- if the map is `nil` or does not contain such an entry, `a[x]` is the zero value for the value type of `M`

Otherwise `a[x]` is illegal.

An index expression on a map `a` of type `map[K]V` may be used in an assignment or initialization of the special form

```
v, ok = a[x]
v, ok := a[x]
var v, ok = a[x]
```

where the result of the index expression is a pair of values with types `(V, bool)`. In this form, the value of `ok` is `true` if the key `x` is present in the map, and `false` otherwise. The value of `v` is the value `a[x]` as in the single-result form.

Assigning to an element of a `nil` map causes a run-time panic.

Slices

For a string, array, pointer to array, or slice `a`, the primary expression

```
a[low : high]
```

constructs a substring or slice. The index expressions `low` and `high` select which elements appear in the result. The result has indexes starting at 0 and length equal to `high - low`. After slicing the array `a`

```
a := [5]int{1, 2, 3, 4, 5}
s := a[1:4]
```

the slice `s` has type `[]int`, length 3, capacity 4, and elements

```
s[0] == 2
s[1] == 3
s[2] == 4
```

For convenience, any of the index expressions may be omitted. A missing `low` index defaults to zero; a missing `high` index defaults to the length of the sliced operand:

```
a[2:]  // same a[2 : len(a)]
a[:3]  // same as a[0 : 3]
a[:]   // same as a[0 : len(a)]
```

For arrays or strings, the indexes `low` and `high` must satisfy 0 <= `low` <= `high` <= length; for slices, the upper bound is the capacity rather than the length.

If the sliced operand is a string or slice, the result of the slice operation is a string or slice of the same type. If the sliced operand is an array, it must be addressable and the result of the slice operation is a slice with the same element type as the array.

Type assertions

For an expression `x` of interface type and a type `T`, the primary expression

```
x.(T)
```

asserts that `x` is not `nil` and that the value stored in `x` is of type `T`. The notation `x.(T)` is called a *type assertion*.

More precisely, if `T` is not an interface type, `x.(T)` asserts that the dynamic type of `x` is identical to the type `T`. If `T` is an interface type, `x.(T)` asserts that the dynamic type of `x` implements the interface `T`.

If the type assertion holds, the value of the expression is the value stored in `x` and its type is `T`. If the type assertion is false, a run-time panic occurs. In other words, even though the dynamic type of `x` is known only at run-time, the type of `x.(T)` is known to be `T` in a correct program.

If a type assertion is used in an assignment or initialization of the form

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
```

the result of the assertion is a pair of values with types `(T, bool)`. If the assertion holds, the expression returns the pair `(x.(T), true)`; otherwise, the expression returns `(Z, false)` where `Z` is the zero value for type `T`. No run-time panic occurs in this case. The type assertion in this construct thus acts like a function call returning a value and a boolean indicating success.

Calls

Given an expression `f` of function type `F`,

```
f(a1, a2, … an)
```

calls `f` with arguments `a1, a2, … an`. Except for one special case, arguments must be single-valued expressions assignable to the parameter types of `F` and are evaluated before the function is called. The type of the expression is the result type of `F`. A method invocation is similar but the method itself is specified as a selector upon a value of the receiver type for the method.

```
math.Atan2(x, y)  // function call
var pt *Point
pt.Scale(3.5)  // method call with receiver pt
```

In a function call, the function value and arguments are evaluated in the usual order. After they are evaluated, the parameters of the call are passed by value to the function and the called function begins execution. The return parameters of the function are passed by value back to the calling function when the function returns.

Calling a `nil` function value causes a run-time panic.

As a special case, if the return parameters of a function or method `g` are equal in number and individually assignable to the parameters of another function or method `f`, then the call `f(g(parameters_of_g))` will invoke `f` after binding the return values of `g` to the parameters of `f` in order. The call of `f` must contain no parameters other than the call of `g`. If `f` has a final `...` parameter, it is assigned the return values of `g` that remain after assignment of regular parameters.

```
func Split(s string, pos int) (string, string) {
    return s[0:pos], s[pos:]
}

func Join(s, t string) string {
    return s + t
}

if Join(Split(value, len(value)/2)) != value {
```

```
    log.Panic("test fails")
}
```

A method call `x.m()` is valid if the method set of (the type of) `x` contains `m` and the argument list can be assigned to the parameter list of `m`. If `x` is addressable and `&x`'s method set contains `m`, `x.m()` is shorthand for `(&x).m()`:

```
var p Point
p.Scale(3.5)
```

There is no distinct method type and there are no method literals.

Passing arguments to ... parameters

If `f` is variadic with final parameter type `...T`, then within the function the argument is equivalent to a parameter of type `[]T`. At each call of `f`, the argument passed to the final parameter is a new slice of type `[]T` whose successive elements are the actual arguments, which all must be assignable to the type `T`. The length of the slice is therefore the number of arguments bound to the final parameter and may differ for each call site.

Given the function and call

```
func Greeting(prefix string, who ...string)
Greeting("hello:", "Joe", "Anna", "Eileen")
```

within `Greeting`, `who` will have the value `[]string{"Joe", "Anna", "Eileen"}`

If the final argument is assignable to a slice type `[]T`, it may be passed unchanged as the value for a `...T` parameter if the argument is followed by `...`. In this case no new slice is created.

Given the slice `s` and call

```
s := []string{"James", "Jasmine"}
Greeting("goodbye:", s...)
```

within `Greeting`, `who` will have the same value as `s` with the same underlying array.

Operators

Operators combine operands into expressions.

```
Expression = UnaryExpr | Expression binary_op UnaryExpr .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .

binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

Comparisons are discussed elsewhere. For other binary operators, the operand types must be identical unless the operation involves shifts or untyped constants. For operations involving constants only, see the section on constant expressions.

Except for shift operations, if one operand is an untyped constant and the other operand is not, the constant is converted to the type of the other operand.

The right operand in a shift expression must have unsigned integer type or be an untyped constant that can be converted to unsigned integer type. If the left operand of a non-constant shift expression is an untyped constant, the type of the constant is what it would be if the shift expression were replaced by its left operand alone; the type is `int` if it cannot be determined from the context (for instance, if the shift expression is an operand in a comparison against an untyped constant).

```
var s uint = 33
var i = 1<<s           // 1 has type int
var j int32 = 1<<s     // 1 has type int32; j == 0
var k = uint64(1<<s)   // 1 has type uint64; k == 1<<33
var m int = 1.0<<s     // 1.0 has type int
var n = 1.0<<s != 0    // 1.0 has type int; n == false if ints are 32bits in size
var o = 1<<s == 2<<s   // 1 and 2 have type int; o == true if ints are 32bits in size
var p = 1<<s == 1<<33  // illegal if ints are 32bits in size: 1 has type int, but 1<<33 overflows int
var u = 1.0<<s         // illegal: 1.0 has type float64, cannot shift
var v float32 = 1<<s   // illegal: 1 has type float32, cannot shift
var w int64 = 1.0<<33  // 1.0<<33 is a constant shift expression
```

Operator precedence

Unary operators have the highest precedence. As the `++` and `--` operators form statements, not expressions, they fall outside the operator hierarchy. As a consequence, statement `*p++` is the same as `(*p)++`.

There are five precedence levels for binary operators. Multiplication operators bind strongest, followed by addition operators, comparison operators, `&&` (logical and), and finally `||` (logical or):

```
Precedence    Operator
    5             *  /  %  <<  >>  &  &^
    4             +  -  |  ^
    3             ==  !=  <  <=  >  >=
    2             &&
    1             ||
```

Binary operators of the same precedence associate from left to right. For instance, `x / y * z` is the same as `(x / y) * z`.

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y+1 && <-chanPtr > 0
```

Arithmetic operators

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The four standard arithmetic operators (`+`, `-`, `*`, `/`) apply to integer, floating-point, and complex types; `+` also applies to strings. All other arithmetic operators apply to integers only.

```
+    sum                    integers, floats, complex values, strings
-    difference             integers, floats, complex values
*    product                integers, floats, complex values
/    quotient               integers, floats, complex values
%    remainder              integers

&    bitwise and            integers
|    bitwise or             integers
^    bitwise xor            integers
&^   bit clear (and not)    integers

<<   left shift             integer << unsigned integer
>>   right shift            integer >> unsigned integer
```

Strings can be concatenated using the + operator or the += assignment operator:

```
s := "hi" + string(c)
s += " and good bye"
```

String addition creates a new string by concatenating the operands.

For two integer values x and y, the integer quotient q = x / y and remainder r = x % y satisfy the following relationships:

```
x = q*y + r  and  |r| < |y|
```

with x / y truncated towards zero ("truncated division").

```
 x     y     x / y     x % y
 5     3       1         2
-5     3      -1        -2
 5    -3      -1         2
-5    -3       1        -2
```

As an exception to this rule, if the dividend x is the most negative value for the int type of x, the quotient q = x / -1 is equal to x (and r = 0).

```
             x, q
int8                      -128
int16                   -32768
int32              -2147483648
int64    -9223372036854775808
```

If the divisor is zero, a run-time panic occurs. If the dividend is positive and the divisor is a constant power of 2, the division may be replaced by a right shift, and computing the remainder may be replaced by a bitwise "and" operation:

```
 x     x / 4     x % 4     x >> 2     x & 3
 11      2         3         2          3
-11     -2        -3        -3          1
```

The shift operators shift the left operand by the shift count specified by the right operand. They implement arithmetic shifts if the left operand is a signed integer and logical shifts if it is an unsigned integer. There is no upper limit on the shift count. Shifts behave as if the left operand is shifted n times by 1 for a shift count of n. As a result, x << 1 is the same as x*2 and x >> 1 is the same as x/2 but truncated towards negative infinity.

For integer operands, the unary operators +, -, and ^ are defined as follows:

```
+x                          is 0 + x
-x    negation              is 0 - x
^x    bitwise complement    is m ^ x  with m = "all bits set to 1" for unsigned x
                                      and  m = -1 for signed x
```

For floating-point and complex numbers, +x is the same as x, while -x is the negation of x. The result of a floating-point or complex division by zero is not specified beyond the IEEE-754 standard; whether a run-time panic occurs is implementation-specific.

Integer overflow

For unsigned integer values, the operations +, -, *, and << are computed modulo 2^n, where n is the bit width of the unsigned integer's type. Loosely speaking, these unsigned integer operations discard high bits upon overflow, and programs may rely on "wrap around".

For signed integers, the operations +, -, *, and << may legally overflow and the resulting value exists and is deterministically defined by the signed integer representation, the operation, and its operands. No exception is raised as a result of overflow. A compiler may not optimize code under the assumption that overflow does not occur. For instance, it may not assume that x < x + 1 is always true.

Comparison operators

Comparison operators compare two operands and yield a boolean value.

```
==    equal
!=    not equal
<     less
<=    less or equal
>     greater
>=    greater or equal
```

In any comparison, the first operand must be assignable to the type of the second operand, or vice versa.

The equality operators `==` and `!=` apply to operands that are *comparable*. The ordering operators `<`, `<=`, `>`, and `>=` apply to operands that are *ordered*. These terms and the result of the comparisons are defined as follows:

- Boolean values are comparable. Two boolean values are equal if they are either both `true` or both `false`.
- Integer values are comparable and ordered, in the usual way.
- Floating point values are comparable and ordered, as defined by the IEEE-754 standard.
- Complex values are comparable. Two complex values `u` and `v` are equal if both `real(u) == real(v)` and `imag(u) == imag(v)`.
- String values are comparable and ordered, lexically byte-wise.
- Pointer values are comparable. Two pointer values are equal if they point to the same variable or if both have value `nil`. Pointers to distinct zero-size variables may or may not be equal.
- Channel values are comparable. Two channel values are equal if they were created by the same call to `make` or if both have value `nil`.
- Interface values are comparable. Two interface values are equal if they have identical dynamic types and equal dynamic values or if both have value `nil`.
- A value `x` of non-interface type `X` and a value `t` of interface type `T` are comparable when values of type `X` are comparable and `X` implements `T`. They are equal if `t`'s dynamic type is identical to `X` and `t`'s dynamic value is equal to `x`.
- Struct values are comparable if all their fields are comparable. Two struct values are equal if their corresponding non-blank fields are equal.
- Array values are comparable if values of the array element type are comparable. Two array values are equal if their corresponding elements are equal.

A comparison of two interface values with identical dynamic types causes a run-time panic if values of that type are not comparable. This behavior applies not only to direct interface value comparisons but also when comparing arrays of interface values or structs with interface-valued fields.

Slice, map, and function values are not comparable. However, as a special case, a slice, map, or function value may be compared to the predeclared identifier `nil`. Comparison of pointer, channel, and interface values to `nil` is also allowed and follows from the general rules above.

The result of a comparison can be assigned to any boolean type. If the context does not demand a specific boolean type, the result has type `bool`.

```
type MyBool bool

var x, y int
var (
    b1 MyBool = x == y // result of comparison has type MyBool
    b2 bool   = x == y // result of comparison has type bool
    b3        = x == y // result of comparison has type bool
)
```

Logical operators

Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally.

```
&&    conditional and    p && q  is  "if p then q else false"
||    conditional or     p || q  is  "if p then true else q"
!     not                !p      is  "not p"
```

Address operators

For an operand `x` of type `T`, the address operation `&x` generates a pointer of type `*T` to `x`. The operand must be *addressable*, that is, either a variable, pointer indirection, or slice indexing operation; or a field selector of an addressable struct operand; or an array indexing operation of an addressable array. As an exception to the addressability requirement, `x` may also be a composite literal.

For an operand `x` of pointer type `*T`, the pointer indirection `*x` denotes the value of type `T` pointed to by `x`. If `x` is `nil`, an attempt to evaluate `*x` will cause a run-time panic.

```
&x
&a[f(2)]
*p
*pf(x)
```

Receive operator

For an operand `ch` of channel type, the value of the receive operation `<-ch` is the value received from the channel `ch`. The type of the value is the element type of the channel. The expression blocks until a value is available. Receiving from a `nil` channel blocks forever. Receiving from a closed channel always succeeds, immediately returning the element type's zero value.

```
v1 := <-ch
v2 = <-ch
f(<-ch)
<-strobe  // wait until clock pulse and discard received value
```

A receive expression used in an assignment or initialization of the form

```
x, ok = <-ch
x, ok := <-ch
var x, ok = <-ch
```

yields an additional result of type `bool` reporting whether the communication succeeded. The value of `ok` is `true` if the value received was delivered by a successful send operation to the channel, or `false` if it is a zero value generated because the channel is closed and empty.

Method expressions

If `M` is in the method set of type `T`, `T.M` is a function that is callable as a regular function with the same arguments as `M` prefixed by an additional argument that is the receiver of the method.

```
MethodExpr    = ReceiverType "." MethodName .
ReceiverType  = TypeName | "(" "*" TypeName ")" .
```

Consider a struct type `T` with two methods, `Mv`, whose receiver is of type `T`, and `Mp`, whose receiver is of type `*T`.

```
type T struct {
    a int
}
func (tv  T) Mv(a int) int         { return 0 }  // value receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer receiver
var t T
```

The expression

```
T.Mv
```

yields a function equivalent to `Mv` but with an explicit receiver as its first argument; it has signature

```
func(tv T, a int) int
```

That function may be called normally with an explicit receiver, so these three invocations are equivalent:

```
t.Mv(7)
T.Mv(t, 7)
f := T.Mv; f(t, 7)
```

Similarly, the expression

```
(*T).Mp
```

yields a function value representing `Mp` with signature

```
func(tp *T, f float32) float32
```

For a method with a value receiver, one can derive a function with an explicit pointer receiver, so

```
(*T).Mv
```

yields a function value representing `Mv` with signature

```
func(tv *T, a int) int
```

Such a function indirects through the receiver to create a value to pass as the receiver to the underlying method; the method does not overwrite the value whose address is passed in the function call.

The final case, a value-receiver function for a pointer-receiver method, is illegal because pointer-receiver methods are not in the method set of the value type.

Function values derived from methods are called with function call syntax; the receiver is provided as the first argument to the call. That is, given `f := T.Mv`, `f` is invoked as `f(t, 7)` not `t.f(7)`. To construct a function that binds the receiver, use a closure.

It is legal to derive a function value from a method of an interface type. The resulting function takes an explicit receiver of that interface type.

Conversions

Conversions are expressions of the form `T(x)` where `T` is a type and `x` is an expression that can be converted to type `T`.

```
Conversion = Type "(" Expression ")" .
```

If the type starts with an operator it must be parenthesized:

```
*Point(p)        // same as *(Point(p))
(*Point)(p)      // p is converted to (*Point)
<-chan int(c)    // same as <-(chan int(c))
(<-chan int)(c)  // c is converted to (<-chan int)
```

A constant value `x` can be converted to type `T` in any of these cases:

- `x` is representable by a value of type `T`.
- `x` is an integer constant and `T` is a string type. The same rule as for non-constant `x` applies in this case.

Converting a constant yields a typed constant as result.

```
uint(iota)               // iota value of type uint
float32(2.718281828)     // 2.718281828 of type float32
complex128(1)            // 1.0 + 0.0i of type complex128
string('x')              // "x" of type string
string(0x266c)           // "♬" of type string
MyString("foo" + "bar")  // "foobar" of type MyString
string([]byte{'a'})      // not a constant: []byte{'a'} is not a constant
(*int)(nil)              // not a constant: nil is not a constant, *int is not a boolean, numeric, or stri
int(1.2)                 // illegal: 1.2 cannot be represented as an int
string(65.0)             // illegal: 65.0 is not an integer constant
```

A non-constant value `x` can be converted to type `T` in any of these cases:

- `x` is assignable to `T`.
- `x`'s type and `T` have identical underlying types.
- `x`'s type and `T` are unnamed pointer types and their pointer base types have identical underlying types.
- `x`'s type and `T` are both integer or floating point types.
- `x`'s type and `T` are both complex types.
- `x` is an integer or has type `[]byte` or `[]rune` and `T` is a string type.
- `x` is a string and `T` is `[]byte` or `[]rune`.

Specific rules apply to (non-constant) conversions between numeric types or to and from a string type. These conversions may change the representation of `x` and incur a run-time cost. All other conversions only change the type but not the representation of `x`.

There is no linguistic mechanism to convert between pointers and integers. The package `unsafe` implements this functionality under restricted circumstances.

Conversions between numeric types For the conversion of non-constant numeric values, the following rules apply:

1. When converting between integer types, if the value is a signed integer, it is sign extended to implicit infinite precision; otherwise it is zero extended. It is then truncated to fit in the result type's size. For example, if `v := uint16(0x10F0)`, then `uint32(int8(v)) == 0xFFFFFFF0`. The conversion always yields a valid value; there is no indication of overflow.
2. When converting a floating-point number to an integer, the fraction is discarded (truncation towards zero).
3. When converting an integer or floating-point number to a floating-point type, or a complex number to another complex type, the result value is rounded to the precision specified by the destination type. For instance, the value of a variable `x` of type `float32` may be stored using additional precision beyond that of an IEEE-754 32-bit number, but float32(x) represents the result of rounding `x`'s value to 32-bit precision. Similarly, `x + 0.1` may use more than 32 bits of precision, but `float32(x + 0.1)` does not.

In all non-constant conversions involving floating-point or complex values, if the result type cannot represent the value the conversion succeeds but the result value is implementation-dependent.

Conversions to and from a string type

1. Converting a signed or unsigned integer value to a string type yields a string containing the UTF-8 representation of the integer. Values outside the range of valid Unicode code points are converted to `"\uFFFD"`.

```
string('a')       // "a"
string(-1)        // "\ufffd" == "\xef\xbf\xbd"
string(0xf8)      // "\u00f8" == "ø" == "\xc3\xb8"
type MyString string
MyString(0x65e5)  // "\u65e5" == " " == "\xe6\x97\xa5"
```

2. Converting a slice of bytes to a string type yields a string whose successive bytes are the elements of the slice. If the slice value is `nil`, the result is the empty string.

```
string([]byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'})  // "hellø"

type MyBytes []byte
string(MyBytes{'h', 'e', 'l', 'l', '\xc3', '\xb8'})  // "hellø"
```

3. Converting a slice of runes to a string type yields a string that is the concatenation of the individual rune values converted to strings. If the slice value is `nil`, the result is the empty string.

```
string([]rune{0x767d, 0x9d6c, 0x7fd4})  // "\u767d\u9d6c\u7fd4" == "  "

type MyRunes []rune
string(MyRunes{0x767d, 0x9d6c, 0x7fd4})  // "\u767d\u9d6c\u7fd4" == "  "
```

4. Converting a value of a string type to a slice of bytes type yields a slice whose successive elements are the bytes of the string. If the string is empty, the result is `[]byte(nil)`.

```
[]byte("hellø")   // []byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'}
MyBytes("hellø")  // []byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'}
```

5. Converting a value of a string type to a slice of runes type yields a slice containing the individual Unicode code points of the string. If the string is empty, the result is `[]rune(nil)`.

```
[]rune(MyString("  "))  // []rune{0x767d, 0x9d6c, 0x7fd4}
MyRunes("  ")           // []rune{0x767d, 0x9d6c, 0x7fd4}
```

Constant expressions

Constant expressions may contain only constant operands and are evaluated at compile-time.

Untyped boolean, numeric, and string constants may be used as operands wherever it is legal to use an operand of boolean, numeric, or string type, respectively. Except for shift operations, if the operands of a binary operation are different kinds of untyped constants, the operation and, for non-boolean operations, the result use the kind that appears later in this list: integer, rune, floating-point, complex. For example, an untyped integer constant divided by an untyped complex constant yields an untyped complex constant.

A constant comparison always yields an untyped boolean constant. If the left operand of a constant shift expression is an untyped constant, the result is an integer constant; otherwise it is a constant of the same type as the left operand, which must be of integer type Applying all other operators to untyped constants results in an untyped constant of the same kind (that is, a boolean, integer, floating-point, complex, or string constant).

```
const a = 2 + 3.0          // a == 5.0   (untyped floating-point constant)
const b = 15 / 4           // b == 3     (untyped integer constant)
const c = 15 / 4.0         // c == 3.75  (untyped floating-point constant)
const θ float64 = 3/2      // θ == 1.5   (type float64)
```

```
const d = 1 << 3.0         // d == 8     (untyped integer constant)
const e = 1.0 << 3         // e == 8     (untyped integer constant)
const f = int32(1) << 33   // f == 0     (type int32)
const g = float64(2) >> 1  // illegal    (float64(2) is a typed floating-point constant)
const h = "foo" > "bar"    // h == true  (untyped boolean constant)
const j = true             // j == true  (untyped boolean constant)
const k = 'w' + 1          // k == 'x'   (untyped rune constant)
const l = "hi"             // l == "hi"  (untyped string constant)
const m = string(k)        // m == "x"   (type string)
const Σ = 1 - 0.707i       //            (untyped complex constant)
const Δ = Σ + 2.0e-4       //            (untyped complex constant)
const Φ = iota*1i - 1/1i   //            (untyped complex constant)
```

Applying the built-in function `complex` to untyped integer, rune, or floating-point constants yields an untyped complex constant.

```
const ic = complex(0, c)   // ic == 3.75i (untyped complex constant)
const iΘ = complex(0, Θ)   // iΘ == 1.5i  (type complex128)
```

Constant expressions are always evaluated exactly; intermediate values and the constants themselves may require precision significantly larger than supported by any predeclared type in the language. The following are legal declarations:

```
const Huge = 1 << 100
const Four int8 = Huge >> 98
```

The values of *typed* constants must always be accurately representable as values of the constant type. The following constant expressions are illegal:

```
uint(-1)     // -1 cannot be represented as a uint
int(3.14)    // 3.14 cannot be represented as an int
int64(Huge)  // 1<<100 cannot be represented as an int64
Four * 300   // 300 cannot be represented as an int8
Four * 100   // 400 cannot be represented as an int8
```

The mask used by the unary bitwise complement operator ^ matches the rule for non-constants: the mask is all 1s for unsigned constants and -1 for signed and untyped constants.

```
^1         // untyped integer constant, equal to -2
uint8(^1)  // error, same as uint8(-2), out of range
^uint8(1)  // typed uint8 constant, same as 0xFF ^ uint8(1) = uint8(0xFE)
int8(^1)   // same as int8(-2)
^int8(1)   // same as -1 ^ int8(1) = -2
```

Implementation restriction: A compiler may use rounding while computing untyped floating-point or complex constant expressions; see the implementation restriction in the section on constants. This rounding may cause a floating-point constant expression to be invalid in an integer context, even if it would be integral when calculated using infinite precision.

Order of evaluation

When evaluating the operands of an expression, assignment, or return statement, all function calls, method calls, and communication operations are evaluated in lexical left-to-right order.

For example, in the assignment

```
y[f()], ok = g(h(), i()+x[j()], <-c), k()
```

the function calls and communication happen in the order `f()`, `h()`, `i()`, `j()`, `<-c`, `g()`, and `k()`. However, the order of those events compared to the evaluation and indexing of `x` and the evaluation of `y` is not specified.

```
a := 1
f := func() int { a = 2; return 3 }
x := []int{a, f()}  // x may be [1, 3] or [2, 3]: evaluation order between a and f() is not specified
```

Floating-point operations within a single expression are evaluated according to the associativity of the operators. Explicit parentheses affect the evaluation by overriding the default associativity. In the expression `x + (y + z)` the addition `y + z` is performed before adding `x`.

Statements

Statements control execution.

```
Statement =
    Declaration | LabeledStmt | SimpleStmt |
    GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |
    FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt | ForStmt |
    DeferStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt | Assignment | ShortVarDecl .
```

Empty statements

The empty statement does nothing.

```
EmptyStmt = .
```

Labeled statements

A labeled statement may be the target of a `goto`, `break` or `continue` statement.

```
LabeledStmt = Label ":" Statement .
Label       = identifier .
```

```
Error: log.Panic("error encountered")
```

Expression statements

Function calls, method calls, and receive operations can appear in statement context. Such statements may be parenthesized.

```
ExpressionStmt = Expression .
```

```
h(x+y)
f.Close()
<-ch
(<-ch)
```

Send statements

A send statement sends a value on a channel. The channel expression must be of channel type and the type of the value must be assignable to the channel's element type.

```
SendStmt = Channel "<-" Expression .
Channel  = Expression .
```

Both the channel and the value expression are evaluated before communication begins. Communication blocks until the send can proceed. A send on an unbuffered channel can proceed if a receiver is ready. A send on a buffered channel can proceed if there is room in the buffer. A send on a closed channel proceeds by causing a run-time panic. A send on a `nil` channel blocks forever.

```
ch <- 3
```

IncDec statements

The "++" and "–" statements increment or decrement their operands by the untyped constant `1`. As with an assignment, the operand must be addressable or a map index expression.

```
IncDecStmt = Expression ( "++" | "--" ) .
```

The following assignment statements are semantically equivalent:

```
IncDec statement    Assignment
x++                 x += 1
x--                 x -= 1
```

Assignments

```
Assignment = ExpressionList assign_op ExpressionList .

assign_op = [ add_op | mul_op ] "=" .
```

Each left-hand side operand must be addressable, a map index expression, or the blank identifier. Operands may be parenthesized.

```
x = 1
*p = f()
a[i] = 23
(k) = <-ch  // same as: k = <-ch
```

An *assignment operation* `x` *op*`=` `y` where *op* is a binary arithmetic operation is equivalent to `x = x` *op* `y` but evaluates `x` only once. The *op*`=` construct is a single token. In assignment operations, both the left- and right-hand expression lists must contain exactly one single-valued expression.

```
a[i] <<= 2
i &^= 1<<n
```

A tuple assignment assigns the individual elements of a multi-valued operation to a list of variables. There are two forms. In the first, the right hand operand is a single multi-valued expression such as a function evaluation or channel or map operation or a type assertion. The number of operands on the left hand side must match the number of values. For instance, if `f` is a function returning two values,

```
x, y = f()
```

assigns the first value to `x` and the second to `y`. The blank identifier provides a way to ignore values returned by a multi-valued expression:

```
x, _ = f()  // ignore second value returned by f()
```

In the second form, the number of operands on the left must equal the number of expressions on the right, each of which must be single-valued, and the *n*th expression on the right is assigned to the *n*th operand on the left.

The assignment proceeds in two phases. First, the operands of index expressions and pointer indirections (including implicit pointer indirections in selectors) on the left and the expressions on the right are all evaluated in the usual order. Second, the assignments are carried out in left-to-right order.

```
a, b = b, a  // exchange a and b

x := []int{1, 2, 3}
i := 0
i, x[i] = 1, 2  // set i = 1, x[0] = 2

i = 0
x[i], i = 2, 1  // set x[0] = 2, i = 1

x[0], x[0] = 1, 2  // set x[0] = 1, then x[0] = 2 (so x[0] == 2 at end)

x[1], x[3] = 4, 5  // set x[1] = 4, then panic setting x[3] = 5.

type Point struct { x, y int }
var p *Point
x[2], p.x = 6, 7  // set x[2] = 6, then panic setting p.x = 7

i = 2
x = []int{3, 5, 7}
for i, x[i] = range x {  // set i, x[2] = 0, x[0]
    break
}
// after this loop, i == 0 and x == []int{3, 5, 3}
```

In assignments, each value must be assignable to the type of the operand to which it is assigned. If an untyped constant is assigned to a variable of interface type, the constant is converted to type `bool`, `rune`, `int`, `float64`, `complex128` or `string` respectively, depending on whether the value is a boolean, rune, integer, floating-point, complex, or string constant.

If statements

"If" statements specify the conditional execution of two branches according to the value of a boolean expression. If the expression evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

```
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ] .
```

```
if x > max {
    x = max
}
```

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
if x := f(); x < y {
    return x
} else if x > z {
```

```
    return z
} else {
    return y
}
```

Switch statements

"Switch" statements provide multi-way execution. An expression or type specifier is compared to the "cases" inside the "switch" to determine which branch to execute.

```
SwitchStmt = ExprSwitchStmt | TypeSwitchStmt .
```

There are two forms: expression switches and type switches. In an expression switch, the cases contain expressions that are compared against the value of the switch expression. In a type switch, the cases contain types that are compared against the type of a specially annotated switch expression.

Expression switches In an expression switch, the switch expression is evaluated and the case expressions, which need not be constants, are evaluated left-to-right and top-to-bottom; the first one that equals the switch expression triggers execution of the statements of the associated case; the other cases are skipped. If no case matches and there is a "default" case, its statements are executed. There can be at most one default case and it may appear anywhere in the "switch" statement. A missing switch expression is equivalent to the expression `true`.

```
ExprSwitchStmt = "switch" [ SimpleStmt ";" ] [ Expression ] "{" { ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" { Statement ";" } .
ExprSwitchCase = "case" ExpressionList | "default" .
```

In a case or default clause, the last statement only may be a "fallthrough" statement to indicate that control should flow from the end of this clause to the first statement of the next clause. Otherwise control flows to the end of the "switch" statement.

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
switch tag {
default: s3()
case 0, 1, 2, 3: s1()
case 4, 5, 6, 7: s2()
}

switch x := f(); {  // missing switch expression means "true"
case x < 0: return -x
default: return x
}

switch {
case x < y: f1()
case x < z: f2()
case x == 4: f3()
}
```

Type switches A type switch compares types rather than values. It is otherwise similar to an expression switch. It is marked by a special switch expression that has the form of a type assertion using the reserved word `type` rather than an actual type. Cases then match literal types against the dynamic type of the expression in the type assertion.

```
TypeSwitchStmt  = "switch" [ SimpleStmt ";" ] TypeSwitchGuard "{" { TypeCaseClause } "}" .
TypeSwitchGuard = [ identifier ":=" ] PrimaryExpr "." "(" "type" ")" .
TypeCaseClause  = TypeSwitchCase ":" { Statement ";" } .
```

```
TypeSwitchCase  = "case" TypeList | "default" .
TypeList        = Type { "," Type } .
```

The TypeSwitchGuard may include a short variable declaration. When that form is used, the variable is declared at the beginning of the implicit block in each clause. In clauses with a case listing exactly one type, the variable has that type; otherwise, the variable has the type of the expression in the TypeSwitchGuard.

The type in a case may be `nil`; that case is used when the expression in the TypeSwitchGuard is a `nil` interface value.

Given an expression `x` of type `interface{}`, the following type switch:

```
switch i := x.(type) {
case nil:
    printString("x is nil")
case int:
    printInt(i)  // i is an int
case float64:
    printFloat64(i)  // i is a float64
case func(int) float64:
    printFunction(i)  // i is a function
case bool, string:
    printString("type is bool or string")  // i is an interface{}
default:
    printString("don't know the type")
}
```

could be rewritten:

```
v := x  // x is evaluated exactly once
if v == nil {
    printString("x is nil")
} else if i, isInt := v.(int); isInt {
    printInt(i)  // i is an int
} else if i, isFloat64 := v.(float64); isFloat64 {
    printFloat64(i)  // i is a float64
} else if i, isFunc := v.(func(int) float64); isFunc {
    printFunction(i)  // i is a function
} else {
    i1, isBool := v.(bool)
    i2, isString := v.(string)
    if isBool || isString {
        i := v
        printString("type is bool or string")  // i is an interface{}
    } else {
        i := v
        printString("don't know the type")  // i is an interface{}
    }
}
```

The type switch guard may be preceded by a simple statement, which executes before the guard is evaluated.

The "fallthrough" statement is not permitted in a type switch.

For statements

A "for" statement specifies repeated execution of a block. The iteration is controlled by a condition, a "for" clause, or a "range" clause.

```
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
Condition = Expression .
```

In its simplest form, a "for" statement specifies the repeated execution of a block as long as a boolean condition evaluates to true. The condition is evaluated before each iteration. If the condition is absent, it is equivalent to `true`.

```
for a < b {
    a *= 2
}
```

A "for" statement with a ForClause is also controlled by its condition, but additionally it may specify an *init* and a *post* statement, such as an assignment, an increment or decrement statement. The init statement may be a short variable declaration, but the post statement must not.

```
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .
```

```
for i := 0; i < 10; i++ {
    f(i)
}
```

If non-empty, the init statement is executed once before evaluating the condition for the first iteration; the post statement is executed after each execution of the block (and only if the block was executed). Any element of the ForClause may be empty but the semicolons are required unless there is only a condition. If the condition is absent, it is equivalent to `true`.

```
for cond { S() }    is the same as    for ; cond ; { S() }
for      { S() }    is the same as    for true     { S() }
```

A "for" statement with a "range" clause iterates through all entries of an array, slice, string or map, or values received on a channel. For each entry it assigns *iteration values* to corresponding *iteration variables* and then executes the block.

```
RangeClause = Expression [ "," Expression ] ( "=" | ":=" ) "range" Expression .
```

The expression on the right in the "range" clause is called the *range expression*, which may be an array, pointer to an array, slice, string, map, or channel. As with an assignment, the operands on the left must be addressable or map index expressions; they denote the iteration variables. If the range expression is a channel, only one iteration variable is permitted, otherwise there may be one or two. If the second iteration variable is the blank identifier, the range clause is equivalent to the same clause with only the first variable present.

The range expression is evaluated once before beginning the loop except if the expression is an array, in which case, depending on the expression, it might not be evaluated (see below). Function calls on the left are evaluated once per iteration. For each iteration, iteration values are produced as follows:

```
Range expression                          1st value          2nd value (if 2nd variable is present)

array or slice  a  [n]E, *[n]E, or []E    index    i  int    a[i]       E
string          s  string type            index    i  int    see below  rune
map             m  map[K]V                key      k  K      m[k]       V
channel         c  chan E                 element  e  E
```

1. For an array, pointer to array, or slice value `a`, the index iteration values are produced in increasing order, starting at element index 0. As a special case, if only the first iteration variable is present, the range loop produces iteration values from 0 up to `len(a)` and does not index into the array or slice itself. For a `nil` slice, the number of iterations is 0.

2. For a string value, the "range" clause iterates over the Unicode code points in the string starting at byte index 0. On successive iterations, the index value will be the index of the first byte of successive UTF-8-encoded code points in the string, and the second value, of type `rune`, will be the value of the corresponding code point. If the iteration encounters an invalid UTF-8 sequence, the second value will be `0xFFFD`, the Unicode replacement character, and the next iteration will advance a single byte in the string.

3. The iteration order over maps is not specified and is not guaranteed to be the same from one iteration to the next. If map entries that have not yet been reached are deleted during iteration, the corresponding iteration values will not be produced. If map entries are inserted during iteration, the behavior is implementation-dependent, but the iteration values for each entry will be produced at most once. If the map is `nil`, the number of iterations is 0.

4. For channels, the iteration values produced are the successive values sent on the channel until the channel is closed. If the channel is `nil`, the range expression blocks forever.

The iteration values are assigned to the respective iteration variables as in an assignment statement.

The iteration variables may be declared by the "range" clause using a form of short variable declaration (`:=`). In this case their types are set to the types of the respective iteration values and their scope ends at the end of the "for" statement; they are re-used in each iteration. If the iteration variables are declared outside the "for" statement, after execution their values will be those of the last iteration.

```
var testdata *struct {
    a *[7]int
}
for i, _ := range testdata.a {
    // testdata.a is never evaluated; len(testdata.a) is constant
    // i ranges from 0 to 6
    f(i)
}

var a [10]string
m := map[string]int{"mon":0, "tue":1, "wed":2, "thu":3, "fri":4, "sat":5, "sun":6}
for i, s := range a {
    // type of i is int
    // type of s is string
    // s == a[i]
    g(i, s)
}

var key string
var val interface {}  // value type of m is assignable to val
for key, val = range m {
    h(key, val)
}
// key == last map key encountered in iteration
// val == map[key]

var ch chan Work = producer()
for w := range ch {
    doWork(w)
}
```

Go statements

A "go" statement starts the execution of a function or method call as an independent concurrent thread of control, or *goroutine*, within the same address space.

```
GoStmt = "go" Expression .
```

The expression must be a call. The function value and parameters are evaluated as usual in the calling goroutine, but unlike with a regular call, program execution does not wait for the invoked function to complete. Instead, the function begins executing independently in a new goroutine. When the function terminates, its goroutine also terminates. If the function has any return values, they are discarded when the function completes.

```
go Server()
go func(ch chan<- bool) { for { sleep(10); ch <- true; }} (c)
```

Select statements

A "select" statement chooses which of a set of possible communications will proceed. It looks similar to a "switch" statement but with the cases all referring to communication operations.

```
SelectStmt = "select" "{" { CommClause } "}" .
CommClause = CommCase ":" { Statement ";" } .
CommCase   = "case" ( SendStmt | RecvStmt ) | "default" .
RecvStmt   = [ Expression [ "," Expression ] ( "=" | ":=" ) ] RecvExpr .
RecvExpr   = Expression .
```

RecvExpr must be a receive operation. For all the cases in the "select" statement, the channel expressions are evaluated in top-to-bottom order, along with any expressions that appear on the right hand side of send statements. A channel may be `nil`, which is equivalent to that case not being present in the select statement except, if a send, its expression is still evaluated. If any of the resulting operations can proceed, one of those is chosen and the corresponding communication and statements are evaluated. Otherwise, if there is a default case, that executes; if there is no default case, the statement blocks until one of the communications can complete. If there are no cases with non-`nil` channels, the statement blocks forever. Even if the statement blocks, the channel and send expressions are evaluated only once, upon entering the select statement.

Since all the channels and send expressions are evaluated, any side effects in that evaluation will occur for all the communications in the "select" statement.

If multiple cases can proceed, a uniform pseudo-random choice is made to decide which single communication will execute.

The receive case may declare one or two new variables using a short variable declaration.

```
var c, c1, c2, c3 chan int
var i1, i2 int
select {
case i1 = <-c1:
    print("received ", i1, " from c1\n")
case c2 <- i2:
    print("sent ", i2, " to c2\n")
case i3, ok := (<-c3):  // same as: i3, ok := <-c3
    if ok {
        print("received ", i3, " from c3\n")
    } else {
        print("c3 is closed\n")
    }
default:
    print("no communication\n")
}

for {  // send random sequence of bits to c
    select {
    case c <- 0:  // note: no statement, no fallthrough, no folding of cases
    case c <- 1:
    }
}
```

```
select {}  // block forever
```

Return statements

A "return" statement terminates execution of the containing function and optionally provides a result value or values to the caller.

```
ReturnStmt = "return" [ ExpressionList ] .
```

In a function without a result type, a "return" statement must not specify any result values.

```
func noResult() {
    return
}
```

There are three ways to return values from a function with a result type:

1. The return value or values may be explicitly listed in the "return" statement. Each expression must be single-valued and assignable to the corresponding element of the function's result type.

   ```
   func simpleF() int {
       return 2
   }

   func complexF1() (re float64, im float64) {
       return -7.0, -4.0
   }
   ```

2. The expression list in the "return" statement may be a single call to a multi-valued function. The effect is as if each value returned from that function were assigned to a temporary variable with the type of the respective value, followed by a "return" statement listing these variables, at which point the rules of the previous case apply.

   ```
   func complexF2() (re float64, im float64) {
       return complexF1()
   }
   ```

3. The expression list may be empty if the function's result type specifies names for its result parameters. The result parameters act as ordinary local variables and the function may assign values to them as necessary. The "return" statement returns the values of these variables.

   ```
   func complexF3() (re float64, im float64) {
       re = 7.0
       im = 4.0
       return
   }

   func (devnull) Write(p []byte) (n int, _ error) {
       n = len(p)
       return
   }
   ```

Regardless of how they are declared, all the result values are initialized to the zero values for their type upon entry to the function.

Break statements

A “break” statement terminates execution of the innermost “for”, “switch” or “select” statement.

```
BreakStmt = "break" [ Label ] .
```

If there is a label, it must be that of an enclosing “for”, “switch” or “select” statement, and that is the one whose execution terminates.

```
L:
    for i < n {
        switch i {
        case 5:
            break L
        }
    }
```

Continue statements

A “continue” statement begins the next iteration of the innermost “for” loop at its post statement.

```
ContinueStmt = "continue" [ Label ] .
```

If there is a label, it must be that of an enclosing “for” statement, and that is the one whose execution advances.

Goto statements

A “goto” statement transfers control to the statement with the corresponding label.

```
GotoStmt = "goto" Label .
```

```
goto Error
```

Executing the “goto” statement must not cause any variables to come into scope that were not already in scope at the point of the goto. For instance, this example:

```
    goto L  // BAD
    v := 3
L:
```

is erroneous because the jump to label `L` skips the creation of `v`.

A “goto” statement outside a block cannot jump to a label inside that block. For instance, this example:

```
if n%2 == 1 {
    goto L1
\}
for n > 0 {
    f()
    n--
L1:
    f()
    n--
\}
```

is erroneous because the label `L1` is inside the “for” statement’s block but the `goto` is not.

Fallthrough statements

A “fallthrough” statement transfers control to the first statement of the next case clause in a expression “switch” statement. It may be used only as the final non-empty statement in a case or default clause in an expression “switch” statement.

```
FallthroughStmt = "fallthrough" .
```

Defer statements

A “defer” statement invokes a function whose execution is deferred to the moment the surrounding function returns.

```
DeferStmt = "defer" Expression .
```

The expression must be a function or method call. Each time the “defer” statement executes, the function value and parameters to the call are evaluated as usual and saved anew but the actual function is not invoked. Instead, deferred calls are executed in LIFO order immediately before the surrounding function returns, after the return values, if any, have been evaluated, but before they are returned to the caller. For instance, if the deferred function is a function literal and the surrounding function has named result parameters that are in scope within the literal, the deferred function may access and modify the result parameters before they are returned. If the deferred function has any return values, they are discarded when the function completes.

```
lock(l)
defer unlock(l)  // unlocking happens before surrounding function returns

// prints 3 2 1 0 before surrounding function returns
for i := 0; i <= 3; i++ {
    defer fmt.Print(i)
}

// f returns 1
func f() (result int) {
    defer func() {
        result++
    }()
    return 0
}
```

Built-in functions

Built-in functions are predeclared. They are called like any other function but some of them accept a type instead of an expression as the first argument.

The built-in functions do not have standard Go types, so they can only appear in call expressions; they cannot be used as function values.

```
BuiltinCall = identifier "(" [ BuiltinArgs [ "," ] ] ")" .
BuiltinArgs = Type [ "," ExpressionList ] | ExpressionList .
```

Close

For a channel `c`, the built-in function `close(c)` records that no more values will be sent on the channel. It is an error if `c` is a receive-only channel. Sending to or closing a closed channel causes a run-time panic. Closing the nil channel also causes a run-time panic. After calling `close`, and after any previously sent values have been received, receive operations will return the zero value for the channel’s type without blocking. The multi-valued receive operation returns a received value along with an indication of whether the channel is closed.

Length and capacity

The built-in functions `len` and `cap` take arguments of various types and return a result of type `int`. The implementation guarantees that the result always fits into an `int`.

```
Call      Argument type    Result

len(s)    string type      string length in bytes
          [n]T, *[n]T      array length (== n)
          []T              slice length
          map[K]T          map length (number of defined keys)
          chan T           number of elements queued in channel buffer

cap(s)    [n]T, *[n]T      array length (== n)
          []T              slice capacity
          chan T           channel buffer capacity
```

The capacity of a slice is the number of elements for which there is space allocated in the underlying array. At any time the following relationship holds:

```
0 <= len(s) <= cap(s)
```

The length and capacity of a `nil` slice, map, or channel are 0.

The expression `len(s)` is constant if `s` is a string constant. The expressions `len(s)` and `cap(s)` are constants if the type of `s` is an array or pointer to an array and the expression `s` does not contain channel receives or function calls; in this case `s` is not evaluated. Otherwise, invocations of `len` and `cap` are not constant and `s` is evaluated.

Allocation

The built-in function `new` takes a type `T` and returns a value of type `*T`. The memory is initialized as described in the section on initial values.

```
new(T)
```

For instance

```
type S struct { a int; b float64 }
new(S)
```

dynamically allocates memory for a variable of type `S`, initializes it (`a=0`, `b=0.0`), and returns a value of type `*S` containing the address of the memory.

Making slices, maps and channels

Slices, maps and channels are reference types that do not require the extra indirection of an allocation with `new`. The built-in function `make` takes a type `T`, which must be a slice, map or channel type, optionally followed by a type-specific list of expressions. It returns a value of type `T` (not `*T`). The memory is initialized as described in the section on initial values.

```
Call             Type T     Result

make(T, n)       slice      slice of type T with length n and capacity n
make(T, n, m)    slice      slice of type T with length n and capacity m

make(T)          map        map of type T
make(T, n)       map        map of type T with initial space for n elements

make(T)          channel    synchronous channel of type T
make(T, n)       channel    asynchronous channel of type T, buffer size n
```

The arguments `n` and `m` must be of integer type. A run-time panic occurs if `n` is negative or larger than `m`, or if `n` or `m` cannot be represented by an `int`.

```
s := make([]int, 10, 100)       // slice with len(s) == 10, cap(s) == 100
s := make([]int, 10)            // slice with len(s) == cap(s) == 10
c := make(chan int, 10)         // channel with a buffer size of 10
m := make(map[string]int, 100)  // map with initial space for 100 elements
```

Appending to and copying slices

Two built-in functions assist in common slice operations.

The variadic function `append` appends zero or more values `x` to `s` of type `S`, which must be a slice type, and returns the resulting slice, also of type `S`. The values `x` are passed to a parameter of type `...T` where `T` is the element type of `S` and the respective parameter passing rules apply. As a special case, `append` also accepts a first argument assignable to type `[]byte` with a second argument of string type followed by `...`. This form appends the bytes of the string.

```
append(s S, x ...T) S  // T is the element type of S
```

If the capacity of `s` is not large enough to fit the additional values, `append` allocates a new, sufficiently large slice that fits both the existing slice elements and the additional values. Thus, the returned slice may refer to a different underlying array.

```
s0 := []int{0, 0}
s1 := append(s0, 2)        // append a single element     s1 == []int{0, 0, 2}
s2 := append(s1, 3, 5, 7)  // append multiple elements    s2 == []int{0, 0, 2, 3, 5, 7}
s3 := append(s2, s0...)    // append a slice              s3 == []int{0, 0, 2, 3, 5, 7, 0, 0}

var t []interface{}
t = append(t, 42, 3.1415, "foo")                           t == []interface{}{42, 3.1415, "foo"}

var b []byte
b = append(b, "bar"...)  // append string contents      b == []byte{'b', 'a', 'r' }
```

The function `copy` copies slice elements from a source `src` to a destination `dst` and returns the number of elements copied. Source and destination may overlap. Both arguments must have identical element type `T` and must be assignable to a slice of type `[]T`. The number of elements copied is the minimum of `len(src)` and `len(dst)`. As a special case, `copy` also accepts a destination argument assignable to type `[]byte` with a source argument of a string type. This form copies the bytes from the string into the byte slice.

```
copy(dst, src []T) int
copy(dst []byte, src string) int
```

Examples:

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
var b = make([]byte, 5)
n1 := copy(s, a[0:])            // n1 == 6, s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])            // n2 == 4, s == []int{2, 3, 4, 5, 4, 5}
n3 := copy(b, "Hello, World!")  // n3 == 5, b == []byte("Hello")
```

Deletion of map elements

The built-in function `delete` removes the element with key `k` from a map `m`. The type of `k` must be assignable to the key type of `m`.

```
delete(m, k)  // remove element m[k] from map m
```

If the element `m[k]` does not exist, `delete` is a no-op. Calling `delete` with a nil map causes a run-time panic.

Manipulating complex numbers

Three functions assemble and disassemble complex numbers. The built-in function `complex` constructs a complex value from a floating-point real and imaginary part, while `real` and `imag` extract the real and imaginary parts of a complex value.

```
complex(realPart, imaginaryPart floatT) complexT
real(complexT) floatT
imag(complexT) floatT
```

The type of the arguments and return value correspond. For `complex`, the two arguments must be of the same floating-point type and the return type is the complex type with the corresponding floating-point constituents: `complex64` for `float32`, `complex128` for `float64`. The `real` and `imag` functions together form the inverse, so for a complex value `z`, `z == complex(real(z), imag(z))`.

If the operands of these functions are all constants, the return value is a constant.

```
var a = complex(2, -2)             // complex128
var b = complex(1.0, -1.4)         // complex128
x := float32(math.Cos(math.Pi/2))  // float32
var c64 = complex(5, -x)           // complex64
var im = imag(b)                   // float64
var rl = real(c64)                 // float32
```

Handling panics

Two built-in functions, `panic` and `recover`, assist in reporting and handling run-time panics and program-defined error conditions.

```
func panic(interface{})
func recover() interface{}
```

When a function `F` calls `panic`, normal execution of `F` stops immediately. Any functions whose execution was deferred by the invocation of `F` are run in the usual way, and then `F` returns to its caller. To the caller, `F` then behaves like a call to `panic`, terminating its own execution and running deferred functions. This continues until all functions in the goroutine have ceased execution, in reverse order. At that point, the program is terminated and the error condition is reported, including the value of the argument to `panic`. This termination sequence is called *panicking*.

```
panic(42)
panic("unreachable")
panic(Error("cannot parse"))
```

The `recover` function allows a program to manage behavior of a panicking goroutine. Executing a `recover` call *inside* a deferred function (but not any function called by it) stops the panicking sequence by restoring normal execution, and retrieves the error value passed to the call of `panic`. If `recover` is called outside the deferred function it will not stop a panicking sequence. In this case, or when the goroutine is not panicking, or if the argument supplied to `panic` was `nil`, `recover` returns `nil`.

The `protect` function in the example below invokes the function argument `g` and protects callers from run-time panics raised by `g`.

```
func protect(g func()) {
    defer func() {
        log.Println("done")  // Println executes normally even if there is a panic
        if x := recover(); x != nil {
            log.Printf("run time panic: %v", x)
        }
    }()
    log.Println("start")
    g()
}
```

Bootstrapping

Current implementations provide several built-in functions useful during bootstrapping. These functions are documented for completeness but are not guaranteed to stay in the language. They do not return a result.

```
Function   Behavior

print      prints all arguments; formatting of arguments is implementation-specific
println    like print but prints spaces between arguments and a newline at the end
```

Packages

Go programs are constructed by linking together *packages*. A package in turn is constructed from one or more source files that together declare constants, types, variables and functions belonging to the package and which are accessible in all files of the same package. Those elements may be exported and used in another package.

Source file organization

Each source file consists of a package clause defining the package to which it belongs, followed by a possibly empty set of import declarations that declare packages whose contents it wishes to use, followed by a possibly empty set of declarations of functions, types, variables, and constants.

```
SourceFile       = PackageClause ";" { ImportDecl ";" } { TopLevelDecl ";" } .
```

Package clause

A package clause begins each source file and defines the package to which the file belongs.

```
PackageClause  = "package" PackageName .
PackageName    = identifier .
```

The PackageName must not be the blank identifier.

```
package math
```

A set of files sharing the same PackageName form the implementation of a package. An implementation may require that all source files for a package inhabit the same directory.

Import declarations

An import declaration states that the source file containing the declaration depends on functionality of the *imported* package and it enables access to exported identifiers of that package. The import names an identifier (PackageName) to be used for access and an ImportPath that specifies the package to be imported.

```
ImportDecl       = "import" ( ImportSpec | "(" { ImportSpec ";" } ")" ) .
ImportSpec       = [ "." | PackageName ] ImportPath .
ImportPath       = string_lit .
```

The PackageName is used in qualified identifiers to access exported identifiers of the package within the importing source file. It is declared in the file block. If the PackageName is omitted, it defaults to the identifier specified in the package clause of the imported package. If an explicit period (`.`) appears instead of a name, all the package's exported identifiers declared in that package's package block will be declared in the importing source file's file block and can be accessed without a qualifier.

The interpretation of the ImportPath is implementation-dependent but it is typically a substring of the full file name of the compiled package and may be relative to a repository of installed packages.

Implementation restriction: A compiler may restrict ImportPaths to non-empty strings using only characters belonging to Unicode's L, M, N, P, and S general categories (the Graphic characters without spaces) and may also exclude the characters `!"#$%&'()*,:;<=>?[\]^`{|}` and the Unicode replacement character U+FFFD.

Assume we have compiled a package containing the package clause `package math`, which exports function `Sin`, and installed the compiled package in the file identified by `"lib/math"`. This table illustrates how `Sin` may be accessed in files that import the package after the various types of import declaration.

```
Import declaration          Local name of Sin

import   "lib/math"         math.Sin
import M "lib/math"         M.Sin
import . "lib/math"         Sin
```

An import declaration declares a dependency relation between the importing and imported package. It is illegal for a package to import itself or to import a package without referring to any of its exported identifiers. To import a package solely for its side-effects (initialization), use the blank identifier as explicit package name:

```
import _ "lib/math"
```

An example package

Here is a complete Go package that implements a concurrent prime sieve.

```
package main

import "fmt"

// Send the sequence 2, 3, 4, … to channel 'ch'.
func generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i  // Send 'i' to channel 'ch'.
    }
}

// Copy the values from channel 'src' to channel 'dst',
// removing those divisible by 'prime'.
func filter(src <-chan int, dst chan<- int, prime int) {
    for i := range src {  // Loop over values received from 'src'.
        if i%prime != 0 {
            dst <- i  // Send 'i' to channel 'dst'.
```

```
        }
    }
}

// The prime sieve: Daisy-chain filter processes together.
func sieve() {
    ch := make(chan int)  // Create a new channel.
    go generate(ch)       // Start generate() as a subprocess.
    for {
        prime := <-ch
        fmt.Print(prime, "\n")
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
    }
}

func main() {
    sieve()
}
```

Program initialization and execution

The zero value

When memory is allocated to store a value, either through a declaration or a call of `make` or `new`, and no explicit initialization is provided, the memory is given a default initialization. Each element of such a value is set to the *zero value* for its type: `false` for booleans, `0` for integers, `0.0` for floats, `""` for strings, and `nil` for pointers, functions, interfaces, slices, channels, and maps. This initialization is done recursively, so for instance each element of an array of structs will have its fields zeroed if no value is specified.

These two simple declarations are equivalent:

```
var i int
var i int = 0
```

After

```
type T struct { i int; f float64; next *T }
t := new(T)
```

the following holds:

```
t.i == 0
t.f == 0.0
t.next == nil
```

The same would also be true after

```
var t T
```

Program execution

A package with no imports is initialized by assigning initial values to all its package-level variables and then calling any package-level function with the name and signature of

```
func init()
```

defined in its source. A package may contain multiple `init` functions, even within a single source file; they execute in unspecified order.

Within a package, package-level variables are initialized, and constant values are determined, in data-dependent order: if the initializer of `A` depends on the value of `B`, `A` will be set after `B`. It is an error if such dependencies form a cycle. Dependency analysis is done lexically: `A` depends on `B` if the value of `A` contains a mention of `B`, contains a value whose initializer mentions `B`, or mentions a function that mentions `B`, recursively. If two items are not interdependent, they will be initialized in the order they appear in the source. Since the dependency analysis is done per package, it can produce unspecified results if `A`'s initializer calls a function defined in another package that refers to `B`.

An `init` function cannot be referred to from anywhere in a program. In particular, `init` cannot be called explicitly, nor can a pointer to `init` be assigned to a function variable.

If a package has imports, the imported packages are initialized before initializing the package itself. If multiple packages import a package `P`, `P` will be initialized only once.

The importing of packages, by construction, guarantees that there can be no cyclic dependencies in initialization.

A complete program is created by linking a single, unimported package called the *main package* with all the packages it imports, transitively. The main package must have package name `main` and declare a function `main` that takes no arguments and returns no value.

```
func main() { … }
```

Program execution begins by initializing the main package and then invoking the function `main`. When the function `main` returns, the program exits. It does not wait for other (non-`main`) goroutines to complete.

Package initialization—variable initialization and the invocation of `init` functions—happens in a single goroutine, sequentially, one package at a time. An `init` function may launch other goroutines, which can run concurrently with the initialization code. However, initialization always sequences the `init` functions: it will not start the next `init` until the previous one has returned.

Errors

The predeclared type `error` is defined as

```
type error interface {
    Error() string
}
```

It is the conventional interface for representing an error condition, with the nil value representing no error. For instance, a function to read data from a file might be defined:

```
func Read(f *File, b []byte) (n int, err error)
```

Run-time panics

Execution errors such as attempting to index an array out of bounds trigger a *run-time panic* equivalent to a call of the built-in function `panic` with a value of the implementation-defined interface type `runtime.Error`. That type satisfies the predeclared interface type `error`. The exact error values that represent distinct run-time error conditions are unspecified.

```
package runtime

type Error interface {
    error
    // and perhaps other methods
}
```

System considerations

Package unsafe

The built-in package `unsafe`, known to the compiler, provides facilities for low-level programming including operations that violate the type system. A package using `unsafe` must be vetted manually for type safety. The package provides the following interface:

```
package unsafe

type ArbitraryType int  // shorthand for an arbitrary Go type; it is not a real type
type Pointer *ArbitraryType

func Alignof(variable ArbitraryType) uintptr
func Offsetof(selector ArbitraryType) uintptr
func Sizeof(variable ArbitraryType) uintptr
```

Any pointer or value of underlying type `uintptr` can be converted into a `Pointer` and vice versa.

The function `Sizeof` takes an expression denoting a variable of any type and returns the size of the variable in bytes.

The function `Offsetof` takes a selector denoting a struct field of any type and returns the field offset in bytes relative to the struct's address. For a struct `s` with field `f`:

```
uintptr(unsafe.Pointer(&s)) + unsafe.Offsetof(s.f) == uintptr(unsafe.Pointer(&s.f))
```

Computer architectures may require memory addresses to be *aligned*; that is, for addresses of a variable to be a multiple of a factor, the variable's type's *alignment*. The function `Alignof` takes an expression denoting a variable of any type and returns the alignment of the (type of the) variable in bytes. For a variable `x`:

```
uintptr(unsafe.Pointer(&x)) % unsafe.Alignof(x) == 0
```

Calls to `Alignof`, `Offsetof`, and `Sizeof` are compile-time constant expressions of type `uintptr`.

Size and alignment guarantees

For the numeric types, the following sizes are guaranteed:

type	size in bytes
byte, uint8, int8	1
uint16, int16	2
uint32, int32, float32	4
uint64, int64, float64, complex64	8
complex128	16

The following minimal alignment properties are guaranteed:

1. For a variable `x` of any type: `unsafe.Alignof(x)` is at least 1.
2. For a variable `x` of struct type: `unsafe.Alignof(x)` is the largest of all the values `unsafe.Alignof(x.f)` for each field `f` of `x`, but at least 1.
3. For a variable `x` of array type: `unsafe.Alignof(x)` is the same as `unsafe.Alignof(x[0])`, but at least 1.

A struct or array type has size zero if it contains no fields (or elements, respectively) that have a size greater than zero. Two distinct zero-size variables may have the same address in memory.

Credits

All of these documents are taken from the Go homepage:

- "How to Write Go Code": `http://golang.org/doc/code.html`
- "Effective Go": `http://golang.org/doc/effective_go.html`
- "The Go Language Specification": `http://golang.org/ref/spec`

The Gopher that appears on the cover is from the `docs/gopher` directory in the source packages (`docs/gopher/frontpage.png`).

The title page was developed from the template from the "Title Creation" section of the LaTeXwikibook.

See Also

- Go, The Standard Library (`http://thestandardlibrary.com/go.html`) by Daniel Huckstep, covers the Go standard library in great detail. It is supported by excellent code examples.