The Go Language Report (CSCI 208) — Jingya Wu

Note: to compile and run sample code files included in this report, first make sure that go is installed on your machine, then cd into the directory `code_samples`, and follow the comment at the start of each file.

Background

As stated in the official FAQ site of Go [1], the language was started on September 21, 2007 by three researchers at Google — Robert Griesemer, Rob Pike and Ken Thompson. Go became a public open source project on November 10, 2009. Many people from the community have contributed ideas, discussions, and code. Go is a compiled language [2]. The purpose of this language is to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. Looking at the job market, several Go programs deployed in production inside Google. Some companies that are currently using Go are: BuzzFeed, Comcast, Dell, Docker, Dropbox, Ebay, Facebook, iBM, Lyft, Medium, Mozilla, Netflix, Oracle, Pinterest, Reddit, Slack, Twitter, Uber, VMWare. A full list of companies can be accessed here [3].

[1] https://golang.org/doc/faq#Origins (The Go Programming Language FAQ — Origins)

[2] https://golang.org/doc/ (Documentation - The Go Programming Language)

[3] https://github.com/golang/go/wiki/GoUsers (GoUsers Github Wiki)

Primitive Types

From the Types section in the Go Programming Language Specification [1], Go has four primitive types — boolean, numeric, string, and error 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`.
- 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

- A string type represents the set of string values. A string value is a (possibly empty) sequence of bytes. Strings are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is `string`.
- An error type is a primitive (or predeclared) type that is defined as [2]

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

The following program shows the memory size of each of these primitive types:

```
// To compile and run: go run primitives.go
package main
import (
  "fmt"
```

```
  "unsafe"
)
func main() {
  var b bool
  fmt.Printf("Size of bool is %d\n", unsafe.Sizeof(b))
  var ui8 uint8
  fmt.Printf("Size of uint8 is %d\n", unsafe.Sizeof(ui8))
  var ui16 uint16
  fmt.Printf("Size of uint16 is %d\n", unsafe.Sizeof(ui16))
  var ui32 uint32
  fmt.Printf("Size of uint32 is %d\n", unsafe.Sizeof(ui32))
  var ui64 uint64
  fmt.Printf("Size of uint64 is %d\n", unsafe.Sizeof(ui64))
  var i8 int8
  fmt.Printf("Size of int8 is %d\n", unsafe.Sizeof(i8))
  var i16 int16
  fmt.Printf("Size of int16 is %d\n", unsafe.Sizeof(i16))
  var i32 int32
  fmt.Printf("Size of int32 is %d\n", unsafe.Sizeof(i32))
  var i64 int64
  fmt.Printf("Size of int64 is %d\n", unsafe.Sizeof(i64))
  var f32 float32
  fmt.Printf("Size of float32 is %d\n", unsafe.Sizeof(f32))
  var f64 float64
  fmt.Printf("Size of float64 is %d\n", unsafe.Sizeof(f64))
  var c64 complex64
  fmt.Printf("Size of complex64 is %d\n", unsafe.Sizeof(c64))
  var c128 complex128
  fmt.Printf("Size of complex128 is %d\n", unsafe.Sizeof(c128))
  var s string
  fmt.Printf("Size of string is %d\n", unsafe.Sizeof(s))
  var e error
  fmt.Printf("Size of error is %d\n", unsafe.Sizeof(e))
}
```

The output should be the same as the following:

```
Size of bool is 1
Size of uint8 is 1
Size of uint16 is 2
Size of uint32 is 4
Size of uint64 is 8
Size of int8 is 1
Size of int16 is 2
Size of int32 is 4
Size of int64 is 8
Size of float32 is 4
Size of float64 is 8
Size of complex64 is 8
Size of complex128 is 16
Size of string is 16
Size of error is 16
```

[1] https://golang.org/ref/spec#Types (The Go Programming Language Specification/Types)

[2] https://golang.org/ref/spec#Errors (The Go Programming Language Specification/Errors)

Composite Types

In addition to the primitive (or predeclared) types specified above, Go supports eight composite types — array, slice, struct, pointer, function, interface, map, and channel types [1].

- 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. An array's length is part of its type, so arrays cannot be resized. Array is a Mapping in Go from the integer indices to its values. Below is an example of array from the official tutorial guide of Go [2]:

  ```
  // To compile and run: go run arrays.go
  package main
  import "fmt"
  ```

```
func main() {
  var a [2]string
  a[0] = "Hello"
  a[1] = "World"
  fmt.Println(a[0], a[1])
  fmt.Println(a)
  primes := [6]int{2, 3, 5, 7, 11, 13}
  fmt.Println(primes)
}
```

The output should be the same as the following:

```
Hello World
[Hello World]
[2 3 5 7 11 13]
```

- A slice is a descriptor for a contiguous segment of an underlying array and provides access to 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`. Unlike arrays, slices are dynamically sized, and thus are much more common than arrays in practice. Slices are like references to arrays, so changing the elements of a slice modifies the corresponding elements of its underlying array. Below is an example of slice from the official tutorial guide of Go [3]:

```
// To compile and run: go run slices.go
package main
import "fmt"
func main() {
  names := []string{ "John", "Paul", "George", "Ringo", }
  fmt.Println(names)
  a := names[0:2]
  b := names[1:3]
  fmt.Println(a, b)
  b[0] = "XXX"
  fmt.Println(a, b)
  fmt.Println(names)
}
```

The output should be the same as the following:

```
[John Paul George Ringo]
[John Paul] [Paul George]
[John XXX] [XXX George]
[John XXX George Ringo]
```

- A struct is a sequence of named elements, called fields, each of which has a name and a type. Struct fields are accessed using a dot. Struct is a Cartesian Product in Go. Below is an example of struct from the official tutorial guide of Go [4]:

```
// To compile and run: go run structs.go
package main
import "fmt"
type Vertex struct {
  X int
  Y int
}
func main() {
  v := Vertex{1, 2}
  fmt.Println(v)
  v.X = 4
  fmt.Println(v.X)
}
```

The output should be the same as the following:

```
```

```
{1 2}
4
```

- 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`. The `&` operator generates a pointer to its operand. The `*` operator denotes the pointer's underlying value. Below is an example of pointer from the official tutorial guide of Go [5]:

```
// To compile and run: go run pointers.go
package main
import "fmt"
func main() {
  i, j := 42, 2701
  p := &i
  fmt.Println(*p)
  *p = 21
  fmt.Println(i)
  p = &j
  *p = *p / 37
  fmt.Println(j)
}
```

  The output should be the same as the following:

```
42
21
73
```

- 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`. Function is a Mapping in Go. Below is an example of function from the official tutorial guide of Go [6]:

```
// To compile and run: go run functions.go
package main
import "fmt"
func add(x int, y int) int { return x + y }
func main() {
  fmt.Println(add(42, 13))
}
```

  The output should be the same as the following:

```
55
```

- 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`. Below is an example of interface from the official tutorial guide of Go [7]:

```
// To compile and run: go run interfaces.go
package main
import "fmt"
type I interface {
  M()
}
type T struct {
  S string
}
func (t T) M() { fmt.Println(t.S) }
func main() {
  var i I = T{"hello"}
  i.M()
}
```

The output should be the same as the following:

```
hello
```

- 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`. Map in Go is similar to dictionary in Python. Below is an example of map from the official tutorial guide of Go [8]:

```
// To compile and run: go run maps.go
package main
import "fmt"
type Vertex struct {
  Lat, Long float64
}
var m map[string]Vertex
func main() {
  m = make(map[string]Vertex)
  m["Bell Labs"] = Vertex{ 40.68433, -74.39967, }
  fmt.Println(m["Bell Labs"])
}
```

The output should be the same as the following:

```
{40.68433 -74.39967}
```

- A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. The value of an uninitialized channel is `nil`. Below is an example of channel from the official tutorial guide of Go [9]:

```
// To compile and run: go run channels.go
package main
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)
}
```

The output should be the same as the following:

```
-5 17 12
```


[1] https://golang.org/ref/spec#Types (The Go Programming Language Specification/Types)

[2] https://tour.golang.org/moretypes/6 (A Tour of Go/Arrays)

[3] https://tour.golang.org/moretypes/8 (A Tour of Go/Slices)

[4] https://tour.golang.org/moretypes/3 (A Tour of Go/Structs)

[5] https://tour.golang.org/moretypes/1 (A Tour of Go/Pointers)

[6] https://tour.golang.org/basics/4 (A Tour of Go/Functions)

[7] https://tour.golang.org/methods/10 (A Tour of Go/Interfaces)

[8] https://tour.golang.org/moretypes/19 (A Tour of Go/Maps)

[9] https://tour.golang.org/concurrency/2 (A Tour of Go/Channels)

Statically Typed

As stated in the GoLang official documentation [1], Go is a statically typed, compiled language. Here is an example showing that Go does type check at compile time instead of run time:

```
// To compile and run: go run staticTyping.go
package main
func main() {
  x := 0
  if x > 0 {
    // This line produces an error even if it will not actually be executed
    x = "Hello world"
  }
}
```

The output should be the same as the following:

```
# command-line-arguments
./staticTyping.go:13:7: cannot use "Hello world" (type string) as type int in assignment
```

From the example above, we can see that variable holds the type in Go, and that types are checked at compile time no matter if the code is actually executed or not.

[1] https://golang.org/doc/ (Documentation - The Go Programming Language)

Implicitly Typed

Go supports the short assignment statement `:=`, where variable type is omitted while declaring a variable, and type is implicitly inferred. Here is an example from the official tutorial guide [1]:

```
// To compile and run: go run implicitlyTyped.go
package main
import "fmt"
func main() {
  var i, j int = 1, 2
  k := 3
  c, python, java := true, false, "no!"
  fmt.Println(i, j, k, c, python, java)
}
```

The output should be the same as the following:

```
1 2 3 true false no!
```

[1] https://tour.golang.org/basics/10 (A Tour of Go/Short variable declarations)

Strongly Typed

As stated in the Go Programming Language Specification [1], Go is strongly typed. Generally it means no implicit coersion will be allowed. Here is an example showing some failures of implicit coersion:

```
// To compile and run: go run stronglyTyped.go
package main
func main() {
  var i int = 10
  // Trying implicit coerse from int to float (FAILED)
  var f float32 = i
  // Trying implicit coerse from int to string (FAILED)
  var s string = i
  var b bool = true
```

```
  // Trying implicit coerse from bool to int (FAILED)
  i = b
}
```

The output should be the same as the following:

```
# command-line-arguments
./stronglyTyped.go:9:6: cannot use i (type int) as type float32 in assignment
./stronglyTyped.go:12:6: cannot use i (type int) as type string in assignment
./stronglyTyped.go:16:4: cannot use b (type bool) as type int in assignment
```

[1] https://golang.org/ref/spec#Introduction (The Go Programming Language Specification/Intro)

Statically (Lexically) Scoped

As stated in the Go Programming Language Specification [1], Go is lexically scoped using blocks. Here is an example showing its static scope:

```
// To compile and run: go run staticScope.go
package main
import "fmt"
var a = 0
func foo() {
  fmt.Println("In foo, a is", a)
  if a == 0 {
    fmt.Println("Go is statically scoped")
  } else {
    fmt.Println("Go is dynamically scoped")
  }
}
func main() {
  var a = 1
  fmt.Println("In main, a is", a)
  foo()
}
```

The output should be the same as the following:

```
In main, a is 1
In foo, a is 0
Go is statically scoped
```

[1] https://golang.org/ref/spec#Declarations_and_scope (The Go Programming Language Specification/Declarations and scope)

Operations on Numbers

As stated in the Go Programming Language Specification [1], 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. The bitwise logical and shift 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

Here is an example showing some non sense operations:

```
// To compile and run: go run mathOps.go
package main
import "fmt"
func main() {
  i := 1
  j := 2
  c := 'c'
  d := 'd'
  fmt.Println("1 / 2 =", i/j)
  fmt.Println("'c' / 2 =", c/2)
  // fmt.Println("'c' / 2 =", c/j)  // Does not compile
  fmt.Println("'c' / 'd' =", c/d)
}
```

The output should be the same as the following:

```
1 / 2 = 0
'c' / 2 = 49
'c' / 'd' = 0
```

It turns out that Go treated characters as its ASCII value when doing non sense operations such as division, and that character divided by integer literal is allowed, but character divided by integer variable is not.

[1] https://golang.org/ref/spec#Arithmetic_operators (The Go Programming Language Specification/Arithmetic Operators)

String Type and Operations on Strings

As stated in the Go Programming Language Specification [1], string is a primitive (defined) type in Go. As mentioned in the primitive types section earlier, string in Go is a sequence of bytes (thus not null-terminated), and is immutable. The length of a string `s` (its size in bytes) can be discovered using the built-in function `len`. The length is a compile-time constant if the string is a constant. A string's bytes can be accessed by integer indices 0 through `len(s)-1`. It is illegal to take the address of such an element; if `s[i]` is the `i`'th byte of a string, `&s[i]` is invalid. Other methods and operations for strings is provided in the documentation of the strings package [2]. Here is an example showing math operations on strings:

```
// To compile and run: go run stringOps.go
package main
import "fmt"
func main() {
  s1 := "hello"
  s2 := "world"
  fmt.Println("hello + world is", s1 + s2)
  // The following two lines do not compile
  //fmt.Println("hello x 2 is", s1 * 2)
  //fmt.Println("hello x world is", s1 * s2)
}
```

The output should be the same as the following:

```
hello + world is helloworld
```

It turns out that only the `+` operator is allowed on strings.

[1] https://golang.org/ref/spec#String_types (The Go Programming Language Specification/String Types)

[2] https://golang.org/pkg/strings/ (strings - The Go Programming Language)

Operator Precedence and Associativity

As stated in the Go Programming Language Specification [1], 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`. Here is an example showing the precedence and associativity of operators:

```
// To compile and run: go run opPrecedence.go
package main
import "fmt"
func main() {
  fmt.Println("1 + 2 * 3 is", 1 + 2 * 3)
  fmt.Println("4 / 2 * 3 is", 4 / 2 * 3)
  fmt.Println("1 + 2 >= 3 is", 1 + 2 >= 3)
}
```

The output should be the same as the following:

```
1 + 2 * 3 is 7
4 / 2 * 3 is 6
1 + 2 >= 3 is true
```

[1] https://golang.org/ref/spec#Operator_precedence (The Go Programming Language Specification/Operator Precedence)

Fix Operators (No Prefix)

As stated in the official FAQ site of Go [1], the ++ and -- operators form statements instead of expressions. Only postfix increment/decrement operators are allowed, no prefix. Here is an example showing the above concepts:

```
// To compile and run: go run fixOps.go
package main
import "fmt"
func main() {
  i := 0;
  j := 1;
  fmt.Println(i + j)  // inflx binary operator
  i++
  fmt.Println(i)  // postfix unary operator
  //++i  // prefix not allowed
  //fmt.Println(i--)  // Error: expecting expression
}
```

The output should be the same as the following:

```
1
1
```

[1] https://golang.org/doc/faq#inc_dec (The Go Programming Language FAQ — Increment & Decrement)

No Overloading of Methods or Operators

As stated in the official FAQ site of Go [1], Go does not support overloading of methods or operators at all. Here is an example showing failure of method overloading:

```
// To compile and run: go run overload.go
package main
func foo() int { return 0 }
func foo(x int) int { return x }
```

The output should be the same as the following:

```
# command-line-arguments
./overload.go:9:17: foo redeclared in this block
        previous declaration at ./overload.go:5:12
```

[1] https://golang.org/doc/faq#overloading (The Go Programming Language FAQ — Overloading)

Parameter Passing — Pass by Value

As stated in the official FAQ site of Go [1], everything in Go is passed by value. A function always gets a copy of the thing being passed, as if there were an assignment statement assigning the value to the parameter. For instance, passing an `int` value to a function makes a copy of the `int`, and passing a pointer value makes a copy of the pointer, but not the data it points to. Thus, pointers can be used to fake pass by reference in Go. Here is an example showing pass by value and fake pass by reference by using pointer:

```
// To compile and run: go run passByValue.go
package main
import "fmt"
func foo(x int) {
  x = x + 1
  fmt.Println("x in foo is", x)
}
func bar(x *int) { *x = *x + 1 }
func main() {
  x := 0
  foo(x)
  fmt.Println("x in main is", x)
  if x == 0 {
    fmt.Println("Go uses pass by value")
  } else {
    fmt.Println("Go uses pass by reference")
  }
  fmt.Println("Fake pass by reference by using pointer")
  bar(&x)
  fmt.Println("x in main is now", x)
}
```

The output should be the same as the following:

```
x in foo is 1
x in main is 0
Go uses pass by value
Fake pass by reference by using pointer
x in main is now 1
```

[1] https://golang.org/doc/faq#pass_by_value (The Go Programming Language FAQ — Pass by Value)

Function Parameters Evaluation Order — Left to Right

As stated in the Go Programming Language Specification [1], at package level, initialization dependencies determine the evaluation order of individual initialization expressions in variable declarations. Otherwise, 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. Here is an example showing the evaluation order:

```
// To compile and run: go run evalOrder.go
package main
import "fmt"
func foo(i int, j int, k int) { fmt.Println("foo is called") }
func arg1() int {
```

```
  fmt.Println("arg1 is called")
  return 1
}
func arg2() int {
  fmt.Println("arg2 is called")
  return 2
}
func main() {
  foo(arg1(), arg1(), arg2())
}
```

The output should be the same as the following:

```
arg1 is called
arg1 is called
arg2 is called
foo is called
```

[1] https://golang.org/ref/spec#Order_of_evaluation (The Go Programming Language Specification/Order of Evaluation)

Short Circuit Evaluation

As stated in the Go Programming Language Specification [1], logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally. This implies that Go uses short circuit evaluation. Here is an example showing short circuit evaluation:

```
// To compile and run: go run shortCircuit.go
package main
import "fmt"
func exp1() bool {
  fmt.Print("exp1 is called  ")
  return true
}
func exp2() bool {
  fmt.Print("exp2 is called  ")
  return false
}
func main() {
  fmt.Println("exp1 (true) && exp2 (false)")
  fmt.Println(exp1() && exp2())
  fmt.Println("exp2 (false) && exp1 (true)")
  fmt.Println(exp2() && exp1())
  fmt.Println("exp1 (true) || exp2 (false)")
  fmt.Println(exp1() || exp2())
  fmt.Println("exp2 (false) || exp1 (true)")
  fmt.Println(exp2() || exp1())
}
```

The output should be the same as the following:

```
exp1 (true) && exp2 (false)
exp1 is called  exp2 is called  false
exp2 (false) && exp1 (true)
exp2 is called  false
exp1 (true) || exp2 (false)
exp1 is called  true
exp2 (false) || exp1 (true)
exp2 is called  exp1 is called  true
```

[1] https://golang.org/ref/spec#Logical_operators (The Go Programming Language Specification/Logical Operators)

Tokens

As stated in the Go Programming Language Specification [1], Go has four classes of tokens — identifiers, keywords, operators and punctuation, 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.

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 } .
```

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 punctuation

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

Literals

```
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 } .

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

imaginary_lit = (decimals | float_lit) "i" .

rune_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" | `\` | "'" | `"` ) .

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

[1] https://golang.org/ref/spec#Tokens (The Go Programming Language Specification/Tokens)

Comments

As stated in the Go Programming Language Specification [1], there are two forms of comments in Go:

- Line comments start with the character sequence `//` and stop at the end of the line.
- General comments start with the character sequence `/*` and stop with the first subsequent character sequence **`*/`.

Here is an example of the two forms of comments:

```
// This is a one-line comment.
/* This is a
```

```
    multi-line comment */
```

[1] https://golang.org/ref/spec#Comments (The Go Programming Language Specification/Comments)

Multi-dimensional Array

Multi-dimensional array is supported by Go. Below is an example of 2D array from the official tutorial guide of Go [1]:

```
// To compile and run: go run multiDmArray.go
package main
import "fmt"
func main() {
  // Create a tic-tac-toe board
  board := [][]string{
    []string{"_", "_", "_"},
    []string{"_", "_", "_"},
    []string{"_", "_", "_"},
  }
  fmt.Println(board)
}
```

The output should be the same as the following:

```
[[_ _ _] [_ _ _] [_ _ _]]
```

[1] https://tour.golang.org/moretypes/14 (A Tour of Go/Slices of Slices)

No Dangling Else

Dangling else is not a problem in Go, since Go requires curly braces that wrap around each `if/else` block. As stated in the Go Programming Language Specification [1], each `if` statement is considered to be in its own implicit block. Here is an example showing how Go handles the ambiguous `if/else` blocks in an unambiguous way using curly braces:

```
// To compile and run: go run danglingElse.go
package main
import "fmt"
func main() {
  x := 0
  y := 0
  // 1
  if x == 0 { if y == 1 { fmt.Println("y1") } } else { fmt.Println("n1") }
  // 2
  if x == 0 { if y == 1 { fmt.Println("y2") } else { fmt.Println("n2") } }
}
```

The output should be the same as the following:

```
n2
```

[1] https://golang.org/ref/spec#Blocks (The Go Programming Language Specification/Blocks)

No Exceptions

As stated in the official FAQ site of Go [1], Go does not have exceptions, but does error handling in a different way. For plain error handling, Go's multi-value returns make it easy to report an error without overloading the return value. Go code uses `error` values to indicate an abnormal state. Below is an example of error handling for file opening from the official Go Blog [2]:

```
// To compile and run: go run exceptions.go
package main
import (
  "log"
```

```
  "os"
)
func main() {
  f, err := os.Open("filename.ext")
  if err != nil { log.Fatal(err) }
  f.Close()
}
```

The output should be the similar to the following:

```
2018/05/05 18:15:07 open filename.ext: no such file or directory
exit status 1
```

[1] https://golang.org/doc/faq#exceptions (The Go Programming Language FAQ — Exceptions)

[2] https://blog.golang.org/error-handling-and-go (The Go Blog — Error Handling and Go)

Anonymous, First Class & Higher Order Functions

As stated in the Go Official Documentation CodeWalk [1], Go supports first class functions, higher-order functions, user-defined function types, function literals, closures, and multiple return values. This rich feature set supports a functional programming style in a strongly typed language. Here is an example showing anonymous, first class, and higher order functions in Go:

```
// To compile and run: go run anonymousFunc.go
package main
import "fmt"
type fn func(string)
func foo(f fn) string {
  f("Higher order function")
}
func main() {
  // Anonymous function
  func(m string) { fmt.Println(m) }("Anonymous function")
  // First class function
  f := func(m string) { fmt.Println(m) }
  f("First class function")
  // Higher order function
  foo(f)
}
```

The output should be the same as the following:

```
Anonymous function
First class function
Higher order function
```

[1] https://golang.org/doc/codewalk/functions/ (CodeWalk: First Class Functions in Go)

Garbage Collection

Go has a garbage collector. As stated in the official FAQ site of Go [1], concurrency is built into Go, and automatic garbage collection makes concurrent code far easier to write. Implementing garbage collection in a concurrent environment is itself a challenge, but meeting it once rather than in every program helps everyone. The current GC implementation is a parallel mark-and-sweep collector. The garbage collector can be triggered manually, but it blocks the caller (or even the entire program) until the garbage collection is complete [2]. Here is an example of triggering GC using the runtime builtin library:

```
// To compile and run: go run garbageCollector.go
package main
import (
  "fmt"
  "runtime"
)
func main() {
  fmt.Println("Start garbage collect")
  runtime.GC()
```

```
  fmt.Println("Garbage collect finished")
}
```

The output should be the same as the following:

```
Start garbage collect
Garbage collect finished
```

[1] https://golang.org/doc/faq#garbage_collection (The Go Programming Language FAQ — Garbage Collection)

[2] https://golang.org/pkg/runtime/#GC (runtime/GC — The Go Programming Language)

Concurrent Programming

As stated in the Go Programming Language Specification [1], Go has explicit support for concurrent programming. A "go" statement starts the execution of a function call as an independent concurrent thread of control, or goroutine, within the same address space [2]. A channel provides a mechanism for the goroutines to communicate by sending and receiving values of a specified element type [3]. Here is an example showing how to use channels and coroutines to execute multiple tasks concurrently:

```
// To compile and run: go run concurrency.go
package main
import (
  "fmt"
  "math/rand"
  "strconv"
  "time"
)
func sleep(i int, chIsFinished chan string) {
  t := time.Duration(rand.Float32() * 10)
  fmt.Printf("Goroutine #%d will sleep for %d seconds\n", i, t)
  time.Sleep(t * 1000000000)
  chIsFinished <- "Goroutine #" + strconv.Itoa(i) + " finished"
}
func main() {
  x := 5
  chIsFinished := make(chan string)
  for i := 0; i < x; i++ {
    go sleep(i, chIsFinished)
  }
  for j := 0; j < x; {
    c := <-chIsFinished
    fmt.Println(c)
    j++
  }
  fmt.Println("Done")
}
```

The output should be similar to the following:

```
Goroutine #4 will sleep for 9 seconds
Goroutine #3 will sleep for 4 seconds
Goroutine #1 will sleep for 6 seconds
Goroutine #2 will sleep for 6 seconds
Goroutine #0 will sleep for 4 seconds
Goroutine #3 finished
Goroutine #0 finished
Goroutine #1 finished
Goroutine #2 finished
Goroutine #4 finished
Done
```

[1] https://golang.org/ref/spec#Introduction (The Go Programming Language Specification/Intro)

[2] https://golang.org/ref/spec#Go_statements (The Go Programming Language Specification/Go Statements)

[3] https://golang.org/ref/spec#Channel_types (The Go Programming Language Specification/Channel Types)