

Go NOTES FOR TIC-80

Fergus Baker

January 29, 2023

Notes for using Go with TIC-80 by targeting WASM with `tinygo`.

CONTENTS

1	Language overview	1
1.1	Types	1
1.2	Declarations	2
1.3	Structs	2
1.4	Functions	2
1.5	Methods	3
1.6	Arrays	3
1.7	Maps	4
1.8	Control flow	4
1.9	Interfaces	5
1.10	Errors	6
1.11	Generics	6
1.12	Exports and imports	7
2	Goroutines and concurrency	7
2.1	Channels	7
2.2	Example	8
2.3	Mutexes	8
3	Common interfaces	8
3.1	Readers and writers	8
4	Standard patterns	8
4.1	Errors and recovery	8
4.2	Parallelism	9
4.3	Enums	9
5	TIC-80 with WASM	9
6	Using tinygo to target TIC-80	10
6.1	Setup	10
6.2	Modifying <code>tic80.go</code>	10

1 LANGUAGE OVERVIEW

1.1 TYPES

See also [1.9 Interfaces](#) concerning *type assertions*.

Go supports the primitives listed on the right hand side.

```
// integral
int, int8, int16, int32, int64
uint, uint8, uint16, uint32, uint64, uintptr
// floating point
float32, float64
complex64, complex128
// other
string, bool
// aliases
byte // uint8
rune // int32 for unicode
```

Types may be converted to some new type `T` using `T(v)` .

Type aliases may be defined using the `type` keyword.

```
type MyFloat float64
```

1.2 DECLARATIONS

Variables may either be explicitly typed or inferred. Constants may be inferred, and support arbitrary precision until coerced.

```
// explicit type
var i int
// explicit type initialized
var i int = 1
// implicit type
i := 1
// constant
const Pi = 3.14
```

Pointers are either declared to point to an array or a variable. References can be taken with the `&` operator.

```
// pointer to array
var a []int
x = a[1]
// pointer to type
var p *int
// dereference
x = *p
// reference
q := &x
```

The allocation primitives are `make` and `new`, and apply to different types. `new` allocates and zeros memory, returning a **pointer** `*T`. The keyword `make` is reserved only for slices, maps, and channels, and **does not** return a pointer.

```
// p is *MyCustomType
p := new(MyCustomType)
// v is MyCustomType
var v MyCustomType
```

1.3 STRUCTS

Structs are a collection of fields, which are {} initialized. Pointers to structs have a free level of indirection, thus `(*p).x` is the same as `p.x`. Uninitialized fields are implicitly zero.

```
type Vertex struct {
    X int
    Y int
}
v1 := Vertex{1, 2}
v2 := Vertex{Y: 2} // X implicitly 0
```

Go implements the concept of **constructors as factories**, which is conventionally the name of the struct prefixed with `New`.

```
func NewVertex(x, y int) NewVertex {
    return NewVertex{x, y}
}
```

1.4 FUNCTIONS

Functions must be explicitly typed and support multiple (named) return types. Function are first class citizens and may be assigned to variables. Functions support closure capture.

```
// function type
func(int32, int32) int32

// anonymous: a and b have same type
adder := func(a, b int32) int32 {return a + b}

// single return type
func foo(a int32, b int32) int32 {
    return a + b
}
```

```
// multiple returns
func mfoo(a, b int32) (int32, int32) {
    return a, b
}

// multiple returns named
func bar(a int32, b int32) (out1 int32, out2 int32) {
    out1 = a
    out2 = b
    return
}

// closure capture
func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}
```

The address of a **local variable** may be returned without issue: the storage of a variable survives the function context. Referencing an r-value **allocates a new** instance.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    // new instance each time it is called
    return &File{fd: fd, name:name}
}
```

1.5 METHODS

Methods may be defined on **types** (such as custom structs). Methods have a special **receiver** argument.

```
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X * v.X + v.Y * v.Y)
}

// invocation
v.Abs()
```

For methods to be mutating they must be declared with **pointer receivers**.

```
func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}
```

1.6 ARRAYS

Go arrays are 0 indexed. Fixed size arrays are declared with `[n]T` syntax. Slices are dynamically sized references to arrays, declared with `[]T`. Slices may be literal.

When slicing, the bounds are implicitly the start and end if excluded.

The **len** of a slice is the number of elements it contains, whereas the **cap** is the number of elements in the underlying array.

nil slices are slices with length and capacity equal to 0.

```
// array of 10 ints
var a [10]int
// slice to 3 ints and indices 1, 2, 3
b := a[1:4]
// slice literal
c := []bool{false, false, false}
```

```
s := []int{2, 3, 5, 7, 11, 13}
// len=6 cap=6 [2 3 5 7 11 13]
s = s[:0]
// len=0 cap=6 []
s = s[:4]
// len=4 cap=6 [2 3 5 7]
s = s[2:]
// len=2 cap=4 [5 7]
```

Slices may be **dynamically allocated** with the

```
// len 5, cap 5
```

`make` function, which allocates and zeros out an array.

```
a := make([]int, 5)
// len 0, cap 5
b := make([]int, 0, 5)
```

1.7 MAPS

A map is a key-value store. The zero value of a map is `nil`. Maps are dynamically allocated and must be initialized with `make`.

Maps may be **mutated** with the usual `[]` syntax. When an entry is read, the map returns both the value and an `ok` boolean. If the key is not in the map, the value is a zero and `ok` is `false`.

Map literals may also be declared.

```
// variable declaration
var m = map[string]int
// init
m = make(map[string]int)
```

```
// add entry
m["Hello World"] = 42
// read entry
value, ok := m["Hello World"]
// remove
delete(m, "Hello World")
```

```
// map literal
m1 = map[string]int{
    "Hello": 13,
    "World": 12,
}
```

1.8 CONTROL FLOW

See also [1.9 Interfaces](#) concerning *type switches* for control flow.

A **defer** statement defers the execution of a function until the surrounding function returns. Defers are executed in LIFO order.

```
defer fmt.Println("world")
fmt.Println("hello")
```

The **for** loop has an initializer, a condition, and a post statement, with the initializer and post statement being optional.

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

The **while** loops have the same syntax, though the semi colons may be dropped.

```
// while
for i < 10 {
    // ...
}
```

Infinite loops are created without any arguments.

```
// infinite
for {
    // ...
}
```

For loops may also be used with **range indexing**, implicitly enumerating the array or slice.

```
var pow = []int{1, 2, 4, 8}
for i, v := range pow {
    // ...
}
```

If statements have need not have brackets, and support optional capture initializers. The variable in the initializer only exists for the scope of the ‘if’ block.

```
if some_condition {
    // ...
} else {
    // ...
}
// with a capture init
if y := a + b; y < 10 {
    return y
}
```

The **switch** statement can be used on any primitive. Cases do not have fall-through, and the cases need not be constants.

```
switch os := runtime.GOOS; os {
case "darwin":
    // ...
case "linux":
    // ...
default:
    // ...
}
```

In the example to the right, `f()` is not invoked unless `i != 0`. A switch statement without a condition is the same as `switch true`.

```
switch i {
case 0:
    // ...
case f():
    // ...
}
```

1.9 INTERFACES

From what I can tell, the Go-ism for interface names is to end the interface with `-er`, e.g. `fmt.Stringer`.

Interfaces are **types** that define a set of method signatures. They may be thought of as a tuple of (value, type)

```
type Absoluter interface {
    Abs() float64
}
```

The **nil interface** is the interface which implement no methods, such as primitives. It is declared with `interface{}`

```
var i interface{}
```

Variables of instance type may be declared and instantiated by any type which implements the interface.

```
type SomeFloat float64
func (f SomeFloat) Abs() float64 {
    // ...
}
// instantiate interface
var a Absoluter
a = SomeFloat(1.0)
```

Care must be taken when dealing with pointers. In this example, `*Vector` *does* implement the interface, but `Vector` *does not*.

```
type Vector struct {
    X, Y float64
}
func (v *Vector) Abs() float64 {
    // ...
}
// ok
v := Vector{}
var a Absoluter = &v
// error
a = v
```

Type assertions may be used to access an interfaces concrete value. It returns a (value, ok) tuple, where the `ok` boolean denotes whether the type assertion was true. The syntax is `t, ok := i.(T)` to assert type `T`.

```
var i interface{} = "Hello World"
// ok is true
s, ok := i.(string)
// ok is false, v is 0.0
f, ok := i.(float64)
```

Switches may also be used on interfaces as control flow with **type switches**.

```
switch v := i.(type) {
case T:
    // here v has type T
case S:
    // here v has type S
default:
    // no match; here v has the same type as i
}
```

Interfaces may **embed** or extend existing interfaces by including them in the definition.

Go differentiates between **basic** and **non-basic** interfaces, where basic interfaces may be entirely implemented and initialized, whereas non-basic interfaces are used primarily in 1.11 Generics.

Non-basic interfaces are interfaces with **type-unions** (the pipe operator), or which embed other non-basic interfaces.

```
type A interface {
    GetName() string
}
// embeds A
type B interface {
    A
    SetValue(v int)
}
```

```
// the above A and B are both basic
// the below C and D are non-basic
type C interface {
    int | int64 | float64
}
type D interface {
    C
    Content() string
}
```

1.10 ERRORS

The error type is an interfaces that implements the `Error() string` method.

Errors are normally returned in a tuple with the result. Errors are handled by testing `err != nil`.

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

```
i, err := strconv.Atoi("42")
if err != nil {
    // ...
    return
}
```

Go also has `panic` and `recover`. The `panic` keyword is reserved for errors that are “unrecoverable”, and `recover` is to recover from them. When a `panic` is called, go immediately begins the stack unwind until it hits a `recover`.

`recover` always returns `nil` unless called from a deferred function.

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

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

1.11 GENERICS

Functions may accept **type parameters** for generics. This parameter appears in `[]` before the functions arguments. The type parameter must fulfill a **constraint**.

Constraints may be defined through interfaces. The super-type of all interfaces is `any`. The pipe syntax denotes type unions.

```
// T must support == comparison
func Index[T comparable](s []T, x T) int {
    // ...
}
```

```
type Number interface {
    int | int64 | float64
}
```

1.12 EXPORTS AND IMPORTS

Exported functions from a package must begin with a capital letter. Packages which make use of `helloworld` may only refer to `helloworld.Foo`.

Go also uses **compiler directives** in the form of annotated functions with **comments** to export or import symbols at link-time.

```
package helloworld
// not exported
func foo() {
    // ....
}
// is exported
func Foo() {
    // ...
}
```

```
// link the symbol _start to the function Init
//go:linkname Init _start
func Init()

// import a function symbol
//go:export
func add(x, y int) int

// export a function symbol
//go:export
func sub(x, y int) int {
    return x - y
}
```

2 GOROUTINES AND CONCURRENCY

A **goroutine** is a lightweight thread that is managed by the runtime. The keyword `go` is reserved for starting a new goroutine. They run in the same address space. The `sync` package provides goroutine primitives.

2.1 CHANNELS

Channels are a typed pipe for IO, and may be used to send or receive with the **channel operator**, i.e. `<-`. Channels are by default blocking to allow goroutines to synchronize without explicit locking mechanisms.

Channels may be **buffered**, which means they have a fixed size and will result in a deadlock if trying to send to a full buffer.

Channels may be **closed** to indicate no additional values will be sent. A second return argument indicates whether a channel is closed or not. It is not necessary to *always* close channels, and should be used only to indicate no additional information.

The `range` keyword may be used in a for-loop to read all values from a channel until closed.

The `select` keyword is analogous to the `switch` statement for waiting on multiple operations. The `select` blocks execution until one of its cases can run, then executes that case. It is non-deterministic if multiple cases are ready simultaneously. The `default` case is optional, and will run if no other cases are ready.

```
// initialize integer channel
ch := make(chan int)
// send
ch <- value
// receive
v := <- ch
```

```
// buffered channel with 100 elements
ch := make(chan int, 100)
```

```
ch := make(chan int)
ch <- 10
// close channel
close(ch)
// ok == false since channel closed
v, ok := <- ch
```

```
// read until closed
for i := range ch {
    // ...
}
```

```
select {
// will only run if ch not full
case ch <- x:
    x = x + 1
// will run if can read from recv
case <- recv:
    fmt.Println("received")
}
```

```
// optional default
default:
    fmt.Println("no operation")
}
```

2.2 EXAMPLE

Below is a concurrent example for summing numbers in an array:

```
package main

func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}
    // make a channel for io between goroutines
    c := make(chan int)
    // spawn two goroutines that sum different parts of s
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    // receive
    x, y := <-c, <-c
}
```

2.3 MUTEXES

The `sync.Mutex` (mutual exclusion) can be used when multiple goroutines need to access the same resource without worrying about race conditions. The standard mutex supports `Lock` and `Unlock` methods.

```
mu := sync.Mutex()
mu.Lock()
// practice is to defer unlocks
defer mu.Unlock()
```

3 COMMON INTERFACES

3.1 READERS AND WRITERS

The `io` package specifies a `io.Reader` interface, which declares the `Read` method.

```
// prototype
func (T) Read(b []byte) (n int, err error)
// example
r := strings.NewReader("Hello, Reader!")
buffer := make([]byte, 8)
for {
    n, err := r.Read(buffer)
    // ...
    if err == io.EOF {
        break
    }
}
```

4 STANDARD PATTERNS

4.1 ERRORS AND RECOVERY

Throw errors with `panic` so that goroutines can handle and recover as needed.

```
func (s *S) error(err string) {
    panic(Error(err))
}
```



```
func Runner(str string) (s *S, err error) {
    s = new(S)
    defer func() {
        if e := recover(); e != nil {
            // Clear return value.
            regexp = nil
            // will re-panic cannot coerce
            err = e.(Error)
        }
    }()
    // will panic if there is a call error
    return s.Call(str), nil
}
```

4.2 PARALLELISM

Goroutines can be used for concurrency by initializing e.g. multiple channels.

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

func (v Vector) DoAll(u Vector) {
    // buffered array
    c := make(chan int, numCPU)
    fraction := len(v) / numCPU
    for i := 0; i < numCPU; i++ {
        go v.DoSome(
            // lower index
            i * fraction,
            // upper index
            (i+1) * fraction,
            u,
            c
        )
    }
    // drain the channel.
    for i := 0; i < numCPU; i++ {
        // wait for one task to complete
        <-c
    }
    // all done.
}
```

4.3 ENUMS

Although Go does not have a concept of an enum, there is the builtin `iota`, which is an automatically incrementing integer scoped to `const` blocks with the initial value of 0. It can be used to quickly define monotonically increasing constants. `iota` may also be used in expressions.

```
const (
    A = 1
    B = 2
    C = 4
)
// becomes
const (
    A = iota + 1
    B
    - // skip
    C
)
```

5 TIC-80 WITH WASM

In order to use WASM with the TIC-80, the TIC-80 executable must be compiled with `-DBUILD_PRO=0n`. The full setup procedure is then

```
git clone "https://github.com/nesbox/TIC-80"
```

```
cd TIC-80/build
# run cmake
cmake .. -DBUILD_PRO=On
make -j4
# link binary
sudo ln -s $(pwd)/bin/tic80 /usr/local/bin/tic80
```

Refer to the [TIC-80 readme](#) for full installation instructions for your OS.

When executing a *.wasmp script, we also require a setupfile with tile set, wave forms, etc. This is already included in the example repository (see [6 Using tinygo to target TIC-80](#)). We will generally use the CLI arguments `--skip --fs .` to skip the startup animation and to mount the current working directory as the filesystem.

We load and start the WASM executable with

```
load wasmdemo.wasmp
import binary cart.wasm
run
```

6 USING TINYGO TO TARGET TIC-80

We have an example repository, containing a Makefile for targetting TIC-80 with `tinygo` in [fjebaker/global-game-jam-2023](#). It is also worth reading the `tinygo` documentation on the [Differences from Go](#).

Finally, it is also worth noting that someone has already implemented a Go module for the TIC-80 [sorucoder/tic80](#) which we can use to base our implementation on. I am reluctant to just use this module as we won't really learn the memory map overview, and wrapping new functions is not too difficult.

6.1 SETUP

`tinygo` requires all sorts of setup configuration, but is probably easiest to use directly from the Docker image provided by the maintainers. As long as you have a Docker runtime installed and running, the Makefile included in the example repository should work fine.

Inline with the restrictions of the TIC-80, we have a `target.json` file included which sets up the memory topology and linker flags needed. Interesting to note is that although we *should not need* to provide an entry point, `go` **still requires** that the `_start` symbol is `invoke` to setup the garbage collector and thread runtime. We can delegate this task to the `BOOT` function.

```
//go:export BOOT
func BOOT() {
    tic80.Init()
}
// still need this since _start calls main
func main() {}
```

6.2 MODIFYING tic80.go

The full memory map of the TIC-80 is probably easiest to understand by looking at the C example [here](#). We add new functions simply with the compiler directives. For example, the `print` function might look like:

```
import unsafe

//go:export print
func print(textBuffer unsafe.Pointer, x, y int32, color, fixed, scale, alt int8) int32
```