What is Go?

• “Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.” (From https://golang.org)

• Conceived in September 2007 at Google by R. Griesemer, R. Pike and K. Thompson, and announced in November 2009

• Goals of the language and its tools:
  – To be expressive, efficient in both compilation and execution, and effective in writing reliable and robust programs

• Go’s ancestors: mainly C and CSP (communicating sequential processes) formal language by T. Hoare
Go and C

• Go: “C-like language” or “C for the 21st century”
• From C, Go inherited
  – Expression syntax
  – Control-flow statements
  – Basic data types
  – Call-by-value parameter passing
  – Pointers
  – Emphasis on programs that compile to efficient machine code and cooperate naturally with the abstractions of OSs

Go and other languages

• New and efficient facilities for concurrency
• Flexible approach to data abstraction and object-oriented programming
• Automatic memory management (garbage collection)
Go and distributed systems

- Go allows you to concentrate on distributed systems problems
  - good support for concurrency
  - good support for RPC
  - garbage-collected (no use after freeing problems)
  - type safe
- Simple language to learn

References

- Online Go tutorial
  https://tour.golang.org/

- Go by Examples
  https://gobyexample.com

Hello world example

package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}

Some notes on the first example

• No semicolon at the end of statements or declarations
• Go natively handles Unicode
• Every Go program is made up of packages (similar to libraries or packages)
  – Package: one or more .go source files in a single directory
• Source file begins with package declaration (which package the file belongs to), followed by list of other imported packages
  – Programs start running in main
  – fmt package contains functions for printing formatted output and scanning input
• Import statement: groups the imports into a parenthesized, “factored” statement (see echo example)
Go tool

- Go is a compiled language
- Go tool: the standard way to fetch, build, and install Go packages and commands
- To run the program, use `go run`
  
  ```
  $ go run hello-world.go
  hello, 世界
  ```
- To build the program into binary, use `go build`
  
  ```
  $ go build hello-world.go
  $ ls hello-world*
  hello-world     hello-world.go
  $ ./hello-world
  hello, 世界
  ```

Functions

- A function can take zero or more arguments
  
  ```
  func add(x int, y int) int {
      return x + y
  }
  ```
- add takes two parameters of type `int`
- The type comes after the variable name
- Shorter version for the input arguments:
  
  ```
  func add(x, y int) int {
  }
  ```
- A function can return any number of results
  
  ```
  func swap(x, y string) (string, string) {
      return y, x
  }
  ```
Variables

- `var` statement: declares a list of variables
  - The type is last
- `var` statement: can be at package or function level
  ```go
  package main
  import "fmt"
  
  var c, python, java bool
  
  func main() {
    var i int
    fmt.Println(i, c, python, java)
  }
  ```
  - Can include initializers, one per variable
  - If an initializer is present, the type can be omitted
- Short variable declaration using `:=`

Types

- Usual basic types
  - `bool, string, int, uint, float32, float64, ...`
- Type conversion
  ```go
  var i int = 42
  var f float64 = float64(i)
  ```
- Type inference
  ```go
  var i int
  j := i // j is an int
  ```
Flow control statements

• for, if, else, switch
• defer

Looping construct

• Go has only one looping construct: the for loop
• Three components
  – Init statement
  – Condition expression
  – Post statement
  ```go
  sum := 0
  for i := 0; i < 10; i++ {
    sum += i
  }
  ```
• No parentheses surrounding the three components of the for statement and the braces { } are always required
Looping construct

• Init and post statements are optional: for is Go's "while"

```go
sum := 1
for sum < 1000 {
    sum += sum
}
```

• Omit the condition: infinite loop

```go
for {
}
```

Example: echo

```go
// Echo1 prints its command-line arguments.
package main
import (
    "fmt"
    "os"
)
func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

`os.Args` is a slice of strings (see next slides)
o and s initialized to empty strings
Conditional statements

• Go's if (and else) statements are like its for loops; expression not surrounded by parentheses ( ) but braces { } are required
  
  ```go
  if v := math.Pow(x, n); v < lim {
    return v
  } else {
    fmt.Printf("%g >= %g\n", v, lim)
  }
  ```

  – Remember that } else must be on the same line

• Also switch statement

Defer statement

• New mechanism to defer the execution of a function until the surrounding function returns
  – The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function that contains defer has terminated

  ```go
  package main
  import "fmt"

  func main() {
    defer fmt.Println("world")
    defer fmt.Println("world")
    fmt.Println("hello")
  }
  ```

  • Deferred function calls pushed onto a stack
    – Deferred calls executed in LIFO order

  • Great for things like closing files or connections!
Composite data types: pointers and arrays

- **Pointer**: holds the memory address of a value
  
  ```go
  var p *int
  ```

- **Aggregate data types**: structs and arrays

- **Struct**: a collection of fields
  - Syntax similar to C, fixed size
    ```go
    type Vertex struct {
      X int
      Y int
    }
    ```

- **Array**: type `[n]T` is an array of `n` values of type `T`
  - Fixed size
    ```go
    var a [2]string
    a[0] = "Hello"
    ```

Composite data types: slices

- **Slice**: dynamically-sized, flexible view into the elements of an array
  - Specify two indices, a low and high bound, separated by a colon: `s[i : j]`
  - Include first element, but excludes last one
    ```go
    primes := [6]int{2, 3, 5, 7, 11, 13}
    var s []int = primes[1:4]
    ```

  - It is a section of an *underlying array*: modify the elements of the corresponding array
  - Can be created with the built-in `make` function
  - New items can be appended to a slice using the built-in `append` function
  - It’s a compile or run-time error to exceed the length (bounds-checked)
Composite data types: maps

• map: maps keys to values
  – Map type map[K]V is a reference to a hash table where K and V are the types of its keys and values
  – Use built-in function make to create a map
    \[
    m = \text{make(map[string]Vertex)}
    \]
    \[
    m["Bell Labs"] = \text{Vertex}\{40.68433, -74.39967,}
    \}
  
  • You can insert or update an element in a map, retrieve an element, delete an element, test if a key is present

Range

• range iterates over elements in a variety of data structures
  – range on arrays and slices provides both the index and value for each entry
  – range on map iterates over key/value pairs
### Range: example

```go
func main() {
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    for k := range kvs {
        fmt.Println("key:", k)
    }
}
```

$ go run range.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b

### Methods

- **Go does not have classes**
  - You can define methods on types
- **A method is a function with a special receiver argument** (extra parameter before the function name)
  - The receiver appears in its own argument list between the `func` keyword and the method name

```go
type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
}
```
Interfaces

- An **interface type** is defined as a named collection of method signatures
- Any type (struct) that implements the required methods, implements that interface
- A type is not explicitly declared to be of a certain interface, it is implicit
  - Just implement the required methods

**Interface: example**

```go
package main

import "fmt"
import "math"

// Here's a basic interface for geometric shapes.
type geometry interface {
    area() float64
    perim() float64
}

// For our example we'll implement this interface on `rect` and `circle` types.
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}
```
// To implement an interface in Go, we just need to
// implement all the methods in the interface. Here we
// implement `geometry` on `rect`s.
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

// The implementation for `circle`s.
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

// If a variable has an interface type, then we can call
// methods that are in the named interface. Here's a
// generic `measure` function taking advantage of this
// to work on any `geometry`.
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    // The `circle` and `rect` struct types both
    // implement the `geometry` interface so we can use
    // instances of
    // these structs as arguments to `measure`.
    measure(r)
    measure(c)

Concurrency in Go

• Go provides concurrency features as part of the core language
• Goroutines and channels
  – Support CSP concurrency model
• Can be used to implement different concurrency patterns

Goroutines

• A goroutine is a lightweight thread managed by the Go runtime
  
  \[
  \text{go } f(x, y, z) \text{ starts a new goroutine running } f(x, y, z)
  \]
• Goroutines run in the same address space, so access to shared memory must be synchronized
Channels

• Communication mechanism that lets one goroutine sends values to another goroutine
  – A channel is a thread-safe queue that the Go language and runtime manages for you
  – It does the right thing with blocking threads that read on it, etc.

• Hides a lot of pain of inter-thread communication
  – Internally, it uses mutexes and semaphores just as one might expect

• Multiple senders can write to the same channel
  – This is really useful for notifications, multiplexing, etc.

• And it’s totally thread-safe!

• But be careful: only one can close the channel, and can’t send after close!

Channels

• A typed conduit through which you can send and receive values using the channel operator `<-

  ```go
  ch <- v      // Send v to channel ch
  v := <-ch   // Receive from ch, and
              // assign value to v
  ```

• Channels must be created before use

  ```go
  ch := make(chan int)
  ```

• Sends and receives block until the other side is ready
  – Goroutines can synchronize without explicit locks or condition variables
import "fmt"

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

func main() {
    s := []int{7, 2, 8, -9, 4, 0}
    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c // receive from c

    fmt.Println(x, y, x+y)
}

More on channels

- Channels can be buffered
  - Buffer length as the second argument to make to initialize a buffered channel
    
    ```go
    ch := make(chan int, 100)
    ```
  - Sends to a buffered channel block only when the buffer is full
  - Receives block when the buffer is empty

- Close and range on buffers
  - Sender can close a channel
  - Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression
    
    ```go
    v, ok := <-ch
    ```
  - ok is false if there are no more values to receive and the channel is closed
  - Use the loop for i := range c to receive values from the channel repeatedly until it is closed
More on channels

- Select can be used to wait for messages on one of several channels
  ```go
  select {
    case <-ch1:
      // ...
    case x := <-ch2:
      // ...use x...
    case ch3 <- y:
      // ...
    default:
      // ...
  }
  ```

- You can implement timeouts by using a timer channel
  ```go
  // to wait 2 seconds
  timer := time.NewTimer(time.Second * 2)
  <- timer.C
  ```

Error handling

- Go code uses error values to indicate abnormal state
- Errors are communicated via an explicit, separate return value
  - By convention, the last return value of a function
  - nil value in the error position: no error
  - “Error handling [in Go] does not obscure the flow of control.” (R. Pike)
    ```go
    result, err := SomeFunction()
    if err != nil {
      // handle the error
    }
    ```

- Built-in error interface
  ```go
  type error interface {
    Error() string
  }
  ```

- errors.New constructs a basic error value with the given error message
  ```go
  ```

See [https://blog.golang.org/error-handling-and-go](https://blog.golang.org/error-handling-and-go)
A few more things

• Go can be somewhat picky
• Unused variables raise errors, not warnings
  – Use "_" for variables you don’t care about
• Unused imports raise errors
  – “goimports” is a command to automatically add/remove imports
    https://godoc.org/golang.org/x/tools/cmd/goimports
• In if-else statements { must be placed at the end of the same line
  – E.g.: } else {
  – E.g.: } else if ... {
  – “gofmt” is a command to auto-indent code
    https://golang.org/cmd/gofmt/

RPC in Go

• Go standard library has support for RPC right out-of-the-box
  – Package net/rpc of the standard Go library
    https://golang.org/pkg/net/rpc/
• Constraints for RPC methods
  – only two arguments are allowed
  – second argument is a pointer
  – an error is always returned
  
    func (t *T) MethodName(argType T1, replyType *T2) error

• Use gob package for parameters marshaling (encode) and unmarshaling (decode)
  https://golang.org/pkg/encoding/gob/
  – gob manages streams of gobs (binary values)
RPC in Go: server

• On the server side
  – Use Register
    • To publish the methods that are part of the given interface on the default RPC server and allows them to be called by clients connecting to the service
  – Use Accept to receive connections on the listener and serve requests for each incoming connection
  – Can also use HTTP handler for RPC messages (see example on the course site)

RPC in Go: client

• On the client side
  – Use Dial to connect to an RPC server at the specified network address
  – Use call to invoke the synchronous RPC
  – Use Go to invoke the asynchronous RPC
    • Associated channel will signal when the call is complete
RPC in Go: synchronous call

- Need some setup in advance of this...
- The method `net/rpc/Call` makes a blocking RPC call

```go
// Synchronous call
args := &server.Args{7, 8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
    log.Fatal("arith error:", err)
}
```

RPC in Go: asynchronous call

- The method `net/rpc/Go` uses a channel as parameter to retrieve the RPC reply when the call is finished

```go
// Asynchronous call
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args, quotient, nil)
replyCall := <-divCall.Done // will be equal to divCall
// check errors, print, etc.
```

```go
func (client *Client) Go(serviceMethod string, args interface{}, reply interface{}, done chan *Call) *Call
```

- For Go’s internal implementation, see [https://golang.org/src/net/rpc/client.go?s=8029:8135#L284](https://golang.org/src/net/rpc/client.go?s=8029:8135#L284)
Summing up: Two styles of RPC implementations

- **Shallow integration:** must use lots of library calls to set things up
  - How to format data
  - Registering which functions are available and how they are invoked

- **Deep integration**
  - Data formatting done based on type declarations
  - (Almost) all public methods of object are registered

- Go is the latter