The Go Programming Language

The Go Programming Language Specification

Version of July 14, 2010

Introduction

This is a reference manual for the Go programming language. For more information and other documents, see http://golang.org.

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. Programs are constructed from *packages*, whose properties allow efficient management of dependencies. The existing implementations use a traditional compile/link model to generate executable binaries.

The grammar is compact and regular, allowing for easy analysis by automatic tools such as integrated development environments.

Notation

The syntax is specified using Extended Backus-Naur Form (EBNF):

```
Production  = production_name "=" Expression "." .
Expression  = Alternative { "|" Alternative } .
Alternative = Term { Term } .
Term        = production_name | token [ "..." token ] | G
roup | Option | Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .
```

Productions are expressions constructed from terms and the following operators, in increasing precedence:

```
|   alternation
()  grouping
[]  option (0 or 1 times)
{}  repetition (0 to n times)
```

Lower-case production names are used to identify lexical tokens. Non-terminals are in CamelCase. Lexical symbols are enclosed in double quotes `""` or back quotes ``` `` ```.

The form `a ... b` represents the set of characters from `a` through `b` as alternatives.

Source code representation

Source code is Unicode text encoded in UTF-8. 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. For

simplicity, this document will use the term *character* to refer to a Unicode code point.

Each code point is distinct; for instance, upper and lower case letters are different characters.

Implementation restriction: For compatibility with other tools, a compiler may disallow the NUL character (U+0000) in the source text.

Characters

The following terms are used to denote specific Unicode character classes:

```
unicode_char   = /* an arbitrary Unicode code point */ .
unicode_letter = /* a Unicode code point classified as "L
etter" */ .
unicode_digit  = /* a Unicode code point classified as "D
igit" */ .
```

In The Unicode Standard 5.2, Section 4.5 General Category-Normative defines a set of character categories. Go treats those characters in category Lu, Ll, Lt, Lm, or Lo as Unicode letters, and those in category Nd as Unicode digits.

Letters and digits

The underscore character _ (U+005F) is considered a letter.

```
letter        = unicode_letter | "_" .
decimal_digit = "0" ... "9" .
octal_digit   = "0" ... "7" .
hex_digit     = "0" ... "9" | "A" ... "F" | "a" ... "f" .
```

Lexical elements

Comments

There are two forms of comments:

1. *Line comments* start with the character sequence `//` and continue through the next newline. A line comment acts like a newline.
2. *General comments* start with the character sequence `/*` and continue through the character sequence `*/`. A general comment that spans multiple lines acts like a newline, otherwise it acts like a space.

Comments do not nest.

Tokens

Tokens form the vocabulary of the Go language. There are four classes: *identifiers*, *keywords*, *operators and delimiters*, 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 may trigger the insertion of a semicolon. While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

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:

1. When the input is broken into tokens, a semicolon is automatically inserted into the token stream at the end of a non-blank line if the line's final token is
   - an identifier
   - an integer, floating-point, imaginary, character, or string literal
   - one of the keywords `break`, `continue`, `fallthrough`, or `return`
   - one of the operators and delimiters `++`, `--`, `)`, `]`, or `}`
2. To allow complex statements to occupy a single line, a semicolon may be omitted before a closing `")"` or `"}"`.

To reflect idiomatic use, code examples in this document elide semicolons using these rules.

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.

```
identifier = letter { letter | unicode_digit } .
```

```
a
_x9
ThisVariableIsExported
αβ
```

Some identifiers are predeclared.

Keywords

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

```
break        default      func         interface    selec
t
case         defer        go           map          struc
t
chan         else         goto         package      switc
h
const        fallthrough  if           range        type
continue     for          import       return       var
```

Operators and Delimiters

The following character sequences represent operators, delimiters, and other special tokens:

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

```
%     >>     %=     >>=     --     !      ...    .     :
      &^            &^=
```

Integer literals

An integer literal is a sequence of digits representing an integer constant. An optional prefix sets a non-decimal base: `0` for octal, `0x` or `0X` for hexadecimal. In hexadecimal literals, letters `a-f` and `A-F` represent values 10 through 15.

```
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 } .
```

```
42
0600
0xBadFace
170141183460469231731687303715884105727
```

Floating-point literals

A floating-point literal is a decimal representation of a floating-point constant. It has an integer part, a decimal point, a fractional part, and an exponent part. The integer and fractional part comprise decimal digits; the exponent part is an `e` or `E` followed by an optionally signed decimal exponent. One of the integer part or the fractional part may be elided; one of the decimal point or the exponent may be elided.

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

```
0.
72.40
072.40  // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
```

Imaginary literals

An imaginary literal is a decimal representation of the imaginary part of a complex constant. It consists of a floating-point literal or decimal integer followed by the lower-case letter `i`.

```
imaginary_lit = (decimals | float_lit) "i" .
```

```
0i
011i  // == 11i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i
```

Character literals

A character literal represents an integer constant, typically a Unicode code point, as one or more characters enclosed in single quotes. Within the quotes, any character may appear except single quote and newline. A single quoted character represents itself, while multi-character sequences beginning with a backslash encode values in various formats.

The simplest form represents the single character within the quotes; since Go source text is Unicode characters encoded in UTF-8, multiple UTF-8-encoded bytes may represent a single integer value. For instance, the literal `'a'` holds a single byte representing a literal `a`, Unicode U+0061, value `0x61`, while `'ä'` holds two bytes (`0xc3 0xa4`) representing a literal `a`-dieresis, U+00E4, value `0xe4`.

Several backslash escapes allow arbitrary values to be represented as ASCII text. There are four ways to represent the integer value as a numeric constant: `\x` followed by exactly two hexadecimal digits; `\u` followed by exactly four hexadecimal digits; `\U` followed by exactly eight hexadecimal digits, and a plain backslash `\` followed by exactly three octal digits. In each case the value of the literal is the value represented by the digits in the corresponding base.

Although these representations all result in an integer, they have different valid ranges. Octal escapes must represent a value between 0 and 255 inclusive. Hexadecimal escapes satisfy this condition by construction. The escapes `\u` and `\U` represent Unicode code points so within them some values are illegal, in particular those above `0x10FFFF` and surrogate halves.

After a backslash, certain single-character escapes represent special values:

```
\a   U+0007 alert or bell
\b   U+0008 backspace
\f   U+000C form feed
\n   U+000A line feed or newline
\r   U+000D carriage return
\t   U+0009 horizontal tab
\v   U+000b vertical tab
\\   U+005c backslash
\'   U+0027 single quote  (valid escape only within chara
cter literals)
\"   U+0022 double quote  (valid escape only within strin
g literals)
```

All other sequences starting with a backslash are illegal inside character literals.

```
char_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_digi
t .
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" | `\` | "'" | `"` ) .
```

```
'a'
'ä'
'本'
'\t'
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
```

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.

Raw string literals are character sequences between back quotes ``. Within the quotes, any character is legal except back quote. The value of a raw string literal is the string composed of the uninterpreted characters between the quotes; in particular, backslashes have no special meaning and the string may span multiple

lines.

Interpreted string literals are character sequences between double quotes "". The text between the quotes, which may not span multiple lines, forms the value of the literal, with backslash escapes interpreted as they are in character literals (except that \' is illegal and \" is legal). The three-digit octal (*nnn*) and two-digit hexadecimal (\x*nn*) escapes represent individual *bytes* of the resulting string; all other escapes represent the (possibly multi-byte) UTF-8 encoding of individual *characters*. Thus inside a string literal \377 and \xFF represent a single byte of value 0xFF=255, while ÿ, \u00FF, \U000000FF and \xc3\xbf represent the two bytes 0xc3 0xbf of the UTF-8 encoding of character U+00FF.

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

```
`abc`  // same as "abc"
`\n
\n`    // same as "\\n\n\\n"
"\n"
""
"Hello, world!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
```

These examples all represent the same string:

```
"日本語"                                 // UTF-8 input te
xt
`日本語`                                 // UTF-8 input te
xt as a raw literal
"\u65e5\u672c\u8a9e"                     // The explicit U
nicode code points
```

```
"\U000065e5\U0000672c\U00008a9e"        // The explicit U
nicode code points
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"  // The explicit U
TF-8 bytes
```

If the source code represents a character as two code points, such as a combining form involving an accent and a letter, the result will be an error if placed in a character literal (it is not a single code point), and will appear as two code points if placed in a string literal.

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 value is represented by an integer, floating-point, imaginary, character, or string literal, an identifier denoting a constant, a constant expression, or the result value of some built-in functions such as `unsafe.Sizeof` applied to any value, `cap` or `len` applied to some expressions, `real` and `imag` applied to a complex constant and `cmplx` applied to numeric constants. The boolean truth values are represented by the predeclared constants `true` and `false`. The predeclared identifier iota denotes an integer constant.

In general, complex constants are a form of constant expression and are discussed in that section.

Numeric constants represent values of arbitrary precision and do not overflow.

Constants may be typed or untyped. Literal constants, `true`, `false`, `iota`, and certain constant expressions containing only untyped constant operands are untyped.

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 accurately represented as a value of the respective type. For instance, `3.0` can be given any integer or any floating-point type, while `2147483648.0` (equal to `1<<31`) can be given the types `float32`, `float64`, or `uint32` but not `int32` or `string`.

There are no constants denoting the IEEE-754 infinity and not-a-number values, but the `math` package's Inf, NaN, IsInf, and IsNaN functions return and test for those values at run time.

Implementation restriction: A compiler may implement numeric constants by choosing an internal representation with at least twice as many bits as any machine type; for floating-point values, both the mantissa and exponent must be twice as large.

Types

A type determines the set of values and operations specific to values of that type. A type may be specified by a (possibly qualified) *type name* (§Qualified identifier, §Type declarations) or a *type literal*, which composes a new type from previously declared types.

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = QualifiedIdent.
TypeLit   = ArrayType | StructType | PointerType | Functi
onType | InterfaceType |
            SliceType | MapType | ChannelType .
```

Named instances of the boolean, numeric, and string types are predeclared. *Composite types*—array, struct, pointer, function, interface, slice, map, and channel types—may be constructed using type literals.

Each type `T` has an *underlying type*: If `T` is a predeclared type or a type literal, the corresponding underlying type is `T` itself. Otherwise, `T`'s underlying type is the underlying type of the type to which `T` refers in its type declaration.

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

The underlying type of `string`, `T1`, and `T2` is `string`. The underlying type of `[]T1`, `T3`, and `T4` is `[]T1`.

A type may have a *method set* associated with it (§Interface types, §Method declarations). The method set of an interface type is its interface. The method set of any other named type `T` consists of all methods with receiver type `T`. The method set of the corresponding pointer type `*T` is the set of all methods with receiver `*T` or `T` (that is, it also contains the method set of `T`). Any other type has an empty method set. In a method set, each method must have a unique name.

The *static type* (or just *type*) of a variable is the type defined by its declaration. Variables of interface type also have a distinct *dynamic type*, which is the actual type of the value stored in the variable at run-time. The dynamic type may vary during execution but is always assignable to the static type of the interface variable. For non-interface types, the dynamic type is always the static type.

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`.

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 t
o 127)
int16       the set of all signed 16-bit integers (-32768
 to 32767)
int32       the set of all signed 32-bit integers (-21474
83648 to 2147483647)
int64       the set of all signed 64-bit integers (-92233
72036854775808 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 r
eal and imaginary parts
complex128  the set of all complex numbers with float64 r
eal and imaginary parts

byte        familiar alias for uint8
```

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

There is also a set of predeclared numeric types with implementation-specific sizes:

```
uint     either 32 or 64 bits
int      either 32 or 64 bits
float    either 32 or 64 bits
complex  real and imaginary parts have type float
uintptr  an unsigned integer large enough to store the un
interpreted bits of a pointer value
```

To avoid portability issues all numeric types are distinct except `byte`, which is an alias for `uint8`. Conversions are required when different numeric types are mixed in an expression or assignment. For instance, `int32` and `int` are not the same type even though they may have the same size on a particular architecture.

String types

A *string type* represents the set of string values. Strings behave like arrays of bytes but are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is `string`.

The elements of strings have type `byte` and may be accessed using the usual

indexing operations. 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. The length of string `s` can be discovered using the built-in function `len`. The length is a compile-time constant if `s` is a string literal.

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 and is never negative.

```
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
```

The length is part of the array's type and must be a constant expression that evaluates to a non-negative integer value. The length of array `a` can be discovered using the built-in function `len(a)`. The elements can be indexed by integer indices 0 through the `len(a)-1` (§Indexes). Array types are always one-dimensional but may be composed to form multi-dimensional types.

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64  // same as [2]([2]([2]float64))
```

Slice types

A slice is a reference to a contiguous segment of an array and contains 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`.

```
SliceType = "[" "]" ElementType .
```

Like arrays, slices are indexable and have a length. The length of a slice `s` can be discovered by the built-in function `len(s)`; unlike with arrays it may change

during execution. The elements can be addressed by integer indices 0 through `len(s)-1` (§Indexes). The slice index of a given element may be less than the index of the same element in the underlying array.

A slice, once initialized, is always associated with an underlying array that holds its elements. A slice therefore shares storage with its array and with other slices of the same array; by contrast, distinct arrays always represent distinct storage.

The array underlying a slice may extend past the end of the slice. The *capacity* is a measure of that extent: it is the sum of the length of the slice and the length of the array beyond the slice; a slice of length up to that capacity can be created by `slicing' a new one from the original slice (§Slices). The capacity of a slice `a` can be discovered using the built-in function `cap(a)`.

A new, initialized slice value for a given element type `T` is made using the built-in function `make`, which takes a slice type and parameters specifying the length and optionally the capacity:

```
make([]T, length)
make([]T, length, capacity)
```

The `make()` call allocates a new, hidden array to which the returned slice value refers. That is, executing

```
make([]T, length, capacity)
```

produces the same slice as allocating an array and slicing it, so these two examples result in the same slice:

```
make([]int, 50, 100)
new([100]int)[0:50]
```

Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. With arrays of arrays, the inner arrays are, by construction, always the same length; however with slices of slices (or arrays of slices), the lengths may vary dynamically. Moreover, the inner slices must be allocated individually (with `make`).

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 (AnonymousField). Within a struct, non-blank field names must be unique.

```
StructType     = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl      = (IdentifierList Type | AnonymousField) [
 Tag ] .
AnonymousField = [ "*" ] TypeName .
Tag            = string_lit .
```

```
// An empty struct.
struct {}

// A struct with 6 fields.
struct {
        x, y int
        u float
        _ float  // padding
        A *[]int
        F func()
}
```

A field declared with a type but no explicit field name is an *anonymous field*. Such a field type must be specified as a type name `T` or as a pointer to a type name `*T`, and `T` itself may not be a pointer type. The unqualified type name acts as the field name.

```
// A struct with four anonymous fields of type T1, *T2, P
.T3 and *P.T4
struct {
        T1        // field name is T1
        *T2       // field name is T2
```

```
        P.T3       // field name is T3
        *P.T4      // field name is T4
        x, y int   // field names are x and y
}
```

The following declaration is illegal because field names must be unique in a struct type:

```
struct {
        T          // conflicts with anonymous field *T an
d *P.T
        *T         // conflicts with anonymous field T and
 *P.T
        *P.T       // conflicts with anonymous field T and
 *T
}
```

Fields and methods (§Method declarations) of an anonymous field are promoted to be ordinary fields and methods of the struct (§Selectors). The following rules apply for a struct type named `S` and a type named `T`:

- If `S` contains an anonymous field `T`, the method set of `S` includes the method set of `T`.
- If `S` contains an anonymous field `*T`, the method set of `S` includes the method set of `*T` (which itself includes the method set of `T`).
- If `S` contains an anonymous field `T` or `*T`, the method set of `*S` includes the method set of `*T` (which itself includes the method set of `T`).

A field declaration may be followed by an optional string literal *tag*, which becomes an attribute for all the fields in the corresponding field declaration. The tags are made visible through a reflection interface but are otherwise ignored.

```
// A struct corresponding to the TimeStamp protocol buffe
r.
// The tag strings define the protocol buffer field numbe
rs.
```

```
struct {
        microsec  uint64 "field 1"
        serverIP6 uint64 "field 2"
        process   string "field 3"
}
```

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 unitialized pointer is `nil`.

```
PointerType = "*" BaseType .
BaseType = Type .
```

```
*int
*map[string] *chan int
```

Function types

A function type denotes the set of all functions with the same parameter and result types. The value of an unitialized variable of function type is `nil`.

```
FunctionType   = "func" Signature .
Signature      = Parameters [ Result ] .
Result         = Parameters | Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
ParameterList  = ParameterDecl { "," ParameterDecl } .
ParameterDecl  = [ IdentifierList ] [ "..." ] Type .
```

Within a list of parameters or results, the names (IdentifierList) must either all be present or all be absent. If present, each name stands for one item (parameter or result) of the specified type; if absent, each type stands for one item of that type. Parameter and result lists are always parenthesized except that if there is exactly one unnamed result it may be written as an unparenthesized type.

If the function's last parameter has a type prefixed with `...`, the function may be invoked with zero or more arguments for that parameter, each of which must be assignable to the type that follows the `...`. Such a function is called *variadic*.

```
func()
func(x int)
func() int
func(prefix string, values ...int)
func(a, b int, z float) bool
func(a, b int, z float) (bool)
func(a, b int, z float, opt ...interface{}) (success bool
)
func(int, int, float) (float, *[]int)
func(n int) func(p *T)
```

Interface types

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 unitialized variable of interface type is `nil`.

```
InterfaceType      = "interface" "{" { MethodSpec ";" } "
}" .
MethodSpec         = MethodName Signature | InterfaceType
Name .
MethodName         = identifier .
InterfaceTypeName  = TypeName .
```

As with all method sets, in an interface type, each method must have a unique name.

```
// A simple File interface
interface {
        Read(b Buffer) bool
        Write(b Buffer) bool
```

```
        Close()
}
```

More than one type may implement an interface. For instance, if two types `S1` and `S2` have the method set

```
func (p T) Read(b Buffer) bool { return ... }
func (p T) Write(b Buffer) bool { return ... }
func (p T) Close() { ... }
```

(where `T` stands for either `S1` or `S2`) then the `File` interface is implemented by both `S1` and `S2`, regardless of what other methods `S1` and `S2` may have or share.

A type implements any interface comprising any subset of its methods and may therefore implement several distinct interfaces. For instance, all types implement the *empty interface*:

```
interface{}
```

Similarly, consider this interface specification, which appears within a type declaration to define an interface called `Lock`:

```
type Lock interface {
        Lock()
        Unlock()
}
```

If `S1` and `S2` also implement

```
func (p T) Lock() { ... }
func (p T) Unlock() { ... }
```

they implement the `Lock` interface as well as the `File` interface.

An interface may contain an interface type name `T` in place of a method specification. The effect is equivalent to enumerating the methods of `T` explicitly in the interface.

```
type ReadWrite interface {
        Read(b Buffer) bool
        Write(b Buffer) bool
}

type File interface {
        ReadWrite  // same as enumerating the methods in
ReadWrite
        Lock       // same as enumerating the methods in
Lock
        Close()
}
```

Map types

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`.

```
MapType     = "map" "[" KeyType "]" ElementType .
KeyType     = Type .
```

The comparison operators `==` and `!=` (§Comparison operators) must be fully defined for operands of the key type; thus the key type must not be a struct, array or slice. If the key type is an interface type, these comparison operators must be defined for the dynamic key values; failure will cause a run-time panic.

```
map [string] int
map [*T] struct { x, y float }
map [string] interface {}
```

The number of map elements is called its length. For a map `m`, it can be

discovered using the built-in function `len(m)` and may change during execution. Values may be added and removed during execution using special forms of assignment.

A new, empty map value is made using the built-in function `make`, which takes the map type and an optional capacity hint as arguments:

```
make(map[string] int)
make(map[string] int, 100)
```

The initial capacity does not bound its size: maps grow to accommodate the number of items stored in them.

Channel types

A channel provides a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type. The value of an uninitialized channel is `nil`.

```
ChannelType = ( "chan" [ "<-" ] | "<-" "chan" ) ElementType .
```

The `<-` operator specifies the channel *direction*, *send* or *receive*. If no direction is given, the channel is *bi-directional*. A channel may be constrained only to send or only to receive by conversion or assignment.

```
chan T          // can be used to send and receive values of type T
chan<- float    // can only be used to send floats
<-chan int      // can only be used to receive ints
```

The `<-` operator associates with the leftmost `chan` possible:

```
chan<- chan int     // same as chan<- (chan int)
chan<- <-chan int   // same as chan<- (<-chan int)
<-chan <-chan int   // same as <-chan (<-chan int)
```

```
chan (<-chan int)
```

A new, initialized channel value can be made using the built-in function `make`, which takes the channel type and an optional capacity as arguments:

```
make(chan int, 100)
```

The capacity, in number of elements, sets the size of the buffer in the channel. If the capacity is greater than zero, the channel is asynchronous: provided the buffer is not full, sends can succeed without blocking. If the capacity is zero or absent, the communication succeeds only when both a sender and receiver are ready.

A channel may be closed and tested for closure with the built-in functions `close` and `closed`.

Properties of types and values

Type identity

Two types are either *identical* or *different*.

Two named types are identical if their type names originate in the same type declaration. A named and an unnamed type are always different. Two unnamed types are identical if the corresponding type literals are identical, that is, if they have the same literal structure and corresponding components have identical types. In detail:

- Two array types are identical if they have identical element types and the same array length.
- Two slice types are identical if they have identical element types.
- Two struct types are identical if they have the same sequence of fields, and if corresponding fields have the same names, and identical types, and identical tags. Two anonymous fields are considered to have the same name. Lower-case field names from different packages are always different.
- Two pointer types are identical if they have identical base types.
- Two function types are identical if they have the same number of parameters

and result values, corresponding parameter and result types are identical, and either both functions are variadic or neither is. Parameter and result names are not required to match.

- Two interface types are identical if they have the same set of methods with the same names and identical function types. Lower-case method names from different packages are always different. The order of the methods is irrelevant.
- Two map types are identical if they have identical key and value types.
- Two channel types are identical if they have identical value types and the same direction.

Given the declarations

```
type (
        T0 []string
        T1 []string
        T2 struct { a, b int }
        T3 struct { a, c int }
        T4 func(int, float) *T0
        T5 func(x int, y float) *[]string
)
```

these types are identical:

```
T0 and T0
[]int and []int
struct { a, b *T5 } and struct { a, b *T5 }
func(x int, y float) *[]string and func(int, float) (resu
lt *[]string)
```

`T0` and `T1` are different because they are named types with distinct declarations; `func(int, float) *T0` and `func(x int, y float) *[]string` are different because `T0` is different from `[]string`.

Assignability

A value `x` is *assignable* to a variable of type `T` ("`x` is assignable to `T`") in any of these cases:

- `x`'s type is identical to `T`.
- `x`'s type `V` or `T` have identical underlying types and `V` or `T` is not a named type.
- `T` is an interface type and `x` implements `T`.
- `x` is a bidirectional channel value, `T` is a channel type, `x`'s type `V` and `T` have identical element types, and `V` or `T` is not a named type.
- `x` is the predeclared identifier `nil` and `T` is a pointer, function, slice, map, channel, or interface type.
- `x` is an untyped constant representable by a value of type `T`.

If `T` is a struct type, either all fields of `T` must be exported, or the assignment must be in the same package in which `T` is declared. In other words, a struct value can be assigned to a struct variable only if every field of the struct may be legally assigned individually by the program.

Any value may be assigned to the blank identifier.

Blocks

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

```
Block = "{" { Statement ";" } "}" .
```

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.

Declarations and scope

A declaration binds a non-blank identifier to a constant, type, variable, function, 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.

```
Declaration   = ConstDecl | TypeDecl | VarDecl .
TopLevelDecl  = Declaration | FunctionDecl | MethodDecl .
```

The *scope* of a declared identifier is the extent of source text in which the identifier denotes the specified constant, type, variable, function, 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 declared at top level (outside any function) is the package block.
3. The scope of an imported package identifier is the file block of the file containing the import declaration.
4. The scope of an identifier denoting a 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 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.

An identifier declared in a block may be redeclared in an inner block. While the identifier of the inner declaration is in scope, it denotes the entity declared by the

inner declaration.

The package clause is not a declaration; the package name does not appear in any scope. Its purpose is to identify the files belonging to the same package and to specify the default package name for import declarations.

Label scopes

Labels are declared by labeled statements and are used in the `break`, `continue`, and `goto` statements (§Break statements, §Continue statements, §Goto statements). In contrast to other identifiers, labels are not block scoped and do not conflict with identifiers that are not labels. The scope of a label is the body of the function in which it is declared and excludes the body of any nested function.

Predeclared identifiers

The following identifiers are implicitly declared in the universe block:

```
Basic types:
        bool byte complex64 complex128 float32 float64
        int8 int16 int32 int64 string uint8 uint16 uint32
 uint64

Architecture-specific convenience types:
        complex float int uint uintptr

Constants:
        true false iota

Zero value:
        nil

Functions:
        cap close closed cmplx copy imag len make
        new panic print println real
```

Exported identifiers

An identifier may be *exported* to permit access to it from another package using a qualified identifier. An identifier is exported if both:

1. the first character of the identifier's name is a Unicode upper case letter (Unicode class "Lu"); and
2. the identifier is declared in the package block or denotes a field or method of a type declared in that block.

All other identifiers are not exported.

Blank identifier

The *blank identifier*, represented by the underscore character _, may be used in a declaration like any other identifier but the declaration does not introduce a new binding.

Constant declarations

A constant declaration binds a list of identifiers (the names of the constants) to the values of a list of constant expressions. The number of identifiers must be equal to the number of expressions, and the *n*th identifier on the left is bound to the value of the *n*th expression on the right.

```
ConstDecl      = "const" ( ConstSpec | "(" { ConstSpec ";
" } ")" ) .
ConstSpec      = IdentifierList [ [ Type ] "=" Expression
List ] .

IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .
```

If the type is present, all constants take the type specified, and the expressions must be assignable to that type. If the type is omitted, the constants take the individual types of the corresponding expressions. If the expression values are untyped constants, the declared constants remain untyped and the constant identifiers denote the constant values. For instance, if the expression is a floating-point literal, the constant identifier denotes a floating-point constant, even if the literal's fractional part is zero.

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0              // untyped floating-point co
nstant
const (
        size int64 = 1024
        eof = -1              // untyped integer constant
)
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo",
untyped integer and string constants
const u, v float = 0, 3      // u = 0.0, v = 3.0
```

Within a parenthesized `const` declaration list the expression list may be omitted from any but the first declaration. Such an empty list is equivalent to the textual substitution of the first preceding non-empty expression list and its type if any. Omitting the list of expressions is therefore equivalent to repeating the previous list. The number of identifiers must be equal to the number of expressions in the previous list. Together with the `iota` constant generator this mechanism permits light-weight declaration of sequential values:

```
const (
        Sunday = iota
        Monday
        Tuesday
        Wednesday
        Thursday
        Friday
        Partyday
        numberOfDays  // this constant is not exported
)
```

Iota

Within a constant declaration, the predeclared identifier `iota` represents successive untyped integer constants. It is reset to 0 whenever the reserved word `const` appears in the source and increments after each ConstSpec. It can be used to construct a set of related constants:

```
const (  // iota is reset to 0
        c0 = iota  // c0 == 0
        c1 = iota  // c1 == 1
        c2 = iota  // c2 == 2
)

const (
        a = 1 << iota  // a == 1 (iota has been reset)
        b = 1 << iota  // b == 2
        c = 1 << iota  // c == 4
)

const (
        u       = iota * 42  // u == 0     (untyped integ
er constant)
        v float = iota * 42  // v == 42.0  (float constan
t)
        w       = iota * 42  // w == 84    (untyped integ
er constant)
)

const x = iota  // x == 0 (iota has been reset)
const y = iota  // y == 0 (iota has been reset)
```

Within an ExpressionList, the value of each `iota` is the same because it is only incremented after each ConstSpec:

```
const (
        bit0, mask0 = 1 << iota, 1 << iota - 1  // bit0 =
= 1, mask0 == 0
        bit1, mask1                             // bit1 =
= 2, mask1 == 1
        _, _                                    // skips
iota == 2
        bit3, mask3                             // bit3 =
= 8, mask3 == 7
)
```

This last example exploits the implicit repetition of the last non-empty expression list.

Type declarations

A type declaration binds an identifier, the *type name*, to a new type that has the same underlying type as an existing type. The new type is different from the existing type.

```
TypeDecl     = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec     = identifier Type .
```

```
type IntArray [16]int

type (
        Point struct { x, y float }
        Polar Point
)

type TreeNode struct {
        left, right *TreeNode
        value *Comparable
}

type Cipher interface {
        BlockSize() int
        Encrypt(src, dst []byte)
        Decrypt(src, dst []byte)
}
```

The declared type does not inherit any methods bound to the existing type, but the method set of an interface type or of elements of a composite type remains unchanged:

```
// A Mutex is a data type with two methods Lock and Unloc
k.
type Mutex struct         { /* Mutex fields */ }
func (m *Mutex) Lock()    { /* Lock implementation */ }
func (m *Mutex) Unlock()  { /* Unlock implementation */ }

// NewMutex has the same composition as Mutex but its met
hod set is empty.
type NewMutex Mutex

// The method set of *PrintableMutex contains the methods
// Lock and Unlock bound to its anonymous field Mutex.
type PrintableMutex struct {
        Mutex
}

// MyCipher is an interface type that has the same method
 set as Cipher.
type MyCipher Cipher
```

A type declaration may be used to define a different boolean, numeric, or string type and attach methods to it:

```
type TimeZone int

const (
        EST TimeZone = -(5 + iota)
        CST
        MST
        PST
)

func (tz TimeZone) String() string {
        return fmt.Sprintf("GMT+%dh", tz)
}
```

Variable declarations

A variable declaration creates a variable, binds an identifier to it and gives it a type and optionally an initial value.

```
VarDecl     = "var" ( VarSpec | "(" { VarSpec ";" } ")" )
 .
VarSpec     = IdentifierList ( Type [ "=" ExpressionList
] | "=" ExpressionList ) .
```

```
var i int
var U, V, W float
var k = 0
var x, y float = -1, -2
var (
        i int
        u, v, s = 2.0, 3.0, "bar"
)
var re, im = complexSqrt(-1)
var _, found = entries[name]  // map lookup; only interes
ted in "found"
```

If a list of expressions is given, the variables are initialized by assigning the expressions to the variables (§Assignments) in order; all expressions must be consumed and all variables initialized from them. Otherwise, each variable is initialized to its zero value.

If the type is present, each variable is given that type. Otherwise, the types are deduced from the assignment of the expression list.

If the type is absent and the corresponding expression evaluates to an untyped constant, the type of the declared variable is `bool`, `int`, `float`, or `string` respectively, depending on whether the value is a boolean, integer, floating-point, or string constant:

```
var b = true    // t has type bool
var i = 0       // i has type int
var f = 3.0     // f has type float
```

```
var s = "OMDB"  // s has type string
```

Short variable declarations

A *short variable declaration* uses the syntax:

```
ShortVarDecl = IdentifierList ":=" ExpressionList .
```

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

```
"var" IdentifierList = ExpressionList .
```

```
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w := os.Pipe(fd)  // os.Pipe() returns two values
_, y, _ := coord(p)  // coord() returns three values; onl
y interested in y coordinate
```

Unlike regular variable declarations, a short variable declaration may redeclare variables provided they were originally declared in the same block 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.

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // redeclares o
ffset
```

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 (§Statements).

Function declarations

A function declaration binds an identifier to a function (§Function types).

```
FunctionDecl = "func" identifier Signature [ Body ] .
Body         = Block.
```

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

```
func min(x int, y int) int {
        if x < y {
                return x
        }
        return y
}

func flushICache(begin, end uintptr)  // implemented exte
rnally
```

Method declarations

A method is a function with a *receiver*. A method declaration binds an identifier to a method.

```
MethodDecl   = "func" Receiver MethodName Signature [ Bod
y ] .
Receiver     = "(" [ identifier ] [ "*" ] BaseTypeName ")
" .
BaseTypeName = identifier .
```

The receiver type must be of the form `T` or `*T` where `T` is a type name. `T` is called the *receiver base type* or just *base type*. The base type must not be a pointer or interface type and must be declared in the same package as the method. The method is said to be *bound* to the base type and is visible only within selectors for that type (§Type declarations, §Selectors).

Given type `Point`, the declarations

```
func (p *Point) Length() float {
        return Math.sqrt(p.x * p.x + p.y * p.y)
}

func (p *Point) Scale(factor float) {
        p.x = p.x * factor
        p.y = p.y * factor
}
```

bind the methods `Length` and `Scale`, with receiver type `*Point`, to the base type `Point`.

If the receiver's value is not referenced inside the body of the method, its identifier may be omitted in the declaration. The same applies in general to parameters of functions and methods.

The type of a method is the type of a function with the receiver as first argument. For instance, the method `Scale` has type

```
(p *Point, factor float)
```

However, a function declared this way is not a method.

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.

```
Operand    = Literal | QualifiedIdent | MethodExpr | "("
Expression ")" .
```

```
Literal    = BasicLit | CompositeLit | FunctionLit .
BasicLit   = int_lit | float_lit | imaginary_lit | char_l
it | string_lit .
```

Qualified identifiers

A qualified identifier is a non-blank identifier qualified by a package name prefix.

```
QualifiedIdent = [ PackageName "." ] identifier .
```

A qualified identifier accesses an identifier in a separate package. The identifier must be exported by that package, which means that it must begin with a Unicode upper case letter.

```
math.Sin
```

Composite literals

Composite literals construct values for structs, arrays, slices, and maps and create a new value each time they are evaluated. They consist of the type of the value followed by a brace-bound list of composite elements. An element may be a single expression or a key-value pair.

```
CompositeLit  = LiteralType "{" [ ElementList [ "," ] ] "
}" .
LiteralType   = StructType | ArrayType | "[" "..." "]" El
ementType |
                SliceType | MapType | TypeName | "(" Lite
ralType ")" .
ElementList   = Element { "," Element } .
Element       = [ Key ":" ] Value .
Key           = FieldName | ElementIndex .
FieldName     = identifier .
ElementIndex  = Expression .
Value         = Expression .
```

The LiteralType must be a struct, array, slice, or map type (the grammar enforces this constraint except when the type is given as a TypeName). The types of the expressions must be assignable to the respective field, element, and key types of the LiteralType; there is no additional conversion. The key is interpreted as a field name for struct literals, an index expression for array and slice literals, and a key for map literals. For map literals, all elements must have a key. It is an error to specify multiple elements with the same field name or constant key value.

For struct literals the following rules apply:

- A key must be a field name declared in the LiteralType.
- A literal that does not contain any keys must list an element for each struct field in the order in which the fields are declared.
- If any element has a key, every element must have a key.
- A literal that contains keys does not need to have an element for each struct field. Omitted fields get the zero value for that field.
- A literal may omit the element list; such a literal evaluates to the zero value for its type.
- It is an error to specify an element for a non-exported field of a struct belonging to a different package.

Given the declarations

```
type Point struct { x, y, z float }
type Line struct { p, q Point }
```

one may write

```
origin := Point{}                            // zero valu
e for Point
line := Line{origin, Point{y: -4, z: 12.3}}  // zero valu
e for line.q.x
```

For array and slice literals the following rules apply:

- Each element has an associated integer index marking its position in the array.
- An element with a key uses the key as its index; the key must be a constant integer expression.
- An element without a key uses the previous element's index plus one. If the first element has no key, its index is zero.

Taking the address of a composite literal (§Address operators) generates a unique pointer to an instance of the literal's value.

```
var pointer *Point = &Point{y: 1000}
```

The length of an array literal is the length specified in the LiteralType. If fewer elements than the length are provided in the literal, the missing elements are set to the zero value for the array element type. It is an error to provide elements with index values outside the index range of the array. The notation `...` specifies an array length equal to the maximum element index plus one.

```
buffer := [10]string{}               // len(buffer) == 10
intSet := [6]int{1, 2, 3, 5}         // len(intSet) == 6
days := [...]string{"Sat", "Sun"}    // len(days) == 2
```

A slice literal describes the entire underlying array literal. Thus, the length and capacity of a slice literal are the maximum element index plus one. A slice literal has the form

```
[]T{x1, x2, ... xn}
```

and is a shortcut for a slice operation applied to an array literal:

```
[n]T{x1, x2, ... xn}[0 : n]
```

A parsing ambiguity arises when a composite literal using the TypeName form of the LiteralType appears in the condition of an "if", "for", or "switch" statement,

because the braces surrounding the expressions in the literal are confused with those introducing a block of statements. To resolve the ambiguity in this rare case, the composite literal must appear within parentheses.

```
if x == (T{a,b,c}[i]) { ... }
if (x == T{a,b,c}[i]) { ... }
```

Examples of valid array, slice, and map literals:

```
// list of prime numbers
primes := []int{2, 3, 5, 7, 9, 11, 13, 17, 19, 991}

// vowels[ch] is true if ch is a vowel
vowels := [128]bool{'a': true, 'e': true, 'i': true, 'o':
 true, 'u': true, 'y': true}

// the array [10]float{-1, 0, 0, 0, -0.1, -0.1, 0, 0, 0,
-1}
filter := [10]float{-1, 4: -0.1, -0.1, 9: -1}

// frequencies in Hz for equal-tempered scale (A4 = 440Hz
)
noteFrequency := map[string]float{
        "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.8
3,
        "G0": 24.50, "A0": 27.50, "B0": 30.87,
}
```

Function literals

A function literal represents an anonymous function. It consists of a specification of the function type and a function body.

```
FunctionLit = FunctionType Body .
```

```
func(a, b int, z float) bool { return a*b < int(z) }
```

A function literal can be assigned to a variable or invoked directly.

```
f := func(x, y int) int { return x + y }
func(ch chan int) { ch <- ACK } (reply_chan)
```

Function literals are *closures*: they may refer to variables defined in a surrounding function. Those variables are then shared between the surrounding function and the function literal, and they survive as long as they are accessible.

Primary expressions

Primary expressions are the operands for unary and binary expressions.

```
PrimaryExpr =
        Operand |
        Conversion |
        BuiltinCall |
        PrimaryExpr Selector |
        PrimaryExpr Index |
        PrimaryExpr Slice |
        PrimaryExpr TypeAssertion |
        PrimaryExpr Call .

Selector       = "." identifier .
Index          = "[" Expression "]" .
Slice          = "[" Expression ":" [ Expression ] "]" .
TypeAssertion  = "." "(" Type ")" .
Call           = "(" [ ExpressionList [ "," ] ] ")" .
```

```
x
2
(s + ".txt")
f(3.1415, true)
```

```
Point{1, 2}
m["foo"]
s[i : j + 1]
obj.color
Math.sin
f.p[i].x()
```

Selectors

A primary expression of the form

```
x.f
```

denotes the field or method `f` of the value denoted by `x` (or of `*x` if `x` is of pointer type). The identifier `f` is called the (field or method) *selector*; it must not be the blank identifier. The type of the expression is the type of `f`.

A selector `f` may denote a field or method `f` of a type `T`, or it may refer to a field or method `f` of a nested anonymous field of `T`. The number of anonymous fields traversed to reach `f` is called its *depth* in `T`. The depth of a field or method `f` declared in `T` is zero. The depth of a field or method `f` declared in an anonymous field `A` in `T` is the depth of `f` in `A` plus one.

The following rules apply to selectors:

1. For a value `x` of type `T` or `*T` where `T` is not an interface type, `x.f` denotes the field or method at the shallowest depth in `T` where there is such an `f`. If there is not exactly one `f` with shallowest depth, the selector expression is illegal.
2. For a variable `x` of type `I` or `*I` where `I` is an interface type, `x.f` denotes the actual method with name `f` of the value assigned to `x` if there is such a method. If no value or `nil` was assigned to `x`, `x.f` is illegal.
3. In all other cases, `x.f` is illegal.

Selectors automatically dereference pointers. If `x` is of pointer type, `x.y` is shorthand for `(*x).y`; if `y` is also of pointer type, `x.y.z` is shorthand for `(*(*x).y).z`, and so on. If `*x` is of pointer type, dereferencing must be explicit;

only one level of automatic dereferencing is provided. For an `x` of type `T` containing an anonymous field declared as `*A`, `x.f` is a shortcut for `(*x.A).f`.

For example, given the declarations:

```
type T0 struct {
        x int
}

func (recv *T0) M0()

type T1 struct {
        y int
}

func (recv T1) M1()

type T2 struct {
        z int
        T1
        *T0
}

func (recv *T2) M2()

var p *T2  // with p != nil and p.T1 != nil
```

one may write:

```
p.z         // (*p).z
p.y         // ((*p).T1).y
p.x         // (*(*p).T0).x

p.M2        // (*p).M2
p.M1        // ((*p).T1).M1
p.M0        // ((*p).T0).M0
```

Indexes

A primary expression of the form

```
a[x]
```

denotes the element of the array, slice, string or map `a` indexed by `x`. The value `x` is called the *index* or *map key*, respectively. The following rules apply:

For `a` of type `A` or `*A` where `A` is an array type, or for `a` of type `S` where `S` is a slice type:

- `x` must be an integer value and `0 <= x < len(a)`
- `a[x]` is the array element at index `x` and the type of `a[x]` is the element type of `A`
- if the index `x` is out of range, a run-time panic occurs

For `a` of type `T` where `T` is a string type:

- `x` must be an integer value and `0 <= x < len(a)`
- `a[x]` is the byte at index `x` and the type of `a[x]` is `byte`
- `a[x]` may not be assigned to
- if the index `x` is out of range, a run-time panic occurs

For `a` of type `M` where `M` is a map type:

- `x`'s type must be assignable to the key type of `M`
- if the map contains an entry with key `x`, `a[x]` is the map value with key `x` and the type of `a[x]` is the value type of `M`
- if the map does not contain such an entry, `a[x]` is the zero value for the value type of `M`

Otherwise `a[x]` is illegal.

An index expression on a map `a` of type `map[K]V` may be used in an assignment or initialization of the special form

```
v, ok = a[x]
v, ok := a[x]
var v, ok = a[x]
```

where the result of the index expression is a pair of values with types `(V, bool)`. In this form, the value of `ok` is `true` if the key `x` is present in the map, and `false` otherwise. The value of `v` is the value `a[x]` as in the single-result form.

Similarly, if an assignment to a map has the special form

```
a[x] = v, ok
```

and boolean `ok` has the value `false`, the entry for key `x` is deleted from the map; if `ok` is `true`, the construct acts like a regular assignment to an element of the map.

Slices

For a string, array, or slice `a`, the primary expression

```
a[lo : hi]
```

constructs a substring or slice. The index expressions `lo` and `hi` select which elements appear in the result. The result has indexes starting at 0 and length equal to `hi` - `lo`. After slicing the array `a`

```
a := [5]int{1, 2, 3, 4, 5}
s := a[1:4]
```

the slice `s` has type `[]int`, length 3, capacity 4, and elements

```
s[0] == 2
s[1] == 3
```

```
s[2] == 4
```

For convenience, the `hi` expression may be omitted; the notation `a[lo :]` is shorthand for `a[lo : len(a)]`. For arrays or strings, the indexes `lo` and `hi` must satisfy 0 <= `lo` <= `hi` <= length; for slices, the upper bound is the capacity rather than the length.

If the sliced operand is a string or slice, the result of the slice operation is a string or slice of the same type. If the sliced operand is an array, the result of the slice operation is a slice with the same element type as the array.

Type assertions

For an expression `x` of interface type and a type `T`, the primary expression

```
x.(T)
```

asserts that `x` is not `nil` and that the value stored in `x` is of type `T`. The notation `x.(T)` is called a *type assertion*.

More precisely, if `T` is not an interface type, `x.(T)` asserts that the dynamic type of `x` is identical to the type `T`. If `T` is an interface type, `x.(T)` asserts that the dynamic type of `x` implements the interface `T` (§Interface types).

If the type assertion holds, the value of the expression is the value stored in `x` and its type is `T`. If the type assertion is false, a run-time panic occurs. In other words, even though the dynamic type of `x` is known only at run-time, the type of `x.(T)` is known to be `T` in a correct program.

If a type assertion is used in an assignment or initialization of the form

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
```

the result of the assertion is a pair of values with types `(T, bool)`. If the assertion holds, the expression returns the pair `(x.(T), true)`; otherwise, the expression returns `(Z, false)` where `Z` is the zero value for type `T`. No run-time

panic occurs in this case. The type assertion in this construct thus acts like a function call returning a value and a boolean indicating success. (§Assignments)

Calls

Given an expression `f` of function type `F`,

```
f(a1, a2, ... an)
```

calls `f` with arguments `a1, a2, ... an`. Except for one special case, arguments must be single-valued expressions assignable to the parameter types of `F` and are evaluated before the function is called. The type of the expression is the result type of `F`. A method invocation is similar but the method itself is specified as a selector upon a value of the receiver type for the method.

```
math.Atan2(x, y)    // function call
var pt *Point
pt.Scale(3.5)  // method call with receiver pt
```

As a special case, if the return parameters of a function or method `g` are equal in number and individually assignable to the parameters of another function or method `f`, then the call `f(g(`*`parameters_of_g`*`))` will invoke `f` after binding the return values of `g` to the parameters of `f` in order. The call of `f` must contain no parameters other than the call of `g`. If `f` has a final `...` parameter, it is assigned the return values of `g` that remain after assignment of regular parameters.

```
func Split(s string, pos int) (string, string) {
        return s[0:pos], s[pos:]
}

func Join(s, t string) string {
        return s + t
}

if Join(Split(value, len(value)/2)) != value {
        log.Crash("test fails")
}
```

A method call `x.m()` is valid if the method set of (the type of) `x` contains `m` and the argument list can be assigned to the parameter list of `m`. If `x` is addressable and `&x`'s method set contains `m`, `x.m()` is shorthand for `(&x).m()`:

```
var p Point
p.Scale(3.5)
```

There is no distinct method type and there are no method literals.

Passing arguments to `...` parameters

If `f` is variadic with final parameter type `...T`, then within the function the argument is equivalent to a parameter of type `[]T`. At each call of `f`, the argument passed to the final parameter is a new slice of type `[]T` whose successive elements are the actual arguments. The length of the slice is therefore the number of arguments bound to the final parameter and may differ for each call site.

Given the function and call

```
func Greeting(prefix string, who ... string)
Greeting("hello:", "Joe", "Anna", "Eileen")
```

Within `Greeting`, `who` will have value `[]string{"Joe", "Anna", "Eileen")`

As a special case, if a function passes its own `...` parameter as the `...` argument in a call to another function with a `...` parameter of identical type, the parameter is passed directly. In short, a formal `...` parameter is passed unchanged as an actual `...` parameter provided the types match.

Operators

Operators combine operands into expressions.

```
Expression = UnaryExpr | Expression binary_op UnaryExpr .
```

```
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .

binary_op  = log_op | com_op | rel_op | add_op | mul_op .
log_op     = "||" | "&&" .
com_op     = "<-" .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

Comparisons are discussed elsewhere. For other binary operators, the operand types must be identical unless the operation involves channels, shifts, or untyped constants. For operations involving constants only, see the section on constant expressions.

In a channel send, the first operand is always a channel and the second must be a value assignable to the channel's element type.

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

The right operand in a shift operation must have unsigned integer type or be an untyped constant that can be converted to unsigned integer type.

If the left operand of a non-constant shift operation is an untyped constant, the type of constant is what it would be if the shift operation were replaced by the left operand alone.

```
var s uint = 33
var i = 1<<s           // 1 has type int
var j = int32(1<<s)    // 1 has type int32; j == 0
var u = uint64(1<<s)   // 1 has type uint64; u == 1<<33
var f = float(1<<s)    // illegal: 1 has type float, canno
t shift
var g = float(1<<33)   // legal; 1<<33 is a constant shift
 operation; g == 1<<33
```

Operator precedence

Unary operators have the highest precedence. As the ++ and -- operators form statements, not expressions, they fall outside the operator hierarchy. As a consequence, statement *p++ is the same as (*p)++.

There are six precedence levels for binary operators. Multiplication operators bind strongest, followed by addition operators, comparison operators, <- (channel send), && (logical and), and finally || (logical or):

```
Precedence    Operator
    6             *  /  %  <<  >>  &  &^
    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.

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y+1 && <-chan_ptr > 0
```

Arithmetic operators

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. All other arithmetic operators apply to integers only.

```
+    sum                    integers, floats, complex val
ues, strings
```

```
-    difference                 integers, floats, complex val
ues
*    product                    integers, floats, complex val
ues
/    quotient                   integers, floats, complex val
ues
%    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
```

Strings can be concatenated using the `+` operator or the `+=` assignment operator:

```
s := "hi" + string(c)
s += " and good bye"
```

String addition creates a new string by concatenating the operands.

For integer values, `/` and `%` satisfy the following relationship:

```
(a / b) * b + a % b == a
```

with `(a / b)` truncated towards zero.

```
 x     y     x / y     x % y
 5     3       1         2
-5     3      -1        -2
 5    -3      -1         2
-5    -3       1        -2
```

If the divisor is zero, a run-time panic occurs. If the dividend is positive and the divisor is a constant power of 2, the division may be replaced by a right shift, and computing the remainder may be replaced by a bitwise "and" operation:

```
 x      x / 4     x % 4     x >> 2     x & 3
 11       2         3         2         3
-11      -2        -3        -3         1
```

The shift operators shift the left operand by the shift count specified by the right operand. They implement arithmetic shifts if the left operand is a signed integer and logical shifts if it is an unsigned integer. The shift count must be an unsigned integer. There is no upper limit on the shift count. Shifts behave as if the left operand is shifted `n` times by 1 for a shift count of `n`. As a result, `x << 1` is the same as `x*2` and `x >> 1` is the same as `x/2` but truncated towards negative infinity.

For integer operands, the unary operators `+`, `-`, and `^` are defined as follows:

```
+x                          is 0 + x
-x    negation              is 0 - x
^x    bitwise complement    is m ^ x  with m = "all bits
set to 1" for unsigned x
                                    and  m = -1 for sig
ned x
```

For floating-point numbers, `+x` is the same as `x`, while `-x` is the negation of `x`. The result of a floating-point division by zero is not specified beyond the IEEE-754 standard; whether a run-time panic occurs is implementation-specific.

Integer overflow

For unsigned integer values, the operations `+`, `-`, `*`, and `<<` are computed modulo 2^n, where *n* is the bit width of the unsigned integer's type (§Numeric types). Loosely speaking, these unsigned integer operations discard high bits upon overflow, and programs may rely on ``wrap around".

For signed integers, the operations `+`, `-`, `*`, and `<<` may legally overflow and the

resulting value exists and is deterministically defined by the signed integer representation, the operation, and its operands. No exception is raised as a result of overflow. A compiler may not optimize code under the assumption that overflow does not occur. For instance, it may not assume that `x < x + 1` is always true.

Comparison operators

Comparison operators compare two operands and yield a value of type `bool`.

```
==    equal
!=    not equal
<     less
<=    less or equal
>     greater
>=    greater or equal
```

The operands must be *comparable*; that is, the first operand must be assignable to the type of the second operand, or vice versa.

The operators `==` and `!=` apply to operands of all types except arrays and structs. All other comparison operators apply only to integer, floating-point and string values. The result of a comparison is defined as follows:

- Integer values are compared in the usual way.
- Floating point values are compared as defined by the IEEE-754 standard.
- Two complex values `u`, `v` are equal if both `real(u) == real(v)` and `imag(u) == imag(v)`.
- String values are compared byte-wise (lexically).
- Boolean values are are equal if they are either both `true` or both `false`.
- Pointer values are equal if they point to the same location or if both are `nil`.
- Function values are equal if they refer to the same function or if both are `nil`.
- A slice value may only be compared to `nil`.
- Channel and map values are equal if they were created by the same call to `make` (§Making slices, maps, and channels) or if both are `nil`.

- Interface values are equal if they have identical dynamic types and equal dynamic values or if both are `nil`.
- An interface value `x` is equal to a non-interface value `y` if the dynamic type of `x` is identical to the static type of `y` and the dynamic value of `x` is equal to `y`.
- A pointer, function, slice, channel, map, or interface value is equal to `nil` if it has been assigned the explicit value `nil`, if it is uninitialized, or if it has been assigned another value equal to `nil`.

Logical operators

Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally.

```
&&    conditional and    p && q  is  "if p then q else fa
lse"
||    conditional or     p || q  is  "if p then true else
 q"
!     not                !p      is  "not p"
```

Address operators

The address-of operator `&` generates the address of its operand, which must be *addressable*, that is, either a variable, pointer indirection, or slice indexing operation; or a field selector of an addressable struct operand; or an array indexing operation of an addressable array. Given an operand of pointer type, the pointer indirection operator `*` retrieves the value pointed to by the operand.

```
&x
&a[f(2)]
*p
*pf(x)
```

Communication operators

The term *channel* means "value of channel type".

The send operation uses the binary operator "<-", which operates on a channel

and a value (expression):

```
ch <- 3
```

The send operation sends the value on the channel. Both the channel and the expression are evaluated before communication begins. Communication blocks until the send can proceed, at which point the value is transmitted on the channel. A send on an unbuffered channel can proceed if a receiver is ready. A send on a buffered channel can proceed if there is room in the buffer.

If the send operation appears in an expression context, the value of the expression is a boolean and the operation is non-blocking. The value of the boolean reports true if the communication succeeded, false if it did not. (The channel and the expression to be sent are evaluated regardless.) These two examples are equivalent:

```
ok := ch <- 3
if ok { print("sent") } else { print("not sent") }

if ch <- 3 { print("sent") } else { print("not sent") }
```

In other words, if the program tests the value of a send operation, the send is non-blocking and the value of the expression is the success of the operation. If the program does not test the value, the operation blocks until it succeeds.

The receive operation uses the prefix unary operator "<-". The value of the expression is the value received, whose type is the element type of the channel.

```
<-ch
```

The expression blocks until a value is available, which then can be assigned to a variable or used like any other expression. If the receive expression does not save the value, the value is discarded.

```
v1 := <-ch
v2 = <-ch
```

```
f(<-ch)
<-strobe  // wait until clock pulse
```

If a receive expression is used in an assignment or initialization of the form

```
x, ok = <-ch
x, ok := <-ch
var x, ok = <-ch
```

the receive operation becomes non-blocking. If the operation can proceed, the boolean variable `ok` will be set to `true` and the value stored in `x`; otherwise `ok` is set to `false` and `x` is set to the zero value for its type (§The zero value).

Except in a communications clause of a select statement, sending or receiving from a `nil` channel causes a run-time panic.

Method expressions

If `M` is in the method set of type `T`, `T.M` is a function that is callable as a regular function with the same arguments as `M` prefixed by an additional argument that is the receiver of the method.

```
MethodExpr    = ReceiverType "." MethodName .
ReceiverType  = TypeName | "(" "*" TypeName ")" .
```

Consider a struct type `T` with two methods, `Mv`, whose receiver is of type `T`, and `Mp`, whose receiver is of type `*T`.

```
type T struct {
        a int
}
func (tv  T) Mv(a int)   int   { return 0 }  // value rec
eiver
func (tp *T) Mp(f float) float { return 1 }  // pointer r
eceiver
var t T
```

The expression

```
T.Mv
```

yields a function equivalent to `Mv` but with an explicit receiver as its first argument; it has signature

```
func(tv T, a int) int
```

That function may be called normally with an explicit receiver, so these three invocations are equivalent:

```
t.Mv(7)
T.Mv(t, 7)
f := T.Mv; f(t, 7)
```

Similarly, the expression

```
(*T).Mp
```

yields a function value representing `Mp` with signature

```
func(tp *T, f float) float
```

For a method with a value receiver, one can derive a function with an explicit pointer receiver, so

```
(*T).Mv
```

yields a function value representing `Mv` with signature

```
func(tv *T, a int) int
```

Such a function indirects through the receiver to create a value to pass as the receiver to the underlying method; the method does not overwrite the value whose address is passed in the function call.

The final case, a value-receiver function for a pointer-receiver method, is illegal because pointer-receiver methods are not in the method set of the value type.

Function values derived from methods are called with function call syntax; the receiver is provided as the first argument to the call. That is, given `f := T.Mv`, `f` is invoked as `f(t, 7)` not `t.f(7)`. To construct a function that binds the receiver, use a closure.

It is legal to derive a function value from a method of an interface type. The resulting function takes an explicit receiver of that interface type.

Conversions

Conversions are expressions of the form `T(x)` where `T` is a type and `x` is an expression that can be converted to type `T`.

```
Conversion = Type "(" Expression ")" .
```

If the type starts with an operator it must be parenthesized:

```
*Point(p)        // same as *(Point(p))
(*Point)(p)      // p is converted to (*Point)
<-chan int(c)    // same as <-(chan int(c))
(<-chan int)(c)  // c is converted to (<-chan int)
```

A value `x` can be converted to type `T` in any of these cases:

- `x` is assignable to `T`.
- `x`'s type and `T` have identical underlying types.
- `x`'s type and `T` are unnamed pointer types and their pointer base types have

identical underlying types.

- `x`'s type and `T` are both integer or floating point types.
- `x`'s type and `T` are both complex types.
- `x` is an integer or has type `[]byte` or `[]int` and `T` is a string type.
- `x` is a string and `T` is `[]byte` or `[]int`.

Specific rules apply to conversions between numeric types or to and from a string type. These conversions may change the representation of `x` and incur a run-time cost. All other conversions only change the type but not the representation of `x`.

Conversions between numeric types

1. When converting between integer types, if the value is a signed integer, it is sign extended to implicit infinite precision; otherwise it is zero extended. It is then truncated to fit in the result type's size. For example, if `v := uint16(0x10F0)`, then `uint32(int8(v)) == 0xFFFFFFF0`. The conversion always yields a valid value; there is no indication of overflow.
2. When converting a floating-point number to an integer, the fraction is discarded (truncation towards zero).
3. When converting an integer or floating-point number to a floating-point type, or a complex number to another complex type, the result value is rounded to the precision specified by the destination type. For instance, the value of a variable `x` of type `float32` may be stored using additional precision beyond that of an IEEE-754 32-bit number, but float32(x) represents the result of rounding `x`'s value to 32-bit precision. Similarly, `x + 0.1` may use more than 32 bits of precision, but `float32(x + 0.1)` does not.

In all conversions involving floating-point or complex values, if the result type cannot represent the value the conversion succeeds but the result value is implementation-dependent.

Conversions to and from a string type

1. Converting a signed or unsigned integer value to a string type yields a string containing the UTF-8 representation of the integer. Values outside the range of valid Unicode code points are converted to `"\uFFFD"`.

```
string('a')            // "a"
string(-1)             // "\ufffd" == "\xef\xbf\xbd "
string(0xf8)           // "\u00f8" == "ø" == "\xc3\xb8"
type MyString string
MyString(0x65e5)       // "\u65e5" == "日" == "\xe6\x97
\xa5"
```

2. Converting a value of type `[]byte` (or the equivalent `[]uint8`) to a string type yields a string whose successive bytes are the elements of the slice. If the slice value is `nil`, the result is the empty string.

```
string([]byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'})  //
 "hellø"
```

3. Converting a value of type `[]int` to a string type yields a string that is the concatenation of the individual integers converted to strings. If the slice value is `nil`, the result is the empty string.

```
string([]int{0x767d, 0x9d6c, 0x7fd4})  // "\u767d\u9d6
c\u7fd4" == "白鵬翔"
```

4. Converting a value of a string type to `[]byte` (or `[]uint8`) yields a slice whose successive elements are the bytes of the string. If the string is empty, the result is `[]byte(nil)`.

```
[]byte("hellø")  // []byte{'h', 'e', 'l', 'l', '\xc3',
 '\xb8'}
```

5. Converting a value of a string type to `[]int` yields a slice containing the individual Unicode code points of the string. If the string is empty, the result is `[]int(nil)`.

```
[]int(MyString("白鵬翔"))  // []int{0x767d, 0x9d6c, 0x7
fd4}
```

There is no linguistic mechanism to convert between pointers and integers. The package `unsafe` implements this functionality under restricted circumstances.

Constant expressions

Constant expressions may contain only constant operands and are evaluated at compile-time.

Untyped boolean, numeric, and string constants may be used as operands wherever it is legal to use an operand of boolean, numeric, or string type, respectively. Except for shift operations, if the operands of a binary operation are an untyped integer constant and an untyped floating-point constant, the integer constant is converted to an untyped floating-point constant (relevant for `/` and `%`). Similarly, untyped integer or floating-point constants may be used as operands wherever it is legal to use an operand of complex type; the integer or floating point constant is converted to a complex constant with a zero imaginary part.

Applying an operator to untyped constants results in an untyped constant of the same kind (that is, a boolean, integer, floating-point, complex, or string constant), except for comparison operators, which result in a constant of type `bool`.

Imaginary literals are untyped complex constants (with zero real part) and may be combined in binary operations with untyped integer and floating-point constants; the result is an untyped complex constant. Complex constants are always constructed from constant expressions involving imaginary literals or constants derived from them, or calls of the built-in function `cmplx`.

```
const Σ = 1 - 0.707i
const Δ = Σ + 2.0e-4 - 1/1i
const Φ = iota * 1i
const iΓ = cmplx(0, Γ)
```

Constant expressions are always evaluated exactly; intermediate values and the constants themselves may require precision significantly larger than supported by any predeclared type in the language. The following are legal declarations:

```
const Huge = 1 << 100
```

```
const Four int8 = Huge >> 98
```

The values of *typed* constants must always be accurately representable as values of the constant type. The following constant expressions are illegal:

```
uint(-1)       // -1 cannot be represented as a uint
int(3.14)      // 3.14 cannot be represented as an int
int64(Huge)    // 1<<100 cannot be represented as an int6
4
Four * 300     // 300 cannot be represented as an int8
Four * 100     // 400 cannot be represented as an int8
```

The mask used by the unary bitwise complement operator `^` matches the rule for non-constants: the mask is all 1s for unsigned constants and -1 for signed and untyped constants.

```
^1          // untyped integer constant, equal to -2
uint8(^1)   // error, same as uint8(-2), out of range
^uint8(1)   // typed uint8 constant, same as 0xFF ^ uint8
(1) = uint8(0xFE)
int8(^1)    // same as int8(-2)
^int8(1)    // same as -1 ^ int8(1) = -2
```

Order of evaluation

When evaluating the elements of an assignment or expression, all function calls, method calls and communication operations are evaluated in lexical left-to-right order.

For example, in the assignment

```
y[f()], ok = g(h(), i() + x[j()], <-c), k()
```

the function calls and communication happen in the order `f()`, `h()`, `i()`, `j()`, `<-c`, `g()`, and `k()`. However, the order of those events compared to the evaluation

and indexing of `x` and the evaluation of `y` is not specified.

Floating-point operations within a single expression are evaluated according to the associativity of the operators. Explicit parentheses affect the evaluation by overriding the default associativity. In the expression `x + (y + z)` the addition `y + z` is performed before adding `x`.

Statements

Statements control execution.

```
Statement =
        Declaration | LabeledStmt | SimpleStmt |
        GoStmt | ReturnStmt | BreakStmt | ContinueStmt |
GotoStmt |
        FallthroughStmt | Block | IfStmt | SwitchStmt | S
electStmt | ForStmt |
        DeferStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | IncDecStmt | As
signment | ShortVarDecl .
```

Empty statements

The empty statement does nothing.

```
EmptyStmt = .
```

Labeled statements

A labeled statement may be the target of a `goto`, `break` or `continue` statement.

```
LabeledStmt = Label ":" Statement .
Label       = identifier .
```

```
Error: log.Crash("error encountered")
```

Expression statements

Function calls, method calls, and channel operations can appear in statement context.

```
ExpressionStmt = Expression .
```

```
f(x+y)
<-ch
```

IncDec statements

The "++" and "--" statements increment or decrement their operands by the untyped constant `1`. As with an assignment, the operand must be a variable, pointer indirection, field selector or index expression.

```
IncDecStmt = Expression ( "++" | "--" ) .
```

The following assignment statements are semantically equivalent:

```
IncDec statement    Assignment
x++                 x += 1
x--                 x -= 1
```

Assignments

```
Assignment = ExpressionList assign_op ExpressionList .

assign_op = [ add_op | mul_op ] "=" .
```

Each left-hand side operand must be addressable, a map index expression, or the

blank identifier.

```
x = 1
*p = f()
a[i] = 23
k = <-ch
```

An *assignment operation* `x` *op*`=` `y` where *op* is a binary arithmetic operation is equivalent to `x = x` *op* `y` but evaluates `x` only once. The *op*`=` construct is a single token. In assignment operations, both the left- and right-hand expression lists must contain exactly one single-valued expression.

```
a[i] <<= 2
i &^= 1<<n
```

A tuple assignment assigns the individual elements of a multi-valued operation to a list of variables. There are two forms. In the first, the right hand operand is a single multi-valued expression such as a function evaluation or channel or map operation or a type assertion. The number of operands on the left hand side must match the number of values. For instance, if `f` is a function returning two values,

```
x, y = f()
```

assigns the first value to `x` and the second to `y`. The blank identifier provides a way to ignore values returned by a multi-valued expression:

```
x, _ = f()  // ignore second value returned by f()
```

In the second form, the number of operands on the left must equal the number of expressions on the right, each of which must be single-valued, and the *n*th expression on the right is assigned to the *n*th operand on the left. The expressions on the right are evaluated before assigning to any of the operands on the left, but otherwise the evaluation order is unspecified beyond the usual rules.

```
a, b = b, a  // exchange a and b
```

In assignments, each value must be assignable to the type of the operand to which it is assigned. If an untyped constant is assigned to a variable of interface type, the constant is converted to type `bool`, `int`, `float`, `complex` or `string` respectively, depending on whether the value is a boolean, integer, floating-point, complex, or string constant.

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. A missing condition is equivalent to `true`.

```
IfStmt    = "if" [ SimpleStmt ";" ] [ Expression ] Block
[ "else" Statement ] .
```

```
if x > 0 {
        return true;
}
```

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
if x := f(); x < y {
        return x
} else if x > z {
        return z
} else {
        return y
}
```

Switch statements

"Switch" statements provide multi-way execution. An expression or type specifier is compared to the "cases" inside the "switch" to determine which branch to execute.

```
SwitchStmt = ExprSwitchStmt | TypeSwitchStmt .
```

There are two forms: expression switches and type switches. In an expression switch, the cases contain expressions that are compared against the value of the switch expression. In a type switch, the cases contain types that are compared against the type of a specially annotated switch expression.

Expression switches

In an expression switch, the switch expression is evaluated and the case expressions, which need not be constants, are evaluated left-to-right and top-to-bottom; the first one that equals the switch expression triggers execution of the statements of the associated case; the other cases are skipped. If no case matches and there is a "default" case, its statements are executed. There can be at most one default case and it may appear anywhere in the "switch" statement. A missing switch expression is equivalent to the expression `true`.

```
ExprSwitchStmt = "switch" [ SimpleStmt ";" ] [ Expression
 ] "{" { ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" { Statement ";" } .
ExprSwitchCase = "case" ExpressionList | "default" .
```

In a case or default clause, the last statement only may be a "fallthrough" statement (§Fallthrough statement) to indicate that control should flow from the end of this clause to the first statement of the next clause. Otherwise control flows to the end of the "switch" statement.

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
switch tag {
default: s3()
case 0, 1, 2, 3: s1()
```

```
case 4, 5, 6, 7: s2()
}

switch x := f(); {  // missing switch expression means "t
rue"
case x < 0: return -x
default: return x
}

switch {
case x < y: f1()
case x < z: f2()
case x == 4: f3()
}
```

Type switches

A type switch compares types rather than values. It is otherwise similar to an expression switch. It is marked by a special switch expression that has the form of a type assertion using the reserved word `type` rather than an actual type. Cases then match literal types against the dynamic type of the expression in the type assertion.

```
TypeSwitchStmt  = "switch" [ SimpleStmt ";" ] TypeSwitchG
uard "{" { TypeCaseClause } "}" .
TypeSwitchGuard = [ identifier ":=" ] PrimaryExpr "." "("
 "type" ")" .
TypeCaseClause  = TypeSwitchCase ":" { Statement ";" } .
TypeSwitchCase  = "case" TypeList | "default" .
TypeList        = Type { "," Type } .
```

The TypeSwitchGuard may include a short variable declaration. When that form is used, the variable is declared in each clause. In clauses with a case listing exactly one type, the variable has that type; otherwise, the variable has the type of the expression in the TypeSwitchGuard.

The type in a case may be `nil` (§Predeclared identifiers); that case is used when

the expression in the TypeSwitchGuard is a `nil` interface value.

Given an expression `x` of type `interface{}`, the following type switch:

```
switch i := x.(type) {
case nil:
        printString("x is nil")
case int:
        printInt(i)  // i is an int
case float:
        printFloat(i)  // i is a float
case func(int) float:
        printFunction(i)  // i is a function
case bool, string:
        printString("type is bool or string")  // i is an
 interface{}
default:
        printString("don't know the type")
}
```

could be rewritten:

```
v := x  // x is evaluated exactly once
if v == nil {
        printString("x is nil")
} else if i, is_int := v.(int); is_int {
        printInt(i)  // i is an int
} else if i, is_float := v.(float); is_float {
        printFloat(i)  // i is a float
} else if i, is_func := v.(func(int) float); is_func {
        printFunction(i)  // i is a function
} else {
        i1, is_bool := v.(bool)
        i2, is_string := v.(string)
        if is_bool || is_string {
                i := v
                printString("type is bool or string")  //
```

```
	i is an interface{}
	} else {
		i := v
		printString("don't know the type")  // i
is an interface{}
	}
}
```

The type switch guard may be preceded by a simple statement, which executes before the guard is evaluated.

The "fallthrough" statement is not permitted in a type switch.

For statements

A "for" statement specifies repeated execution of a block. The iteration is controlled by a condition, a "for" clause, or a "range" clause.

```
ForStmt = "for" [ Condition | ForClause | RangeClause ] B
lock .
Condition = Expression .
```

In its simplest form, a "for" statement specifies the repeated execution of a block as long as a boolean condition evaluates to true. The condition is evaluated before each iteration. If the condition is absent, it is equivalent to `true`.

```
for a < b {
	a *= 2
}
```

A "for" statement with a ForClause is also controlled by its condition, but additionally it may specify an *init* and a *post* statement, such as an assignment, an increment or decrement statement. The init statement may be a short variable declaration, but the post statement must not.

```
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt
```

```
 ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .


for i := 0; i < 10; i++ {
        f(i)
}
```

If non-empty, the init statement is executed once before evaluating the condition for the first iteration; the post statement is executed after each execution of the block (and only if the block was executed). Any element of the ForClause may be empty but the semicolons are required unless there is only a condition. If the condition is absent, it is equivalent to `true`.

```
for cond { S() }    is the same as    for ; cond ; { S()
}
for      { S() }    is the same as    for true     { S()
}
```

A "for" statement with a "range" clause iterates through all entries of an array, slice, string or map, or values received on a channel. For each entry it first assigns the current index or key to an iteration variable - or the current (index, element) or (key, value) pair to a pair of iteration variables - and then executes the block.

```
RangeClause = ExpressionList ( "=" | ":=" ) "range" Expre
ssion .
```

The type of the right-hand expression in the "range" clause must be an array, slice, string or map, or a pointer to an array; or it may be a channel. Except for channels, the identifier list must contain one or two expressions (as in assignments, these must be a variable, pointer indirection, field selector, or index expression) denoting the iteration variables. On each iteration, the first variable is set to the string, array or slice index or map key, and the second variable, if present, is set to the corresponding string or array element or map value. The types of the array or slice index (always `int`) and element, or of the map key and

value respectively, must be assignable to the type of the iteration variables. The expression on the right hand side is evaluated once before beginning the loop. At each iteration of the loop, the values produced by the range clause are assigned to the left hand side as in an assignment statement. Function calls on the left hand side will be evaluated exactly once per iteration.

For a value of a string type, the "range" clause iterates over the Unicode code points in the string. On successive iterations, the index variable will be the index of the first byte of successive UTF-8-encoded code points in the string, and the second variable, of type `int`, will be the value of the corresponding code point. If the iteration encounters an invalid UTF-8 sequence, the second variable will be `0xFFFD`, the Unicode replacement character, and the next iteration will advance a single byte in the string.

For channels, the identifier list must contain one identifier. The iteration receives values sent on the channel until the channel is closed; it does not process the zero value sent before the channel is closed.

The iteration variables may be declared by the "range" clause (":="), in which case their scope ends at the end of the "for" statement (§Declarations and scope rules). In this case their types are set to `int` and the array element type, or the map key and value types, respectively. If the iteration variables are declared outside the "for" statement, after execution their values will be those of the last iteration.

```
var a [10]string
m := map[string]int{"mon":0, "tue":1, "wed":2, "thu":3, "
fri":4, "sat":5, "sun":6}

for i, s := range a {
        // type of i is int
        // type of s is string
        // s == a[i]
        g(i, s)
}

var key string
var val interface {}  // value type of m is assignable to
 val
for key, val = range m {
```

```
        h(key, val)
}
// key == last map key encountered in iteration
// val == map[key]
```

If map entries that have not yet been processed are deleted during iteration, they will not be processed. If map entries are inserted during iteration, the behavior is implementation-dependent, but each entry will be processed at most once.

Go statements

A "go" statement starts the execution of a function or method call as an independent concurrent thread of control, or *goroutine*, within the same address space.

```
GoStmt = "go" Expression .
```

The expression must be a call, and unlike with a regular call, program execution does not wait for the invoked function to complete.

```
go Server()
go func(ch chan<- bool) { for { sleep(10); ch <- true; }}
 (c)
```

Select statements

A "select" statement chooses which of a set of possible communications will proceed. It looks similar to a "switch" statement but with the cases all referring to communication operations.

```
SelectStmt = "select" "{" { CommClause } "}" .
CommClause = CommCase ":" { Statement ";" } .
CommCase = "case" ( SendExpr | RecvExpr) | "default" .
SendExpr =  Expression "<-" Expression .
RecvExpr =  [ Expression ( "=" | ":=" ) ] "<-" Expression
 .
```

For all the send and receive expressions in the "select" statement, the channel expressions are evaluated in top-to-bottom order, along with any expressions that appear on the right hand side of send expressions. A channel pointer may be `nil`, which is equivalent to that case not being present in the select statement except, if a send, its expression is still evaluated. If any of the resulting operations can proceed, one of those is chosen and the corresponding communication and statements are evaluated. Otherwise, if there is a default case, that executes; if there is no default case, the statement blocks until one of the communications can complete. If there are no cases with non-`nil` channels, the statement blocks forever. Even if the statement blocks, the channel and send expressions are evaluated only once, upon entering the select statement.

Since all the channels and send expressions are evaluated, any side effects in that evaluation will occur for all the communications in the "select" statement.

If multiple cases can proceed, a pseudo-random fair choice is made to decide which single communication will execute.

The receive case may declare a new variable using a short variable declaration.

```
var c, c1, c2 chan int
var i1, i2 int
select {
case i1 = <-c1:
        print("received ", i1, " from c1\n")
case c2 <- i2:
        print("sent ", i2, " to c2\n")
default:
        print("no communication\n")
}

for {  // send random sequence of bits to c
        select {
        case c <- 0:  // note: no statement, no fallthrou
gh, no folding of cases
        case c <- 1:
        }
}
```

```
select { }  // block forever
```

Return statements

A "return" statement terminates execution of the containing function and optionally provides a result value or values to the caller.

```
ReturnStmt = "return" [ ExpressionList ] .
```

In a function without a result type, a "return" statement must not specify any result values.

```
func no_result() {
        return
}
```

There are three ways to return values from a function with a result type:

1. The return value or values may be explicitly listed in the "return" statement. Each expression must be single-valued and assignable to the corresponding element of the function's result type.

   ```
   func simple_f() int {
           return 2
   }

   func complex_f1() (re float, im float) {
           return -7.0, -4.0
   }
   ```

2. The expression list in the "return" statement may be a single call to a multi-valued function. The effect is as if each value returned from that function were assigned to a temporary variable with the type of the respective value, followed by a "return" statement listing these variables, at which point the

rules of the previous case apply.

```
func complex_f2() (re float, im float) {
        return complex_f1()
}
```

3. The expression list may be empty if the functions's result type specifies names for its result parameters (§Function Types). The result parameters act as ordinary local variables and the function may assign values to them as necessary. The "return" statement returns the values of these variables.

```
func complex_f3() (re float, im float) {
        re = 7.0
        im = 4.0
        return
}
```

Regardless of how they are declared, all the result values are initialized to the zero values for their type (§The zero value) upon entry to the function.

Break statements

A "break" statement terminates execution of the innermost "for", "switch" or "select" statement.

```
BreakStmt = "break" [ Label ] .
```

If there is a label, it must be that of an enclosing "for", "switch" or "select" statement, and that is the one whose execution terminates (§For statements, §Switch statements, §Select statements).

```
L: for i < n {
        switch i {
                case 5: break L
        }
```

```
}
```

Continue statements

A "continue" statement begins the next iteration of the innermost "for" loop at its post statement (§For statements).

```
ContinueStmt = "continue" [ Label ] .
```

If there is a label, it must be that of an enclosing "for" statement, and that is the one whose execution advances (§For statements).

Goto statements

A "goto" statement transfers control to the statement with the corresponding label.

```
GotoStmt = "goto" Label .
```

```
goto Error
```

Executing the "goto" statement must not cause any variables to come into scope that were not already in scope at the point of the goto. For instance, this example:

```
goto L  // BAD
v := 3
L:
```

is erroneous because the jump to label `L` skips the creation of `v`.

Fallthrough statements

A "fallthrough" statement transfers control to the first statement of the next case clause in a expression "switch" statement (§Expression switches). It may be used only as the final non-empty statement in a case or default clause in an expression "switch" statement.

```
FallthroughStmt = "fallthrough" .
```

Defer statements

A "defer" statement invokes a function whose execution is deferred to the moment the surrounding function returns.

```
DeferStmt = "defer" Expression .
```

The expression must be a function or method call. Each time the "defer" statement executes, the parameters to the function call are evaluated and saved anew but the function is not invoked. Deferred function calls are executed in LIFO order immediately before the surrounding function returns, after the return values, if any, have been evaluated, but before they are returned to the caller. For instance, if the deferred function is a function literal and the surrounding function has named result parameters that are in scope within the literal, the deferred function may access and modify the result parameters before they are returned.

```
lock(l)
defer unlock(l)  // unlocking happens before surrounding
function returns

// prints 3 2 1 0 before surrounding function returns
for i := 0; i <= 3; i++ {
        defer fmt.Print(i)
}

// f returns 1
func f() (result int) {
        defer func() {
                result++
        }()
        return 0
}
```

Built-in functions

Built-in functions are predeclared. They are called like any other function but some of them accept a type instead of an expression as the first argument.

The built-in functions do not have standard Go types, so they can only appear in call expressions; they cannot be used as function values.

```
BuiltinCall = identifier "(" [ BuiltinArgs ] ")" .
BuiltinArgs = Type [ "," ExpressionList ] | ExpressionLis
t .
```

Close and closed

For a channel `c`, the built-in function `close(c)` marks the channel as unable to accept more values through a send operation; values sent to a closed channed are ignored. After calling `close`, and after any previously sent values have been received, receive operations will return the zero value for the channel's type without blocking. After at least one such zero value has been received, `closed(c)` returns true.

Length and capacity

The built-in functions `len` and `cap` take arguments of various types and return a result of type `int`. The implementation guarantees that the result always fits into an `int`.

```
Call      Argument type        Result

len(s)    string type          string length in bytes
          [n]T, *[n]T          array length (== n)
          []T                  slice length
          map[K]T              map length (number of defi
ned keys)
          chan T               number of elements queued
in channel buffer
```

```
cap(s)    [n]T, *[n]T          array length (== n)
          []T                  slice capacity
          chan T               channel buffer capacity
```

The capacity of a slice is the number of elements for which there is space allocated in the underlying array. At any time the following relationship holds:

```
0 <= len(s) <= cap(s)
```

The length and capacity of a `nil` slice, map, or channel are 0.

The expression `len(s)` is a constant if `s` is a string constant. The expressions `len(s)` and `cap(s)` are constants if `s` is an (optionally parenthesized) identifier or qualified identifier denoting an array or pointer to array. Otherwise invocations of `len` and `cap` are not constant.

Allocation

The built-in function `new` takes a type `T` and returns a value of type `*T`. The memory is initialized as described in the section on initial values (§The zero value).

```
new(T)
```

For instance

```
type S struct { a int; b float }
new(S)
```

dynamically allocates memory for a variable of type `S`, initializes it (`a=0`, `b=0.0`), and returns a value of type `*S` containing the address of the memory.

Making slices, maps and channels

Slices, maps and channels are reference types that do not require the extra indirection of an allocation with `new`. The built-in function `make` takes a type `T`,

which must be a slice, map or channel type, optionally followed by a type-specific list of expressions. It returns a value of type `T` (not `*T`). The memory is initialized as described in the section on initial values (§The zero value).

```
Call             Type T     Result

make(T, n)       slice      slice of type T with length n
 and capacity n
make(T, n, m)    slice      slice of type T with length n
 and capacity m

make(T)          map        map of type T
make(T, n)       map        map of type T with initial sp
ace for n elements

make(T)          channel    synchronous channel of type T
make(T, n)       channel    asynchronous channel of type
T, buffer size n
```

The arguments `n` and `m` must be of integer type. A run-time panic occurs if `n` is negative or larger than `m`, or if `n` or `m` cannot be represented by an `int`.

```
s := make([]int, 10, 100)        // slice with len(s) ==
10, cap(s) == 100
s := make([]int, 10)             // slice with len(s) ==
cap(s) == 10
c := make(chan int, 10)          // channel with a buffer
 size of 10
m := make(map[string] int, 100)  // map with initial spac
e for 100 elements
```

Copying slices

The built-in function `copy` copies slice elements from a source `src` to a destination `dst` and returns the number of elements copied. Source and destination may overlap. Both arguments must have identical element type `T` and must be assignable to a slice of type `[]T`. The number of arguments copied is the

minimum of `len(src)` and `len(dst)`.

```
copy(dst, src []T) int
```

Examples:

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
n1 := copy(s, a[0:])  // n1 == 6, s == []int{0, 1, 2, 3,
4, 5}
n2 := copy(s, s[2:])  // n2 == 4, s == []int{2, 3, 4, 5,
4, 5}
```

Assembling and disassembling complex numbers

Three functions assemble and disassemble complex numbers. The built-in function `cmplx` constructs a complex value from a floating-point real and imaginary part, while `real` and `imag` extract the real and imaginary parts of a complex value.

```
cmplx(realPart, imaginaryPart floatT) complexT
real(complexT) floatT
imag(complexT) floatT
```

The type of the arguments and return value correspond. For `cmplx`, the two arguments must be of the same floating-point type and the return type is the complex type with the corresponding floating-point constituents: `complex` for `float`, `complex64` for `float32`, `complex128` for `float64`. The `real` and `imag` functions together form the inverse, so for a complex value `z`, `z == cmplx(real(z), imag(z))`.

If the operands of these functions are all constants, the return value is a constant.

```
var a = cmplx(2, -2)  // has type complex
var b = cmplx(1.0, -1.4)  // has type complex
```

```
x := float32(math.Cos(math.Pi/2))
var c64 = cmplx(5, -x)  // has type complex64
var im = imag(b)  // has type float
var rl = real(c64)  // type float32
```

Handling panics

Two built-in functions, `panic` and `recover`, assist in reporting and handling run-time panics and program-defined error conditions.

```
func panic(interface{})
func recover() interface{}
```

TODO: Most of this text could move to the respective comments in `runtime.go` once the functions are implemented. They are here, at least for now, for reference and discussion.

When a function `F` calls `panic`, normal execution of `F` stops immediately. Any functions whose execution was deferred by the invocation of `F` are run in the usual way, and then `F` returns to its caller. To the caller, `F` then behaves like a call to `panic`, terminating its own execution and running deferred functions. This continues until all functions in the goroutine have ceased execution, in reverse order. At that point, the program is terminated and the error condition is reported, including the value of the argument to `panic`. This termination sequence is called *panicking*.

The `recover` function allows a program to manage behavior of a panicking goroutine. Executing a `recover` call inside a deferred function (but not any function called by it) stops the panicking sequence by restoring normal execution, and retrieves the error value passed to the call of `panic`. If `recover` is called outside the deferred function it will not stop a panicking sequence. In this case, and when the goroutine is not panicking, `recover` returns `nil`.

If the function defined here,

```
func f(hideErrors bool) {
        defer func() {
                if x := recover(); x != nil {
```

```
                println("panicking with value", x
)
                if !hideErrors {
                        panic(x)  // go back to p
anicking
                }
        }
        println("function returns normally") // e
xecutes only when hideErrors==true
    }()
    println("before")
    p()
    println("after")        // never executes
}

func p() {
    panic(3)
}
```

is called with `hideErrors=true`, it prints

```
before
panicking with value 3
function returns normally
```

and resumes normal execution in the function that called `f`. Otherwise, it prints

```
before
panicking with value 3
```

and, absent further `recover` calls, terminates the program.

Since deferred functions run before assigning the return values to the caller of the deferring function, a deferred invocation of a function literal may modify the invoking function's return values in the event of a panic. This permits a function to protect its caller from panics that occur in functions it calls.

```
func IsPrintable(s string) (ok bool) {
        ok = true
        defer func() {
                if recover() != nil {
                        println("input is not printable")
                        ok = false
                }
                // Panicking has stopped; execution will
resume normally in caller.
                // The return value will be true normally
, false if a panic occurred.
        }
        panicIfNotPrintable(s)  // will panic if validati
ons fails.
}
```

Bootstrapping

Current implementations provide several built-in functions useful during bootstrapping. These functions are documented for completeness but are not guaranteed to stay in the language. They do not return a result.

```
Function   Behavior

print      prints all arguments; formatting of arguments
is implementation-specific
println    like print but prints spaces between arguments
 and a newline at the end
```

Packages

Go programs are constructed by linking together *packages*. A package in turn is constructed from one or more source files that together declare constants, types, variables and functions belonging to the package and which are accessible in all files of the same package. Those elements may be exported and used in another

package.

Source file organization

Each source file consists of a package clause defining the package to which it belongs, followed by a possibly empty set of import declarations that declare packages whose contents it wishes to use, followed by a possibly empty set of declarations of functions, types, variables, and constants.

```
SourceFile       = PackageClause ";" { ImportDecl ";" } {
 TopLevelDecl ";" } .
```

Package clause

A package clause begins each source file and defines the package to which the file belongs.

```
PackageClause  = "package" PackageName .
PackageName    = identifier .
```

The PackageName must not be the blank identifier.

```
package math
```

A set of files sharing the same PackageName form the implementation of a package. An implementation may require that all source files for a package inhabit the same directory.

Import declarations

An import declaration states that the source file containing the declaration uses identifiers exported by the *imported* package and enables access to them. The import names an identifier (PackageName) to be used for access and an ImportPath that specifies the package to be imported.

```
ImportDecl       = "import" ( ImportSpec | "(" { ImportSp
```

```
ec ";" } ")" ) .
ImportSpec       = [ "." | PackageName ] ImportPath .
ImportPath       = string_lit .
```

The PackageName is used in qualified identifiers to access the exported identifiers of the package within the importing source file. It is declared in the file block. If the PackageName is omitted, it defaults to the identifier specified in the package clause of the imported package. If an explicit period (`.`) appears instead of a name, all the package's exported identifiers will be declared in the current file's file block and can be accessed without a qualifier.

The interpretation of the ImportPath is implementation-dependent but it is typically a substring of the full file name of the compiled package and may be relative to a repository of installed packages.

Assume we have compiled a package containing the package clause `package math`, which exports function `Sin`, and installed the compiled package in the file identified by `"lib/math"`. This table illustrates how `Sin` may be accessed in files that import the package after the various types of import declaration.

```
Import declaration          Local name of Sin

import   "lib/math"         math.Sin
import M "lib/math"         M.Sin
import . "lib/math"         Sin
```

An import declaration declares a dependency relation between the importing and imported package. It is illegal for a package to import itself or to import a package without referring to any of its exported identifiers. To import a package solely for its side-effects (initialization), use the blank identifier as explicit package name:

```
import _ "lib/math"
```

An example package

Here is a complete Go package that implements a concurrent prime sieve.

```
package main

import "fmt"

// Send the sequence 2, 3, 4, ... to channel 'ch'.
func generate(ch chan<- int) {
        for i := 2; ; i++ {
                ch <- i  // Send 'i' to channel 'ch'.
        }
}

// Copy the values from channel 'src' to channel 'dst',
// removing those divisible by 'prime'.
func filter(src <-chan int, dst chan<- int, prime int) {
        for i := range src {    // Loop over values recei
ved from 'src'.
                if i%prime != 0 {
                        dst <- i  // Send 'i' to channel
'dst'.
                }
        }
}

// The prime sieve: Daisy-chain filter processes together
.
func sieve() {
        ch := make(chan int)  // Create a new channel.
        go generate(ch)       // Start generate() as a su
bprocess.
        for {
                prime := <-ch
                fmt.Print(prime, "\n")
                ch1 := make(chan int)
                go filter(ch, ch1, prime)
                ch = ch1
        }
}
```

```
func main() {
        sieve()
}
```

Program initialization and execution

The zero value

When memory is allocated to store a value, either through a declaration or `make()` or `new()` call, and no explicit initialization is provided, the memory is given a default initialization. Each element of such a value is set to the *zero value* for its type: `false` for booleans, `0` for integers, `0.0` for floats, `""` for strings, and `nil` for pointers, functions, interfaces, slices, channels, and maps. This initialization is done recursively, so for instance each element of an array of structs will have its fields zeroed if no value is specified.

These two simple declarations are equivalent:

```
var i int
var i int = 0
```

After

```
type T struct { i int; f float; next *T }
t := new(T)
```

the following holds:

```
t.i == 0
t.f == 0.0
t.next == nil
```

The same would also be true after

```
var t T
```

Program execution

A package with no imports is initialized by assigning initial values to all its package-level variables and then calling any package-level function with the name and signature of

```
func init()
```

defined in its source. A package may contain multiple `init()` functions, even within a single source file; they execute in unspecified order.

Within a package, package-level variables are initialized, and constant values are determined, in data-dependent order: if the initializer of `A` depends on the value of `B`, `A` will be set after `B`. It is an error if such dependencies form a cycle. Dependency analysis is done lexically: `A` depends on `B` if the value of `A` contains a mention of `B`, contains a value whose initializer mentions `B`, or mentions a function that mentions `B`, recursively. If two items are not interdependent, they will be initialized in the order they appear in the source. Since the dependency analysis is done per package, it can produce unspecified results if `A`'s initializer calls a function defined in another package that refers to `B`.

Initialization code may contain "go" statements, but the functions they invoke do not begin execution until initialization of the entire program is complete. Therefore, all initialization code is run in a single goroutine.

An `init()` function cannot be referred to from anywhere in a program. In particular, `init()` cannot be called explicitly, nor can a pointer to `init` be assigned to a function variable.

If a package has imports, the imported packages are initialized before initializing the package itself. If multiple packages import a package `P`, `P` will be initialized only once.

The importing of packages, by construction, guarantees that there can be no cyclic dependencies in initialization.

A complete program, possibly created by linking multiple packages, must have

one package called `main`, with a function

```
func main() { ... }
```

defined. The function `main.main()` takes no arguments and returns no value.

Program execution begins by initializing the `main` package and then invoking `main.main()`.

When `main.main()` returns, the program exits. It does not wait for other (non-`main`) goroutines to complete.

Implementation restriction: The compiler assumes package `main` is not imported by any other package.

Run-time panics

Execution errors such as attempting to index an array out of bounds trigger a *run-time panic* equivalent to a call of the built-in function `panic` with a value of the implementation-defined interface type `runtime.Error`. That type defines at least the method `String() string`. The exact error values that represent distinct run-time error conditions are unspecified, at least for now.

```
package runtime

type Error interface {
        String() string
        // and perhaps others
}
```

System considerations

Package `unsafe`

The built-in package `unsafe`, known to the compiler, provides facilities for low-level programming including operations that violate the type system. A package

using `unsafe` must be vetted manually for type safety. The package provides the following interface:

```
package unsafe

type ArbitraryType int  // shorthand for an arbitrary Go
type; it is not a real type
type Pointer *ArbitraryType

func Alignof(variable ArbitraryType) int
func Offsetof(selector ArbitraryType) int
func Sizeof(variable ArbitraryType) int

func Reflect(val interface {}) (typ runtime.Type, addr ui
ntptr)
func Typeof(val interface {}) reflect.Type
func Unreflect(typ runtime.Type, addr uintptr) interface{
}
```

Any pointer or value of type `uintptr` can be converted into a `Pointer` and vice versa.

The function `Sizeof` takes an expression denoting a variable of any type and returns the size of the variable in bytes.

The function `Offsetof` takes a selector (§Selectors) denoting a struct field of any type and returns the field offset in bytes relative to the struct's address. For a struct `s` with field `f`:

```
uintptr(unsafe.Pointer(&s)) + uintptr(unsafe.Offsetof(s.f
)) == uintptr(unsafe.Pointer(&s.f))
```

Computer architectures may require memory addresses to be *aligned*; that is, for addresses of a variable to be a multiple of a factor, the variable's type's *alignment*. The function `Alignof` takes an expression denoting a variable of any type and returns the alignment of the (type of the) variable in bytes. For a variable `x`:

```
uintptr(unsafe.Pointer(&x)) % uintptr(unsafe.Alignof(x))
== 0
```

Calls to `Alignof`, `Offsetof`, and `Sizeof` are compile-time constant expressions of type `int`.

The functions `unsafe.Typeof`, `unsafe.Reflect`, and `unsafe.Unreflect` allow access at run time to the dynamic types and values stored in interfaces. `Typeof` returns a representation of `val`'s dynamic type as a `runtime.Type`. `Reflect` allocates a copy of `val`'s dynamic value and returns both the type and the address of the copy. `Unreflect` inverts `Reflect`, creating an interface value from a type and address. The `reflect` package built on these primitives provides a safe, more convenient way to inspect interface values.

Size and alignment guarantees

For the numeric types (§Numeric types), the following sizes are guaranteed:

```
type                        size in bytes

byte, uint8, int8           1
uint16, int16               2
uint32, int32, float32      4
uint64, int64, float64      8
```

The following minimal alignment properties are guaranteed:

1. For a variable `x` of any type: `1 <= unsafe.Alignof(x) <= unsafe.Maxalign`.
2. For a variable `x` of numeric type: `unsafe.Alignof(x)` is the smaller of `unsafe.Sizeof(x)` and `unsafe.Maxalign`, but at least 1.
3. For a variable `x` of struct type: `unsafe.Alignof(x)` is the largest of all the values `unsafe.Alignof(x.f)` for each field `f` of x, but at least 1.
4. For a variable `x` of array type: `unsafe.Alignof(x)` is the same as `unsafe.Alignof(x[0])`, but at least 1.

Implementation differences - TODO

- Implementation does not honor the restriction on goto statements and targets (no intervening declarations).
- Method expressions are partially implemented.
- Gccgo: allows only one init() function per source file.
- Gccgo: Deferred functions cannot access the surrounding function's result parameters.
- Gccgo: Function results are not addressable.
- Gccgo: Recover is not implemented.
- Gccgo: The implemented version of panic differs from its specification.