GoBackwards Lanuage Manual

Shaquan Nelson, Catherine Zhao, Tahmid Munat, Julian Silerio, Peter Richards

sdn2115, caz2114, tfm2109, jjs2245, pfr2109

COMSW4115 Programming Language and Translators

Contents

1 Introduction 3

2 Scope 3
   2.1 Uses 3
   2.2 Restrictions and Limits 4

3 Environment 4
   3.1 Program Structure 4
   3.2 Program Startup 7

4 Language 7
   4.1 Characters 8
   4.2 Types 8
   4.3 Lexical Elements & Keywords 8
   4.4 Declarations 8

5 Library 9
   5.1 Introduction 9
   5.2 Error 9
   5.3 IO/IOUtil 10
      5.3.1 Functions - ReadFile 11
   5.4 HTTP 11
      5.4.1 Defaults 11
      5.4.2 Function 12
1 Introduction

This is a reference manual for the programming language GoBackwards. GoBackwards offers an innovative approach to Go and its problems by making a simpler, easier to understand language while also implementing the web-server platform that Go is known for.

This manual will cover types of programs that can be created, syntax, operators, error handling, and included libraries.

2 Scope

2.1 Uses

This language is useful for web server development. Programs can be built to handle TCP/IP protocols, HTTP requests and more. GoBackwards also allows users to create and store data types.

Sample programs uses include: wifi modules (ex: ESP8266), websites, curl/wget commands, and data storage. The object-oriented nature of this language makes for a ton of possibilities in terms of data structures. The internet protocols allow for a plethora of possibilities in terms of real world application.
2.2 Restrictions and Limits

This language will not worry about garbage collection. Also, users will not be able to add new keywords, overload operators or import new packages (for the sake of this project). These restrictions will allow users to create robust programs that run almost instantly because they depend solely on the provided GoBackwards library.

3 Environment

3.1 Program Structure

GoBackwards’ language structure will help organize data and have a collection of variables of different types.

In keeping up with the ANSI standard, we define structure assignment - that is structures may be copied and assigned to, passed to and returned by functions. Similar to the traditional Go language, we have a structure defined and used as follows:

```
Listing 1: Example of a Structure in GoBackwards
package main
    type Vertex struct {
        X int
        Y int
    }

    func main() {
        Println(Vertex{1, 2})
        // This will print {1 2}
    }
```

The first keyword struct introduces a structure declaration, which is defined by its type. So, We are following a mix of Go, C and Python in our program.

Struct fields are accessed using a dot(.) symbol. For example, using the previous example:

```
Listing 2: Example of a Accessing a Structure in GoBackwards
func main() {
```

As we are going to utilize the net/http package, we also need to account for pointers and accessing structs through a struct pointer. In Go, a pointer holds the memory address of a value. For example, the type *P is a pointer to a P value. The & operator generates a pointer to its operand. We can see the following example to explore this in more detail:

Listing 3: Struct and Pointers in GoBackwards

```go
func main() {
    i, j := 1, 2

    p := &i // points to i
    println(*p) // reads i through pointer (print 1)

    *p = 7 // sets i through the pointer
    println(i) // reads the new value of i (print 7)

    p = &j // points to j
    *p = *p / 2 // divides j through the pointer
    println(j) // reads the new value of j (print 1)
}
```

Now we can access struct field using struct pointers in the following way:

Listing 4: Struct using Pointers in GoBackwards

```go
func main() {
    v := Vertex{1, 2}
    p := &v
    p.X = 42
    println(v) // prints {42 2}
}
```

Function signatures can vary. For example, from the Go Language manual, we have the following example: func (p *Page) save() error. The sig-
nature means "this is a method named save that takes as its receiver p, a pointer to Page. It takes no parameters, and returns a value of type error." (https://golang.org/doc/articles/wiki/").

Similar to Go, in GoBackwords, functions are also values that can be passed around like other values. So we can have a function declared and defined outside of the main function and use it in main. Functions can take zero or more arguments and function values may be used as arguments and return values. GoBackwards uses the type name after the variable name in arguments, such as 'x int' or 'x, y int.'

Following is an example of using structs and functions:

```
Listing 5: Example Program using Pointers

type rect struct {
    width, height int //same line declaration
}

func (r *rect) area() int {
    return r.width * r.height
}

func main() {
    r => {width: 10, height: 5}
    println("area: ", r.area())
    rp => &r
    println("area: ", rp.area())
}
```

The compilation will be

```
$ gob build sample.gob
$ ./sample
$ area: 50
$ area: 50
```

A 'var' statement can be used to declare a list of variables. The type is used at the end. This statement can be at package or function level. For example:
Variables can be initialized. If an initializer is present, the type can be omitted as follows:

Variables declared without an explicit initial value are given their zero value: 0 for numeric types, false for booleans, and "" (the empty string) for strings.

In GoBackwards, the ⇒ short assignment statement can be used in place of a var declaration in a function (with implicit type). The ⇒ construct is not available outside a function. This is a change from the := construct in Go. In ASCII notation, we will use = followed by a > sign.

GoBackwards will not worry about garbage collection or manual memory management for the time being. Other restrictions are listed under Restriction and Limits(2.2).

### 3.2 Program Startup

We can run the GoBackwords program from the command line. In a *NIX environment, we can use the command 'go build' with the name of the file (for example, sample.gob) to build and then the ./filename command (in this case: ./sample) to run.

```
$ gob build sample.gob
$ ./sample
```

We are preserving the main function theme of Go, meaning we use the main function of the program to determine what to run.

### 4 Language

This section outlines the language of GoBackwards.
4.1 Characters

Source code is Unicode text encoded in UTF-8. Each code point is distinct; for instance, upper and lower case letters are different characters. Unlike full Go, the compiler will allow both the NUL character (U+0000) and the byte order mark (U+FEFF) in the source text. Just as in full Go, 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. The characters of GoBackwards are made up of newline, unicode_char, unicode_letter, and unicode_digit.

4.2 Types

Just as in full Go, a type determines a set of values together with operations and methods specific to those values. To denote a type, use a type name (if applicable) or a type literal. If using a type other than the standard boolean, numeric, or sting, it must be declared with a type declaration.

4.3 Lexical Elements & Keywords

There are two ways to write a comment in GoBackwards. For a single line comment, start the comment with the character sequence //. For a multiline comment, start the comment with /* and end it with */.

GoBackwards is made up of identifiers, keywords, operators and punctuation, and literals, just like full Go. Just as in full Go, white space is ignored except as it separates tokens that would otherwise combine into a single token. GoBackwards uses single semicolons “;” as terminators.

Regarding Identifiers, GoBackwards uses sequences of at least one letter(s) and digit(s). The following are reserved and cannot 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 are the same as full Go: + & += &= && == != ( ) - | -|= |= || < <= [ ] * ^ *= ^= < - > >= { } / <<= <<= ++ = := , % >> %= >>= -= ! ... . : &^ &^=

4.4 Declarations

Similar to Go, a declaration binds a non-blank identifier to a constant, type, variable, function, label, 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.

GoBackwards does not support the declaration of blank identifiers. The scope of every declaration is determined by its position within the program. In lieu of packages, GoBackwards defines scope universally within the program of declaration. The compiler will reject any identifier that is declared by never used.

5 Library

5.1 Introduction

Unlike Go, GoBackwards will not have the abilities to import libraries. This section will go over libraries that will implemented in GoBackwards. In particular, this will be able to create a basic web server.

5.2 Error

This package will define customs errors. GoBackwards code will use error values to indicate an abnormal state.

We utilize `New` in Go that returns an error that formats as the given text.

```go
func New(text string) error
```

We can create a struct and implement functions to manipulate errors as we see fit.

Listing 8: error implementation that includes a time and message

```go
// Sample GoBackwords error struct based on Go
type CustomError struct {
    When time.Time
    What string
}

func (e MyError) Error() string {
    return Println(e.When, e.What)
}
```
Another useful operation for our http package could be giving the user a simple error message that contains the status code.

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

func issue() error {
    return CustomError{
        time.Date(2017, 16 10 00, 0, 0, time.UTC),
        "This should be checked for errors",
    }
}

func main() {
    if err := issue(); err != nil {
        println(err)
    }
}
```

Now we can use the struct to modify the appHandler type. Then we can return *appError values.

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

Overall, similar to Go, Gobackwards error type is an interface type. Here is the interface’s declaration:

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

### 5.3 IO/IOUtil

The I/O function allows basic utility functions. In the server, this will be able to read html files.
5.3.1 Functions - ReadFile

    func ReadFile(filename string) ([]byte, error)

    ReadFile uses the string named filename to return the content. If the call was successful, the error is null. If the file exist, but does not contain anything EOF will be reported.

5.4 HTTP

This sections will focus on the build-in function for a HTTP-Server. In particular it helps implements HTTP client and server communications.

5.4.1 Defaults

This section will cover assumptions for the packages.

5.4.1.1 Common HTTP methods

<table>
<thead>
<tr>
<th>Listing 11: Common HTTP methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>const</strong> (</td>
</tr>
<tr>
<td>MethodGet = &quot;GET&quot;</td>
</tr>
<tr>
<td>MethodHead = &quot;HEAD&quot;</td>
</tr>
<tr>
<td>MethodPost = &quot;POST&quot;</td>
</tr>
<tr>
<td>MethodPut = &quot;PUT&quot;</td>
</tr>
<tr>
<td>MethodPatch = &quot;PATCH&quot;</td>
</tr>
<tr>
<td>MethodDelete = &quot;DELETE&quot;</td>
</tr>
<tr>
<td>MethodConnect = &quot;CONNECT&quot;</td>
</tr>
<tr>
<td>MethodOptions = &quot;OPTIONS&quot;</td>
</tr>
<tr>
<td>MethodTrace = &quot;TRACE&quot;</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

5.4.1.2 Common Error Handling Messages

This section takes the standard error code from industry. This is the same context from the Net-http package. See Appendix A
5.4.2 Function
5.4.2.1 HandleFunc

    func HandleFunc(pattern string,
     handler func(RespsonseWrite, *Request))

This function will match the string and execute a function by using predefined path from the url. A simple example is `http.HandleFunc("/hello", HelloServer)`. The language will extract the url to find the word "/hello". This will then redirect to the function `HelloServer` to be executed.

5.4.2.2 ListenandServe

    func ListenAndServe(addr string, handler Handler) error

This function will listen to an address addr and handle incoming request. This enables the HandleFunc to be respond. An simple examples is `http.ListenAndServe(":1111",nil)`. Once the program is ran the local host 1111 will be connected to this program. If the connection is not found, null will be returned.

5.4.3 Structures
5.4.3.1 Client

A client is an HTTP clients. It has the following functions Do, Get, Head, Post, and PostForm. The structure of client is shown below.

<table>
<thead>
<tr>
<th>Listing 12: Structure for Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>type Client struct {</td>
</tr>
<tr>
<td>CheckRedirect func(req *Request, via []*Request) error</td>
</tr>
</tbody>
</table>
|}

This is simplified from Go due to limiting the complexity.

We will now go into each function

    func (c *Client) Do(req *Request) (*Response, error)

This function will return a Response and error. Do sends an HTTP request and returns an HTTP response. An error is returned when there is no HTTP connection. If there is no error, the Response will contain a Body.

The following is the Get function:
func (c *Client) Get(url string) (resp *Response, err error)

The Get function returns a Response and error. The Response’s body will return the content of the site. This is similar to the curl function in terminal. If the response contains a redirect code, it will use client’s redirect code to the correct function. The code are as followed:

301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
308 (Permanent Redirect)

The following is the Head function:

func (c *Client) Head(url string) (resp *Response, err error)

The Get function returns a Response and error. The Response’s body will return the head content of the specified URL. It also contains the same redirect function as Get.

The following is the Post function:

func (c *Client) Post(url string, contentType string, body io.Reader) (resp *Response, err error)

Post issues a POST to the specified URL.

func (c *Client) PostForm(url string, data url.Values) (resp *Response, err error)

PostForm issues a POST to a specific URL. The data’s keys and values URL-encoded as the request body.

The following contains a simple example using client:

```
Listing 13: http-client Example

func main() {
    resp, err := http.Get("beesbeesbees.com")
    if err != nil {
        println("Error in Get")
    }
    defer resp.Body.Close()
}
```
body, err := ioutil.ReadFile(resp.Body)
Println(body)

The previous code returns

$ gob build http-client.gob
$ ./http-client.gob
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<title>beesbeesbees.com</title>
</head>
<frameset>
<frame src="http://www.maxistentialism.com/bees" name="redir">
<noframes>
<p>Original location:
</p>
</noframes>
</frameset>
</html>

5.4.3.2 URL
The URL structure is as followed:

Listing 14: URL struct

```go
type URL struct {
    Scheme string
    // encoded opaque data
    Opaque string
    // username and password information
    User *Userinfo
}
```
The general form represented is:

```
[scheme:]//[userinfo@]host[/]path[?query][#fragment]
```

URLs that do not start with a slash after the scheme are interpreted as:

```
scheme:opaque[?query][#fragment]
```

This section is the same as Go’s URL struct.

### 5.4.3.3 UserInfo

This section is identical to Go’s `userInfo`. It is used in the URL.

Listing 15: Stuct of UserInfo
```go
type Userinfo struct {
    // contains filtered or unexported fields
}
```

The following contains the function for Userinfo

```go```
```
User returns a Userinfo containing the provided username and no password set.

    func UserPassword(username, password string) *Userinfo

UserPassword returns a Userinfo containing the provided username and password.
This functionality should only be used with legacy web sites. RFC 2396 warns that interpreting Userinfo this way is NOT RECOMMENDED, because the passing of authentication information in clear text (such as URI) has proven to be a security risk in almost every case where it has been used.

    func (u *Userinfo) Password() (string, bool)

Password returns the password in case it is set, and whether it is set.

    func (u *Userinfo) String() string

String returns the encoded userinfo information in the standard form of "username[:password]".

    func (u *Userinfo) Username() string

Username returns the username.

### 5.4.3.4 Response

Listing 16: Response Struct

type Response struct {
    Status string // e.g. "200 OK"
    StatusCode int // e.g. 200
    Proto string // e.g. "HTTP/1.0"
    ProtoMajor int // e.g. 1
    ProtoMinor int // e.g. 0

    Header Header

    Body io.ReadCloser
}
Header maps header keys to values. If the response had multiple headers with the same key, they may be concatenated, with comma delimiters.

Body io.ReadCloser

Body represents the response body.

ContentLength int64

ContentLength records the length of the associated content.

TransferEncoding []string

Contains transfer encodings from outer-most to inner-most.

Close bool

Close records whether the header directed that the connection be closed after reading Body.

Uncompressed bool
Uncompressed reports whether the response was sent compressed but was decompressed by the http package.

Trailer Header

Trailer maps trailer keys to values in the same format as Header.

Request *Request

Request is the request that was sent to obtain this Response.

5.4.3.5 ResponseWriter

A ResponseWriter interface is used by an HTTP handler to construct an HTTP response.

Listing 17: ResponseWriter Interface

type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(int)
}

The following is a more descriptive definition of the struct:

Header() Header

Header returns the header map that will be sent by WriteHeader.

Write([]byte) (int, error)

Write writes the data to the connection as part of an HTTP reply.

WriteHeader(int)

WriteHeader sends an HTTP response header with status code.
5.4.3.6 Request

An HTTP Request received by a server or to be sent by a client.

Listing 18: Request Struct

```go
type Request struct {
    Method string // GET, POST, PUT, etc.
    URL *url.URL // Server URL to access or URI requested
    Proto string // "HTTP/1.0"
    ProtoMajor int // 1
    ProtoMinor int // 0
    Header Header // HTTP request header fields
}
```

Other function for Request include:

```go```
func NewRequest(method, url string) (*Request, error)

NewRequest returns a new Request for a given method and URL.

func ReadRequest(b string) (*Request, error)

ReadRequest returns and returns a Request from b.

func Write(r Request) (string, error)

Write writes an HTTP/1.1 Request. It writes the request in the format below and returns it as a string.

Listing 19: HTTP Request format

```
Host
URL
Method (defaults to "GET")
Header
ContentLength
TransferEncoding
```
### 5.4.3.7 Server

#### Listing 20: Server Struct

```go
type Server struct {
    Addr string // Address for server to listen on
    Handler handler
    ReadTimeout time.Duration
    ReadHeaderTimeout time.Duration
    IdleTimeout time.Duration
}
```

The parameters for running an HTTP Server.

```go
func (srv *Server) Close() error
```

Closes all active listeners on a server. If any errors occur, this function returns them accordingly.

```go
func (srv *Server) ListenAndServe() error
```

Listens to the specified server address and handles requests on incoming connections.

```go
func (srv *Server) Serve(l net.Listener) error
```

Accepts incoming connections and uses the server handler to respond accordingly.

```go
func (srv *Server) Shutdown(ctx context.Context) error
```

Closes all active listeners on a server, then all idle listeners, then waits for other connections to become idle before closing. Once all connections have been closed, the server shuts down. If any errors occur, this function returns them accordingly.
5.5 String
5.5.1 Functions
5.5.1.1 Compare

    func Compare(a, b string) int

Returns a value to determine if two strings are equivalent. 0 if \( a == b \). -1 if \( a < b \). +1 if \( b > a \).

5.5.1.2 Contains

    func Contains(a, b string) bool

Returns true if the string \( b \) is contains in \( a \).

5.5.1.3 Count

    func Count(a, b string) int

Returns the number of time the string \( b \) appears in \( a \) without overlapping.

5.5.1.4 Join

    func Join(a []string, sep string) string

Returns a string containing the string in \( a \) separated by \( sep \).

5.5.1.5 Index

    func Index(s, substr string) int

Returns the index of the first time \( substr \) appears in \( s \). If the \( substr \) is not contained in \( s \), it will return -1.

5.5.1.6 Split

    func Split(s, sep string) []string

Returns an array of string separated by \( sep \). If \( sep==s \) or \( sep \) is not contained in \( s \), then it will return \( s \) in a array. If \( sep=="\" \), then it will separate each character.
5.5.1.7 Length

```go
func Len(a int
```

Returns the length of the string.

5.5.2 Simple Example

The code below details example code and output for the string library.

```go
Listing 21: String example
```
```
func main() {
    Println(string.Compare(“a”, “b”))
    \ False
    Println(string.Contains(“abc”, “b”))
    \ True
    Println(string.Count(“acdc”, “c”))
    \ 2
    Println(string.Count(“aaaaaaaa”, “aa”))
    \ 4
    Println(string.Join(“rock”, “ and roll”))
    \ “rock and roll”
    Println(string.Index(“i love rock and roll”, “ro”))
    \ 7
    Println(string.Split(“i love rock and roll”, “ ”))
    \ [“i”, “love”, “rock”, “and”, “roll”]
    Println(string.Split(“i love rock and roll”, “ro”))
    \ [“i love “, “ck and “, “ll “]
    Println(string.Split(“hello”, “hello”))
    \ [“h”, “e”, “l”, “l”, “o”]
    Println(string.Len(“hello world”))
```
```
```
}
```

5.6 Time

This section examines the built-in time package that allows the display and measurement of time.
5.6.1 Constants

By default time is returned in the following format, as specified in the Go manual. Time will be implemented in UTC, and we’ll ignore Daylight savings for this implementation.

Mon Jan 2 15:04:05 2006

The following constants can be passed to Time.Format to format the time accordingly, as specified in the Go manual.

```
const (  
    ANSI   = "Mon Jan _2 15:04:05 2006"
    RubyDate = "Mon Jan 02 15:04:05 2006"
    RFC822  = "02 Jan 06 15:04"
    RFC850  = "Monday, 02-Jan-06 15:04:05"
    RFC1123 = "Mon, 02 Jan 2006 15:04:05"
    RFC3339 = "2006-01-02T15:04:05Z07:00"
    RFC3339Nano = "2006-01-02T15:04:05.999999999Z07:00"
    Kitchen = "3:04PM"
    Stamp   = "Jan _2 15:04:05"
    StampMilli = "Jan _2 15:04:05.000"
    StampMicro = "Jan _2 15:04:05.000000"
    StampNano = "Jan _2 15:04:05.000000000"
)
```

5.6.2 Functions

5.6.2.1 After

```
func After(d Duration) Time

Returns the time after the given duration is reached.
```

5.6.2.2 Sleep

```
func Sleep(d Duration)

Exits the current process after the given duration is reached.
```
5.6.3 Types
5.6.3.1 Duration

Listing 22: Duration struct

type Duration int

The time between two instances represented in nanoseconds of type int. To use a different unit of measure, simply multiply the duration by the time format desired.

```go
define duration
seconds => 10
  time.Duration(seconds)*time.Second
  // returns 10s
```

See below for the supported Duration formats.

Listing 23: Supported Duration Formats

```go
const (
  Nanosecond   Duration = 1
  Microsecond  = 1000 * Nanosecond
  Millisecond  = 1000 * Microsecond
  Second       = 1000 * Millisecond
  Minute       = 60   * Second
  Hour         = 60   * Minute
)
```

The following contains the since function:

```go
func Since(t Time) Duration
Returns the time passed since t. Basically a shortcut for time.Now().Sub(t).
```

The following contains the util function:

```go
func Until(t Time) Duration
Returns the time until t. Basically a shortcut for t.Sub(time.Now()).
```

The following contains the hour function:

```go
func (d Duration) Hours() float
Returns the duration in hours as type float.
```

The following contains the minutes function:

```go
func (d Duration) Minutes() float
```
func (d Duration) Minutes() float

Returns the duration in minutes as type float.

The following contains the nanoseconds function:

func (d Duration) Nanoseconds() int

Returns the duration in hours as type int.

The following contains the seconds function:

func (d Duration) Seconds() float

Returns the duration in seconds as type float.

The following contains the string function:

func (d Duration) String() string

Returns the duration in the form of a string in the same format Go does ("72h3m0.5s"). As the units for the duration get smaller, the format will use the appropriate units to make sure the string doesn’t begin with a 0. As in Go, a duration of 0 returns the string "0s".

Listing 24: Month struct

| type Month | int |

A month of the year in type int.

const (  
January Month = 1  
February Month = 2  
March Month = 3  
April Month = 4  
May Month = 5  
June Month = 6  
July Month = 7  
August Month = 8  
September Month = 9  
October Month = 10  
November Month = 11  
December Month = 12  
)
The following contains the `String` function:

```go
func (m Month) String() string
```

Returns the name of the corresponding month as a string.

```
Listing 25: Time struct

type Time struct {
    year int
    month Month
    day, hour, min, sec, nsec int
}
```

A measurement of time with nanosecond precision.

The following contains the `Date` function:

```go
func Date(year int,
    month Month,
    day, hour, min, sec, nsec int,
) Time
```

Returns a `Time` object with the date specified in the following format:

```
yyyy-mm-dd hh:mm:ss + nsec nanoseconds
```

The following contains the `Now` function:

```go
func Now() Time
```

Returns the current local time.

The following contains the `Add` function:

```go
func (t Time) Add(d Duration) Time
```

Returns the sum of the `Time` `t` and `Duration` `d` as type `Time`.

The following contains the `AddDate` function:

```go
func (t Time) AddDate(years int, months int, days int) Time
```

Returns the sum of the `Time` `t` with the date given.

The following contains the `After` function:
func (t Time) After(u Time) bool

Returns whether or not the Time t is after the Time u.

The following contains the before function:

func (t Time) Before(u Time) bool

Returns whether or not the Time t is before the Time u.

The following contains the clock function:

func (t Time) Clock() (hour, min, sec int)

Returns the hour, minute, and seconds for the Time given.

The following contains the date function:

func (t Time) Date() (year int, month Month, day int)

Returns the year, month, and day (otherwise known as the date) for the Time given.

The following contains the date function:

func (t Time) Day() int

Returns the day of the month for the Time given.

The following contains the equal function:

func (t Time) Equal(u Time) bool

Returns whether or not two Time objects are equal. These objects can be equal despite having different locations.

The following contains the format function:

func (t Time) Format(layout string) string

Returns the Time t as a string in the format described by layout. The formats are described at the beginning of this section.

The following contains the hour function:

func (t Time) Hour() int

Returns the hour of the Time given.

The following contains the isZero function:

func (t Time) IsZero() bool
Returns whether or not the time given is the zero time instant as described by Go (January 1, year 1, 00:00:00).

The following contains the minute function:

```
func (t Time) Minute() int
```

Returns the minute(s) of the Time given.

The following contains the month function:

```
func (t Time) Month() Month
```

Returns the month of the Time given.

The following contains the nanosecond function:

```
func (t Time) Nanosecond() int
```

Returns the nanosecond(s) of the time given.

The following contains the second function:

```
func (t Time) Second() int
```

Returns the second(s) of the time given.

The following contains the String function:

```
func (t Time) String() string
```

Returns the Time given as a string. The string format is in the form "2006-01-02 15:04:05.999999999"

The following contains the sub function:

```
func (t Time) Sub(u Time) Duration
```

Returns the Duration passed between Time objects t and u.

The following contains the weekday function:

```
func (t Time) Weekday() Weekday
```

Returns the Weekday of the Time given.

The following contains the year function:

```
func (t Time) Year() int
```

Returns the year of the Time given.

The following contains the yearDay function:

```
func (t Time) YearDay() int
```

Returns the day of the year for the Time given. This day falls in the range [1,365] for regular years and [1,366] for leap years.
5.6.3.4 Weekday

Listing 26: Weekday struct

```go
type Weekday int
```

The day of the week given by an int. The week begins on Sunday.

Listing 27: Day Constant

```go
const (
    Sunday Weekday = 0
    Monday Weekday = 1
    Tuesday Weekday = 2
    Wednesday Weekday = 3
    Thursday Weekday = 4
    Friday Weekday = 5
    Saturday Weekday = 6
)
```

String

```go
func (d Weekday) String() string
```

Returns the name of the day for a given Weekday.

Appendices

A Common Error Handling Messages

```go
const (
    StatusContinue = 100 // RFC 7231, 6.2.1
    StatusSwitchingProtocols = 101 // RFC 7231, 6.2.2
    StatusProcessing = 102 // RFC 2518, 10.1
    StatusOK = 200 // RFC 7231, 6.3.1
    StatusCreated = 201 // RFC 7231, 6.3.2
    StatusAccepted = 202 // RFC 7231, 6.3.3
)
StatusNonAuthoritativeInfo = 203 // RFC 7231, 6.3.4
StatusNoContent = 204 // RFC 7231, 6.3.5
StatusResetContent = 205 // RFC 7231, 6.3.6
StatusPartialContent = 206 // RFC 7233, 4.1
StatusMultiStatus = 207 // RFC 4918, 11.1
StatusAlreadyReported = 208 // RFC 5842, 7.1
StatusIMUsed = 226 // RFC 3229, 10.4.1

StatusMultipleChoices = 300 // RFC 7231, 6.4.1
StatusMovedPermanently = 301 // RFC 7231, 6.4.2
StatusFound = 302 // RFC 7231, 6.4.3
StatusSeeOther = 303 // RFC 7231, 6.4.4
StatusNotModified = 304 // RFC 7232, 4.1
StatusUseProxy = 305 // RFC 7231, 6.4.5

StatusTemporaryRedirect = 307 // RFC 7231, 6.4.7
StatusPermanentRedirect = 308 // RFC 7538, 3

StatusBadRequest = 400 // RFC 7231, 6.5.1
StatusUnauthorized = 401 // RFC 7235, 3.1
StatusPaymentRequired = 402 // RFC 7231, 6.5.2
StatusForbidden = 403 // RFC 7231, 6.5.3
StatusNotFound = 404 // RFC 7231, 6.5.4
StatusMethodNotAllowed = 405 // RFC 7231, 6.5.5
StatusNotAcceptable = 406 // RFC 7231, 6.5.6
StatusProxyAuthRequired = 407 // RFC 7235, 3.2
StatusRequestTimeout = 408 // RFC 7231, 6.5.7
StatusConflict = 409 // RFC 7231, 6.5.8
StatusGone = 410 // RFC 7231, 6.5.9
StatusLengthRequired = 411 // RFC 7231, 6.5.10
StatusPreconditionFailed = 412 // RFC 7232, 4.2
StatusRequestEntityTooLarge = 413 // RFC 7231, 6.5.11
StatusRequestURITooLong = 414 // RFC 7231, 6.5.12
StatusUnsupportedMediaType = 415 // RFC 7231, 6.5.13
StatusRequestedRangeNotSatisfiable = 416 // RFC 7233, 4.4
StatusExpectationFailed = 417 // RFC 7231, 6.5.14
StatusTeapot = 418 // RFC 7168, 2.3.3
StatusUnprocessableEntity = 422 // RFC 4918, 11.2
StatusLocked = 423 // RFC 4918, 11.3
StatusFailedDependency = 424 // RFC 4918, 11.4
StatusUpgradeRequired = 426 // RFC 7231, 6.5.15
StatusPreconditionRequired = 428 // RFC 6585, 3
StatusTooManyRequests = 429 // RFC 6585, 4
StatusRequestHeaderFieldsTooLarge = 431 // RFC 6585, 5
StatusUnavailableForLegalReasons = 451 // RFC 7725, 3

StatusInternalServerError = 500 // RFC 7231, 6.6.1
StatusNotImplemented = 501 // RFC 7231, 6.6.2
StatusBadGateway = 502 // RFC 7231, 6.6.3
StatusServiceUnavailable = 503 // RFC 7231, 6.6.4
StatusGatewayTimeout = 504 // RFC 7231, 6.6.5
StatusHTTPVersionNotSupported = 505 // RFC 7231, 6.6.6
StatusVariantAlsoNegotiates = 506 // RFC 2295, 8.1
StatusInsufficientStorage = 507 // RFC 4918, 11.5
StatusLoopDetected = 508 // RFC 5842, 7.2
StatusNotExtended = 510 // RFC 2774, 7
StatusNetworkAuthenticationRequired = 511 // RFC 6585, 6

)