# CS335 Milestone 1

Adit Khokar, Nibir Baruah, Manish, Manish Mayank

# GO Language Manual

## Preface

This is a reference manual for the Go programming language. Go is a general-purpose language designed with systems programming in mind. It is strongly typed and garbage-collected and has explicit support for concurrent programming.

## Blocks

A *block* is a possibly empty sequence of declarations and statements within matching brace brackets.

In addition to explicit blocks in the source code, there are implicit blocks:

1. The *universe block* encompasses all Go source text.
2. Each package has a *package block* containing all Go source text for that package.
3. Each file has a *file block* containing all Go source text in that file.
4. Each "if", "for", and "switch" statement is considered to be in its own implicit block.
5. Each clause in a "switch" or "select" statement acts as an implicit block.

Blocks nest and influence scoping.

## Lexical elements

### Comments

Comments serve as program documentation. There are two forms:

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

A comment cannot start inside a string literal, or inside a comment.

### Semicolons

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

When the input is broken into tokens, a semicolon is automatically inserted into the token stream immediately after a line's final token if that token is

- an identifier
- an integer, floating-point, imaginary, or string literal
- one of the keywords break, continue, or return

one of the operators and punctuation ++, --, ), ], or }

To allow complex statements to occupy a single line, a semicolon may be omitted before a closing ")" or "}".

## Identifiers

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

Some identifiers are predeclared.

## Keywords

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

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

## Operators and punctuation

The following character sequences represent operators (including assignment operators) and punctuation:

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

## Integer literals

An integer literal is a sequence of digits representing an integer constant. An optional prefix sets a non-decimal base: `0b` or `0B` for binary, `0`, `0o`, or `0O` for octal, and `0x` or `0X` for hexadecimal. A single `0` is considered a decimal zero.

## Floating-point literals

A floating-point literal is a decimal or hexadecimal representation of a floating-point constant.

A decimal floating-point literal consists of an integer part (decimal digits), a decimal point, a fractional part (decimal digits), and an exponent part (e or E followed by an optional sign and decimal digits). One of the integer part or the fractional part may be elided; one of the decimal point or the exponent part may be elided. An exponent value exp scales the mantissa (integer and fractional part) by $10^{exp}$.

A hexadecimal floating-point literal consists of a `0x` or `0X` prefix, an integer part (hexadecimal digits), a radix point, a fractional part (hexadecimal digits), and an exponent part (p or P followed by an optional sign and decimal digits). One of the integer part or the fractional part may be elided; the radix point may be elided as well, but the exponent part is required. (This syntax matches the one given in IEEE 754-2008 §5.12.3.) An exponent value exp scales the mantissa (integer and fractional part) by $2^{exp}$.

## Imaginary literals

An imaginary literal represents the imaginary part of a complex constant. It consists of an integer or floating-point literal followed by the lower-case letter i. The value of an imaginary literal is the value of the respective integer or floating-point literal multiplied by the imaginary unit *i*.

## String literals

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

## Constants

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

A constant may be given a type explicitly by a constant declaration or conversion, or implicitly when used in a variable declaration or an assignment or as an operand in an expression. It is an error if the constant value cannot be represented as a value of the respective type.

## Variables

A variable is a storage location for holding a *value*. The set of permissible values is determined by the variable's *type*.

A variable declaration or, for function parameters and results, the signature of a function declaration or function literal reserves storage for a named variable. Calling the built-in function new or taking the address of a composite literal allocates storage for a variable at run time. Such an anonymous variable is referred to via a (possibly implicit) pointer indirection.

*Structured* variables of array, slice, and struct types have elements and fields that may be addressed individually. Each such element acts like a variable.

A variable's value is retrieved by referring to the variable in an expression; it is the most recent value assigned to the variable. If a variable has not yet been assigned a value, its value is the zero value for its type.

## Types

A type determines a set of values together with operations and methods specific to those values. A type may be denoted by a *type name*, if it has one, or specified using a *type literal*, which composes a type from existing types.

The language predeclares certain type names. Others are introduced with type declarations. *Composite types*—array, struct, pointer, function and slice types—may be constructed using type literals.

## Boolean types

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

## Numeric types

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

```
uint8       the set of all unsigned  8-bit integers (0 to 255)
```

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

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

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

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

byte        alias for uint8
```

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

## String types

A *string type* represents the set of string values. A string value is a (possibly empty) sequence of bytes. The number of bytes is called the length of the string and is never negative. Strings are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is string; it is a defined type.

The length of a string s 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.

## Array types

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

The length is part of the array's type; it must evaluate to a non-negative constant representable by a value of type int. The length of array a can be discovered using the built-in function len. The elements can be addressed by integer indices 0 through len(a)-1. Array types are always one-dimensional but may be composed to form multi-dimensional types.

### Slice types

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 number of elements is called the length of the slice and is never negative. The value of an uninitialized slice is nil.

### Struct types

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

### Pointer types

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

## Functions

Functions can have multiple return values. For multiple return values, multiple variables need to be placed on the left hand side of assignment separated by comma. To ignore a return value, '_' is used.

Multiple parameters are separated by comma. If there are multiple input parameters in succession with the same data type, the data type only needs to be stated with the last parameter, with previous parameters being assigned that data type as well.

Functions themselves are treated like values, they can be passed around or assigned to variables.

## Declarations and scope

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.

The blank identifier may be used like any other identifier in a declaration, but it does not introduce a binding and thus is not declared. In the package block, the identifier init may only

be used for init function declarations, and like the blank identifier it does not introduce a new binding.

The scope of a declared identifier is the extent of source text in which the identifier denotes the specified constant, type, variable, function, label, or package.

Go is lexically scoped using blocks:

1. The scope of a predeclared identifier is the universe block.
2. The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at top level (outside any function) is the package block.
3. The scope of the package name of an imported package is the file block of the file containing the import declaration.
4. The scope of an identifier denoting a method receiver, function parameter, or result variable is the function body.
5. The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec (ShortVarDecl for short variable declarations) and ends at the end of the innermost containing block.
6. The scope of a type identifier declared inside a function begins at the identifier in the TypeSpec and ends at the end of the innermost containing block.

## Predeclared identifiers

The following identifiers are implicitly declared in the universe block:

**Types:** `bool byte complex64 complex128 error float32 float64int int8 int16 int32 int64 rune string uint uint8 uint16 uint32 uint64 uintptr`
**Constants:** `true false iota`
**Zero value:** `nil`
**Functions:** `append cap close complex copy delete imag lenmake new panic print println real recover`

## Uniqueness of identifiers

Given a set of identifiers, an identifier is called unique if it is different from every other in the set. Two identifiers are different if they are spelled differently, or if they appear in different packages and are not exported. Otherwise, they are the same.

## Type definitions

A type definition creates a new, distinct type with the same underlying type and operations as the given type, and binds an identifier to it.

The new type is called a defined type. It is different from any other type, including the type it is created from.

A defined type may have methods associated with it. It does not inherit any methods bound to the given type, but the method set of an interface type or of elements of a composite type remains unchanged:

### Variable declarations

A variable declaration creates one or more variables, binds corresponding identifiers to them, and gives each a type and an initial value.

If a list of expressions is given, the variables are initialized with the expressions following the rules for assignments. Otherwise, each variable is initialized to its zero value.

If a type is present, each variable is given that type. Otherwise, each variable is given the type of the corresponding initialization value in the assignment.

### Function declarations

A function declaration binds an identifier, the function name, to a function.

If the function's signature declares result parameters, the function body's statement list must end in a terminating statement.

A function declaration may omit the body. Such a declaration provides the signature for a function implemented outside Go, such as an assembly routine.

## Expressions

An expression specifies the computation of a value by applying operators and functions to operands.

### Operands

Operands denote the elementary values in an expression. An operand may be a literal, a (possibly qualified) non-blank identifier denoting a constant, variable, or function, or a parenthesized expression.

### Operators

Operators combine operands into expressions.

For most binary operators, the operand types must be identical unless the operation involves shifts or untyped constants. For operations involving constants only, see the section on constant expressions.

Except for shift operations, if one operand is an untyped constant and the other operand is not, the constant is implicitly converted to the type of the other operand.

## Control Structures

### If statements

"`If`" statements specify the conditional execution of two branches according to the value of a boolean expression. If the expression evaluates to true, the "`if`" branch is executed, otherwise, if present, the "`else`" branch is executed.

### For statements

A "`for`" statement specifies repeated execution of a block. There are three forms: The iteration may be controlled by a single condition, a "`for`" clause, or a "`range`" clause.

### Input Statements

The Scan function is part of "fmt" package. fmt also comes with Scanf (for specifying string formatting) and Scanln (for scanning until the end of the line). The Scan functions store to a pointer variable.

The scan function itself returns the number of successfully scanned items and if necessary, an error (in that order). It is good practice to error check when using the Scan function.

The Scan functions are used for splitting space-delimited tokens, whereas the reader is used to read full lines.

### Syntax

```
//stores space separated values into successive arguments
var storageVariable variableTypefmt.Scan(&storageVariable) //assuming fmt is imported
//reads line all in one go
reader := bufio.NewReader(os.Stdin) //create new reader, assuming bufio imported
var storageString stringstorageString, _ := reader.ReadString('\n')
```

### Output Statements

The 'fmt' package must be imported to print. Aside from PrintLn, other C-style prints like Printf can be used. Variable Specifier is used to output a variable in its place, indicated by %. The default variable specifier is '%v', which is a catch-all for most variable types.

Specifiers depend on variable type: %d (decimal), %f (floating-point), %e or %E (floating-point in scientific notation), %g or %G (uses the shorter version of %f or %e / %E), %s (strings).

### Syntax

```
fmt.Println("Message") //assuming fmt is imported
//variables
fmt.Println("Text variableSpecifier", variableIdentifier)
```

## Statements

A statement is a syntactic unit that expresses some action to be carried out.

### Continue statements

A "continue" statement begins the next iteration of the innermost "for" loop at its post statement. The "for" loop must be within the same function. If there is a label, it must be that of an enclosing "for" statement, and that is the one whose execution advances.

### Return statements

A "return" statement in a function F terminates the execution of F, and optionally provides one or more result values. Any functions deferred by F are executed before F returns to its caller.

# Advanced Features

## Global Variables

Global variables are defined outside of a function, usually on top of the program. Global variables hold their value throughout the lifetime of the program and they can be accessed inside any of the functions defined for the program. A global variable can be accessed by any function. That is, a global variable is available for use throughout the program after its declaration. The following example uses both global and local variables −

## Multilevel Pointers

A pointer is pointer to another pointer which can be pointer to others pointers and so on is known as multilevel pointers. We can have any level of pointers.

## Higher Order Functions

If a function is passed as an argument to another function, then such types of functions are known as a Higher-Order function. This passing function as an argument is also known as a callback function or first-class function.

## Short Variable Declaration

It is shorthand for a regular variable declaration with initializer expressions but no types:

```
i, j := 0, 10
r, w, _ := os.Pipe()  // os.Pipe() returns a connected pair of Files and an
error, if any
```

Unlike regular variable declarations, a short variable declaration may *redeclare* variables provided they were originally declared earlier in the same block (or the parameter lists if the block is the function body) with the same type, and at least one of the non-blank variables is new. As a consequence, redeclaration can only appear in a multi-variable short declaration. Redeclaration does not introduce a new variable; it just assigns a new value to the original.

Short variable declarations may appear only inside functions. In some contexts such as the initializers for if, for, or switch statements, they can be used to declare local temporary variables.

# Optimizations

## Constant Folding

This is an optimization technique which eliminates expressions that calculate a value that can be determined before code execution.

If operands are known at compile time, then the compiler performs the operations statically.

An Example,

```
int x = (2 + 3) * y → int x = 5 * y
int z = 300 * 78 * 6 → int z = 140400
```

Compilers will not go ahead and generate two symbols addition and multiplication in the first case or two multiplication symbols in the second case but instead will identify such constructs and substitute the computed values as shown.

## Short Circuit Evaluation

Short circuit evaluation is the semantics of some Boolean operators in which the second argument is executed or evaluated only if the first argument does not suffice to determine the value of the expression: when the first argument of the AND function evaluates to false, the overall value must be false; and when the first argument of the OR function evaluates to true, the overall value must be true.

# External Libraries

## Math

Package math provides basic constants like $\pi, e, \ln 10$ and mathematical functions such as the various trigonometric and logarithmic functions and other functions such as `absolute`, `floor` etc.

## String

Package strings implements simple functions to manipulate UTF-8 encoded strings. Some of the functions implemented are `Compare`, `HasPrefix`, `Trim`, `Replace`, `Join`.