The Go Programming Language Specification

Version of Aug 2, 2023

Table of Contents

Introduction

This is the reference manual for the Go programming language. The pre-Go1.18 version, without generics, can be found here. For more information and other documents, see 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 syntax is compact and simple to parse, allowing for easy analysis by automatic tools such as integrated development environments.

Notation

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

```
Syntax      = { Production } .
Production  = production_name "=" [ Expression ] "." .
Expression  = Term { "|" Term } .
Term        = Factor { Factor } .
Factor      = production_name | token [ "…" token ] | Group | 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)
```

Lowercase production names are used to identify lexical (terminal) tokens. Non-terminals are in CamelCase. Lexical tokens are enclosed in double quotes "" or back quotes ``.

The form a … b represents the set of characters from a through b as alternatives. The horizontal ellipsis … is also used elsewhere in the spec to informally denote various enumerations or code snippets that are not further specified. The character … (as opposed to the three characters `...`) is not a token of the Go language.

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 unqualified term *character* to refer to a Unicode code point in the source text.

Each code point is distinct; for instance, uppercase and lowercase letters are different characters.

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

Implementation restriction: For compatibility with other tools, a compiler may ignore a UTF-8-encoded byte order mark (U+FEFF) if it is the first Unicode code point in the source text. A byte order mark may be disallowed anywhere else in the source.

Characters

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

```
newline        = /* the Unicode code point U+000A */ .
unicode_char   = /* an arbitrary Unicode code point except newline */ .
unicode_letter = /* a Unicode code point categorized as "Letter" */ .
unicode_digit  = /* a Unicode code point categorized as "Number, decimal digit" */ .
```

In The Unicode Standard 8.0, Section 4.5 "General Category" defines a set of character categories. Go treats all characters in any of the Letter categories Lu, Ll, Lt, Lm, or Lo as Unicode letters, and those in the Number category Nd as Unicode digits.

Letters and digits

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

```
letter        = unicode_letter | "_" .
decimal_digit = "0" … "9" .
binary_digit  = "0" | "1" .
octal_digit   = "0" … "7" .
hex_digit     = "0" … "9" | "A" … "F" | "a" … "f" .
```

Lexical elements

Comments

Comments serve as program documentation. There are two forms:

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

A comment cannot start inside a rune or string literal, or inside a comment. A general comment containing no newlines acts like a space. Any other comment acts like a newline.

Tokens

Tokens form the vocabulary of the Go language. There are four classes: *identifiers*, *keywords*, *operators and punctuation*, and *literals*. *White space*, formed from spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token. Also, a newline or end of file may trigger the insertion of a semicolon. While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

Semicolons

The formal syntax 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 immediately after a line's final

token if that token is

- an identifier
- an integer, floating-point, imaginary, rune, or string literal
- one of the keywords `break`, `continue`, `fallthrough`, or `return`
- one of the operators and punctuation `++`, `--`, `)`, `]`, 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    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
```

```
continue     for          import       return       var
```

Operators and punctuation

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

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

Integer literals

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

For readability, an underscore character `_` may appear after a base prefix or between successive digits; such underscores do not change the literal's value.

```
int_lit        = decimal_lit | binary_lit | octal_lit | hex_lit .
decimal_lit    = "0" | ( "1" … "9" ) [ [ "_" ] decimal_digits ] .
binary_lit     = "0" ( "b" | "B" ) [ "_" ] binary_digits .
octal_lit      = "0" [ "o" | "O" ] [ "_" ] octal_digits .
hex_lit        = "0" ( "x" | "X" ) [ "_" ] hex_digits .

decimal_digits = decimal_digit { [ "_" ] decimal_digit } .
binary_digits  = binary_digit { [ "_" ] binary_digit } .
octal_digits   = octal_digit { [ "_" ] octal_digit } .
hex_digits     = hex_digit { [ "_" ] hex_digit } .
```

```
42
```

```
4_2
0600
0_600
0o600
0O600       // second character is capital letter 'O'
0xBadFace
0xBad_Face
0x_67_7a_2f_cc_40_c6
170141183460469231731687303715884105727
170_141183_460469_231731_687303_715884_105727

_42         // an identifier, not an integer literal
42_         // invalid: _ must separate successive digits
4__2        // invalid: only one _ at a time
0_xBadFace  // invalid: _ must separate successive digits
```

Floating-point literals

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

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

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

For readability, an underscore character `_` may appear after a base prefix or between successive digits; such underscores do not change the literal value.

```
float_lit         = decimal_float_lit | hex_float_lit .
```

```
decimal_float_lit = decimal_digits "." [ decimal_digits ] [ decimal_exponent ] |
                    decimal_digits decimal_exponent |
                    "." decimal_digits [ decimal_exponent ] .
decimal_exponent  = ( "e" | "E" ) [ "+" | "-" ] decimal_digits .

hex_float_lit     = "0" ( "x" | "X" ) hex_mantissa hex_exponent .
hex_mantissa      = [ "_" ] hex_digits "." [ hex_digits ] |
                    [ "_" ] hex_digits |
                    "." hex_digits .
hex_exponent      = ( "p" | "P" ) [ "+" | "-" ] decimal_digits .
```

```
0.
72.40
072.40       // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
1_5.         // == 15.0
0.15e+0_2    // == 15.0

0x1p-2       // == 0.25
0x2.p10      // == 2048.0
0x1.Fp+0     // == 1.9375
0X.8p-0      // == 0.5
0X_1FFFP-16  // == 0.1249847412109375
0x15e-2      // == 0x15e - 2 (integer subtraction)

0x.p1        // invalid: mantissa has no digits
1p-2         // invalid: p exponent requires hexadecimal mantissa
0x1.5e-2     // invalid: hexadecimal mantissa requires p exponent
1_.5         // invalid: _ must separate successive digits
1._5         // invalid: _ must separate successive digits
1.5_e1       // invalid: _ must separate successive digits
```

```
1.5e_1       // invalid: _ must separate successive digits
1.5e1_       // invalid: _ must separate successive digits
```

Imaginary literals

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

```
imaginary_lit = (decimal_digits | int_lit | float_lit) "i" .
```

For backward compatibility, an imaginary literal's integer part consisting entirely of decimal digits (and possibly underscores) is considered a decimal integer, even if it starts with a leading `0`.

```
0i
0123i         // == 123i for backward-compatibility
0o123i        // == 0o123 * 1i == 83i
0xabci        // == 0xabc * 1i == 2748i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i
0x1p-2i       // == 0x1p-2 * 1i == 0.25i
```

Rune literals

A rune literal represents a rune constant, an integer value identifying a Unicode code point. A rune literal is expressed as one or more characters enclosed in single quotes, as in `'x'` or `'\n'`. Within the quotes, any character may appear except newline and unescaped single quote. A single quoted character represents the Unicode value of the character 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 encoded 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 rune literals)
\"   U+0022 double quote  (valid escape only within string literals)
```

An unrecognized character following a backslash in a rune literal is illegal.

```
rune_lit         = "'" ( unicode_value | byte_value ) "'" .
unicode_value    = unicode_char | little_u_value | big_u_value | escaped_char .
byte_value       = octal_byte_value | hex_byte_value .
octal_byte_value = `\` octal_digit octal_digit octal_digit .
hex_byte_value   = `\` "x" hex_digit hex_digit .
little_u_value   = `\` "u" hex_digit hex_digit hex_digit hex_digit .
```

```
big_u_value      = `\` "U" hex_digit hex_digit hex_digit hex_digit
                           hex_digit hex_digit hex_digit hex_digit .
escaped_char     = `\` ( "a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"` ) .
```

```
'a'
'ä'
'本'
'\t'
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
'\''         // rune literal containing single quote character
'aa'         // illegal: too many characters
'\k'         // illegal: k is not recognized after a backslash
'\xa'        // illegal: too few hexadecimal digits
'\0'         // illegal: too few octal digits
'\400'       // illegal: octal value over 255
'\uDFFF'     // illegal: surrogate half
'\U00110000' // illegal: invalid Unicode code point
```

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, as in `` `foo` ``. Within the quotes, any character may appear except back quote. The value of a raw string literal is the string composed of the uninterpreted (implicitly UTF-8-encoded) characters between the quotes; in particular, backslashes have no special meaning and the string may contain newlines. Carriage return characters ('\r') inside raw string literals are discarded from the raw string value.

Interpreted string literals are character sequences between double quotes, as in `"bar"`. Within the quotes, any character may appear except newline and unescaped double quote. The text between the quotes forms the value of the literal, with backslash escapes interpreted as they are in rune literals (except that `\'` is illegal and `\"` is legal), with the same restrictions. 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_string_lit .
raw_string_lit         = "`" { unicode_char | newline } "`" .
interpreted_string_lit = `"` { unicode_value | byte_value } `"` .
```

```
`abc`                // same as "abc"
`\n
\n`                  // same as "\\n\n\\n"
"\n"
"\""                 // same as `"`
"Hello, world!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
"\uD800"             // illegal: surrogate half
"\U00110000"         // illegal: invalid Unicode code point
```

These examples all represent the same string:

```
"日本語"                                 // UTF-8 input text
`日本語`                                 // UTF-8 input text as a raw literal
"\u65e5\u672c\u8a9e"                    // the explicit Unicode code points
"\U000065e5\U0000672c\U00008a9e"        // the explicit Unicode code points
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"  // the explicit UTF-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 rune 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*, *rune constants*, *integer constants*, *floating-point constants*, *complex constants*, and *string constants*. Rune, integer, floating-point, and complex constants are collectively called *numeric constants*.

A constant value is represented by a rune, integer, floating-point, imaginary, or string literal, an identifier denoting a constant, a constant expression, a conversion with a result that is a constant, or the result value of some built-in functions such as `min` or `max` applied to constant arguments, `unsafe.Sizeof` applied to certain values, `cap` or `len` applied to some expressions, `real` and `imag` applied to a complex constant and `complex` 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 exact values of arbitrary precision and do not overflow. Consequently, there are no constants denoting the IEEE-754 negative zero, infinity, and not-a-number values.

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 statement or as an operand in an expression. It is an error if the constant value cannot be represented as a value of the respective type. If the type is a type parameter, the constant is converted into a non-constant value of the type parameter.

An untyped constant has a *default type* which is the type to which the constant is implicitly converted in contexts where a typed value is required, for instance, in a short variable declaration such as `i := 0` where there is no explicit type. The default type of an untyped constant is `bool`, `rune`, `int`, `float64`, `complex128`, or `string` respectively, depending on whether it is a boolean, rune, integer, floating-point, complex, or string constant.

Implementation restriction: Although numeric constants have arbitrary precision in the language, a compiler may implement them using an internal representation with limited precision. That said, every implementation must:

- Represent integer constants with at least 256 bits.

- Represent floating-point constants, including the parts of a complex constant, with a mantissa of at least 256 bits and a signed binary exponent of at least 16 bits.
- Give an error if unable to represent an integer constant precisely.
- Give an error if unable to represent a floating-point or complex constant due to overflow.
- Round to the nearest representable constant if unable to represent a floating-point or complex constant due to limits on precision.

These requirements apply both to literal constants and to the result of evaluating constant expressions.

Variables

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

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

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

The *static type* (or just *type*) of a variable is the type given in its declaration, the type provided in the `new` call or composite literal, or the type of an element of a structured variable. Variables of interface type also have a distinct *dynamic type*, which is the (non-interface) type of the value assigned to the variable at run time (unless the value is the predeclared identifier `nil`, which has no type). The dynamic type may vary during execution but values stored in interface variables are always assignable to the static type of the variable.

```
var x interface{}  // x is nil and has static type interface{}
var v *T           // v has value nil, static type *T
x = 42             // x has value 42 and dynamic type int
x = v              // x has value (*T)(nil) and dynamic type *T
```

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

Types

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

```
Type      = TypeName [ TypeArgs ] | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeArgs  = "[" TypeList [ "," ] "]" .
TypeList  = Type { "," Type } .
TypeLit   = ArrayType | StructType | PointerType | FunctionType | InterfaceType |
            SliceType | MapType | ChannelType .
```

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

Predeclared types, defined types, and type parameters are called *named types*. An alias denotes a named type if the type given in the alias declaration is a named 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`; it is a defined type.

Numeric types

An *integer*, *floating-point*, or *complex* type represents the set of integer, floating-point, or complex values, respectively. They are collectively called *numeric types*. The predeclared architecture-independent numeric types are:

```
uint8       the set of all unsigned  8-bit integers (0 to 255)
uint16      the set of all unsigned 16-bit integers (0 to 65535)
uint32      the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      the set of all unsigned 64-bit integers (0 to 18446744073709551615)
```

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

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

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

byte        alias for uint8
rune        alias for int32
```

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 integer types with implementation-specific sizes:

```
uint     either 32 or 64 bits
int      same size as uint
uintptr  an unsigned integer large enough to store the uninterpreted bits of a pointer value
```

To avoid portability issues all numeric types are defined types and thus distinct except `byte`, which is an alias for `uint8`, and `rune`, which is an alias for `int32`. Explicit 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. A string value is a (possibly empty) sequence of bytes. The number of bytes is called the length of the string and is never negative. Strings are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is `string`; it is a defined type.

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

A string's bytes can be accessed by integer indices 0 through `len(s)-1`. 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.

Array types

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

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

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

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

An array type `T` may not have an element of type `T`, or of a type containing `T` as a component, directly or indirectly, if those containing types are only array or struct types.

```
// invalid array types
type (
	T1 [10]T1                 // element type of T1 is T1
	T2 [10]struct{ f T2 }     // T2 contains T2 as component of a struct
	T3 [10]T4                 // T3 contains T3 as component of a struct in T4
	T4 struct{ f T3 }         // T4 contains T4 as component of array T3 in a struct
)
```

```
// valid array types
type (
	T5 [10]*T5                // T5 contains T5 as component of a pointer
	T6 [10]func() T6          // T6 contains T6 as component of a function type
	T7 [10]struct{ f []T7 }   // T7 contains T7 as component of a slice in a struct
)
```

Slice types

A slice is a descriptor for a contiguous segment of an *underlying array* and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The number of elements is called the length of the slice and is never negative. The value of an uninitialized slice is `nil`.

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

The length of a slice `s` can be discovered by the built-in function `len`; unlike with arrays it may change during execution. The elements can be addressed by integer indices 0 through `len(s)-1`. 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. 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` may be made using the built-in function `make`, which takes a slice type and parameters specifying the length and optionally the capacity. A slice created with `make` always 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 expressions are equivalent:

```
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 inner lengths may vary dynamically. Moreover, the inner slices must be initialized individually.

Struct types

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

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

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

// A struct with 6 fields.
struct {
	x, y int
	u float32
	_ float32  // padding
	A *[]int
	F func()
}
```

A field declared with a type but no explicit field name is called an *embedded field*. An embedded field must be specified as a type name `T` or as a pointer to a non-interface 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 embedded fields of types 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 embedded field *T and *P.T
	*T    // conflicts with embedded field T and *P.T
	*P.T  // conflicts with embedded field T and *T
}
```

A field or method `f` of an embedded field in a struct `x` is called *promoted* if `x.f` is a legal selector that denotes that field or method `f`.

Promoted fields act like ordinary fields of a struct except that they cannot be used as field names in composite literals of the struct.

Given a struct type `S` and a named type `T`, promoted methods are included in the method set of the struct as follows:

- If `S` contains an embedded field `T`, the method sets of `S` and `*S` both include promoted methods with receiver `T`. The method set of `*S` also includes promoted methods with receiver `*T`.
- If `S` contains an embedded field `*T`, the method sets of `S` and `*S` both include promoted methods with receiver `T` or `*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. An empty tag string is equivalent to an absent tag. The tags are made visible through a reflection interface and take part in type identity for structs but are otherwise ignored.

```
struct {
	x, y float64 ""  // an empty tag string is like an absent tag
	name string  "any string is permitted as a tag"
```

```
	_    [4]byte "ceci n'est pas un champ de structure"
}

// A struct corresponding to a TimeStamp protocol buffer.
// The tag strings define the protocol buffer field numbers;
// they follow the convention outlined by the reflect package.
struct {
	microsec  uint64 `protobuf:"1"`
	serverIP6 uint64 `protobuf:"2"`
}
```

A struct type `T` may not contain a field of type `T`, or of a type containing `T` as a component, directly or indirectly, if those containing types are only array or struct types.

```
// invalid struct types
type (
	T1 struct{ T1 }            // T1 contains a field of T1
	T2 struct{ f [10]T2 }      // T2 contains T2 as component of an array
	T3 struct{ T4 }            // T3 contains T3 as component of an array in struct T4
	T4 struct{ f [10]T3 }      // T4 contains T4 as component of struct T3 in an array
)

// valid struct types
type (
	T5 struct{ f *T5 }         // T5 contains T5 as component of a pointer
	T6 struct{ f func() T6 }   // T6 contains T6 as component of a function type
	T7 struct{ f [10][]T7 }    // T7 contains T7 as component of a slice in an array
)
```

Pointer types

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

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

```
BaseType    = Type .
```

```
*Point
*[4]int
```

Function types

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

```
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 and all non-blank names in the signature must be unique. 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.

The final incoming parameter in a function signature may have a type prefixed with `...`. A function with such a parameter is called *variadic* and may be invoked with zero or more arguments for that parameter.

```
func()
func(x int) int
func(a, _ int, z float32) bool
func(a, b int, z float32) (bool)
func(prefix string, values ...int)
func(a, b int, z float64, opt ...interface{}) (success bool)
func(int, int, float64) (float64, *[]int)
func(n int) func(p *T)
```

Interface types

An interface type defines a *type set*. A variable of interface type can store a value of any type that is in the type set of the interface. Such a type is said to implement the interface. The value of an uninitialized variable of interface type is `nil`.

```
InterfaceType  = "interface" "{" { InterfaceElem ";" } "}" .
InterfaceElem  = MethodElem | TypeElem .
MethodElem     = MethodName Signature .
MethodName     = identifier .
TypeElem       = TypeTerm { "|" TypeTerm } .
TypeTerm       = Type | UnderlyingType .
UnderlyingType = "~" Type .
```

An interface type is specified by a list of *interface elements*. An interface element is either a *method* or a *type element*, where a type element is a union of one or more *type terms*. A type term is either a single type or a single underlying type.

Basic interfaces

In its most basic form an interface specifies a (possibly empty) list of methods. The type set defined by such an interface is the set of types which implement all of those methods, and the corresponding method set consists exactly of the methods specified by the interface. Interfaces whose type sets can be defined entirely by a list of methods are called *basic interfaces.*

```
// A simple File interface.
interface {
	Read([]byte) (int, error)
	Write([]byte) (int, error)
	Close() error
}
```

The name of each explicitly specified method must be unique and not blank.

```
interface {
	String() string
	String() string  // illegal: String not unique
```

```
        _(x int)         // illegal: method must have non-blank name
}
```

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

```
func (p T) Read(p []byte) (n int, err error)
func (p T) Write(p []byte) (n int, err error)
func (p T) Close() error
```

(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.

Every type that is a member of the type set of an interface implements that interface. Any given type may implement several distinct interfaces. For instance, all types implement the *empty interface* which stands for the set of all (non-interface) types:

```
interface{}
```

For convenience, the predeclared type any is an alias for the empty interface.

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

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

If `S1` and `S2` also implement

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

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

Embedded interfaces

In a slightly more general form an interface `T` may use a (possibly qualified) interface type name `E` as an interface element. This is called *embedding* interface `E` in `T`. The type set of `T` is the *intersection* of the type sets defined by `T`'s explicitly declared methods and the type sets of `T`'s embedded interfaces. In other words, the type set of `T` is the set of all types that implement all the explicitly declared methods of `T` and also all the methods of `E`.

```
type Reader interface {
	Read(p []byte) (n int, err error)
	Close() error
}

type Writer interface {
	Write(p []byte) (n int, err error)
	Close() error
}

// ReadWriter's methods are Read, Write, and Close.
type ReadWriter interface {
	Reader  // includes methods of Reader in ReadWriter's method set
	Writer  // includes methods of Writer in ReadWriter's method set
}
```

When embedding interfaces, methods with the same names must have identical signatures.

```
type ReadCloser interface {
	Reader   // includes methods of Reader in ReadCloser's method set
	Close()  // illegal: signatures of Reader.Close and Close are different
}
```

General interfaces

In their most general form, an interface element may also be an arbitrary type term `T`, or a term of the form `~T` specifying the underlying type `T`, or a union of terms $t_1|t_2|…|t_n$. Together with method specifications, these elements enable the precise definition of an interface's type set as follows:

- The type set of the empty interface is the set of all non-interface types.
- The type set of a non-empty interface is the intersection of the type sets of its interface elements.
- The type set of a method specification is the set of all non-interface types whose method sets include that method.
- The type set of a non-interface type term is the set consisting of just that type.
- The type set of a term of the form `~T` is the set of all types whose underlying type is `T`.
- The type set of a *union* of terms $t_1|t_2|\ldots|t_n$ is the union of the type sets of the terms.

The quantification "the set of all non-interface types" refers not just to all (non-interface) types declared in the program at hand, but all possible types in all possible programs, and hence is infinite. Similarly, given the set of all non-interface types that implement a particular method, the intersection of the method sets of those types will contain exactly that method, even if all types in the program at hand always pair that method with another method.

By construction, an interface's type set never contains an interface type.

```
// An interface representing only the type int.
interface {
	int
}

// An interface representing all types with underlying type int.
interface {
	~int
}

// An interface representing all types with underlying type int that implement the String method.
interface {
	~int
	String() string
}

// An interface representing an empty type set: there is no type that is both an int and a string.
interface {
```

```
		int
		string
	}
```

In a term of the form `~T`, the underlying type of `T` must be itself, and `T` cannot be an interface.

```
	type MyInt int

	interface {
		~[]byte  // the underlying type of []byte is itself
		~MyInt   // illegal: the underlying type of MyInt is not MyInt
		~error   // illegal: error is an interface
	}
```

Union elements denote unions of type sets:

```
	// The Float interface represents all floating-point types
	// (including any named types whose underlying types are
	// either float32 or float64).
	type Float interface {
		~float32 | ~float64
	}
```

The type `T` in a term of the form `T` or `~T` cannot be a type parameter, and the type sets of all non-interface terms must be pairwise disjoint (the pairwise intersection of the type sets must be empty). Given a type parameter `P`:

```
	interface {
		P                // illegal: P is a type parameter
		int | ~P         // illegal: P is a type parameter
		~int | MyInt     // illegal: the type sets for ~int and MyInt are not disjoint (~int includes MyInt)
		float32 | Float  // overlapping type sets but Float is an interface
	}
```

Implementation restriction: A union (with more than one term) cannot contain the predeclared identifier `comparable` or interfaces that specify methods, or embed `comparable` or interfaces that specify methods.

Interfaces that are not basic may only be used as type constraints, or as elements of other interfaces used as constraints. They cannot be the types of values or variables, or components of other, non-interface types.

```
var x Float                     // illegal: Float is not a basic interface

var x interface{} = Float(nil)  // illegal

type Floatish struct {
	f Float                 // illegal
}
```

An interface type `T` may not embed a type element that is, contains, or embeds `T`, directly or indirectly.

```
// illegal: Bad may not embed itself
type Bad interface {
	Bad
}

// illegal: Bad1 may not embed itself using Bad2
type Bad1 interface {
	Bad2
}
type Bad2 interface {
	Bad1
}

// illegal: Bad3 may not embed a union containing Bad3
type Bad3 interface {
	~int | ~string | Bad3
}

// illegal: Bad4 may not embed an array containing Bad4 as element type
type Bad4 interface {
```

```
        [10]Bad4
}
```

Implementing an interface

A type `T` implements an interface `I` if

- `T` is not an interface and is an element of the type set of `I`; or
- `T` is an interface and the type set of `T` is a subset of the type set of `I`.

A value of type `T` implements an interface if `T` implements the interface.

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 `!=` must be fully defined for operands of the key type; thus the key type must not be a function, map, 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 float64 }
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` and may change during execution. Elements may be added during execution using assignments and retrieved with index expressions; they may be removed with the `delete` and `clear` built-in function.

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, with the exception of `nil` maps. A `nil` map is equivalent to an empty map except that no elements may be added.

Channel types

A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. The value of an uninitialized channel is `nil`.

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

The optional `<-` operator specifies the channel *direction*, *send* or *receive*. If a direction is given, the channel is *directional*, otherwise it is *bidirectional*. A channel may be constrained only to send or only to receive by assignment or explicit conversion.

```
chan T          // can be used to send and receive values of type T
chan<- float64  // can only be used to send float64s
<-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 zero or absent, the channel is unbuffered and communication succeeds only when both a sender and receiver are ready. Otherwise, the channel is buffered and communication succeeds without blocking if the buffer is not full (sends) or not empty (receives). A `nil` channel is never ready for communication.

A channel may be closed with the built-in function `close`. The multi-valued assignment form of the receive operator reports whether a received value was sent before the channel was closed.

A single channel may be used in send statements, receive operations, and calls to the built-in functions `cap` and `len` by any number of goroutines without further synchronization. Channels act as first-in-first-out queues. For example, if one goroutine sends values on a channel and a second goroutine receives them, the values are received in the order sent.

Properties of types and values

Underlying types

Each type `T` has an *underlying type*: If `T` is one of the predeclared boolean, numeric, or string types, 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 declaration. For a type parameter that is the underlying type of its type constraint, which is always an interface.

```
type (
	A1 = string
	A2 = A1
)

type (
	B1 string
	B2 B1
	B3 []B1
	B4 B3
)

func f[P any](x P) { … }
```

The underlying type of `string`, `A1`, `A2`, `B1`, and `B2` is `string`. The underlying type of `[]B1`, `B3`, and `B4` is `[]B1`. The underlying type of `P` is `interface{}`.

Core types

Each non-interface type `T` has a *core type*, which is the same as the underlying type of `T`.

An interface `T` has a core type if one of the following conditions is satisfied:

1. There is a single type `U` which is the underlying type of all types in the type set of `T`; or
2. the type set of `T` contains only channel types with identical element type `E`, and all directional channels have the same direction.

No other interfaces have a core type.

The core type of an interface is, depending on the condition that is satisfied, either:

1. the type `U`; or
2. the type `chan E` if `T` contains only bidirectional channels, or the type `chan<- E` or `<-chan E` depending on the direction of the directional channels present.

By definition, a core type is never a defined type, type parameter, or interface type.

Examples of interfaces with core types:

```
type Celsius float32
type Kelvin  float32

interface{ int }                          // int
interface{ Celsius|Kelvin }               // float32
interface{ ~chan int }                    // chan int
interface{ ~chan int|~chan<- int }        // chan<- int
interface{ ~[]*data; String() string }    // []*data
```

Examples of interfaces without core types:

```
interface{}                               // no single underlying type
interface{ Celsius|float64 }              // no single underlying type
interface{ chan int | chan<- string }     // channels have different element types
interface{ <-chan int | chan<- int }      // directional channels have different directions
```

Some operations (slice expressions, `append` and `copy`) rely on a slightly more loose form of core types which accept byte slices and strings. Specifically, if there are exactly two types, `[]byte` and `string`, which are the underlying types of all types in the type set of interface `T`, the core type of `T` is called `bytestring`.

Examples of interfaces with `bytestring` core types:

```
interface{ int }                          // int (same as ordinary core type)
interface{ []byte | string }              // bytestring
interface{ ~[]byte | myString }           // bytestring
```

Note that `bytestring` is not a real type; it cannot be used to declare variables or compose other types. It exists solely to describe the behavior of some operations that read from a sequence of bytes, which may be a byte slice or a string.

Type identity

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

A named type is always different from any other type. Otherwise, two types are identical if their underlying type literals are structurally equivalent; that is, 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. Non-exported 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 define the same type set.
- Two map types are identical if they have identical key and element types.
- Two channel types are identical if they have identical element types and the same direction.
- Two instantiated types are identical if their defined types and all type arguments are identical.

Given the declarations

```
type (
	A0 = []string
	A1 = A0
	A2 = struct{ a, b int }
	A3 = int
	A4 = func(A3, float64) *A0
	A5 = func(x int, _ float64) *[]string

	B0 A0
	B1 []string
	B2 struct{ a, b int }
	B3 struct{ a, c int }
	B4 func(int, float64) *B0
	B5 func(x int, y float64) *A1

	C0 = B0
	D0[P1, P2 any] struct{ x P1; y P2 }
	E0 = D0[int, string]
)
```

these types are identical:

```
A0, A1, and []string
A2 and struct{ a, b int }
A3 and int
A4, func(int, float64) *[]string, and A5

B0 and C0
```

```
D0[int, string] and E0
[]int and []int
struct{ a, b *B5 } and struct{ a, b *B5 }
func(x int, y float64) *[]string, func(int, float64) (result *[]string), and A5
```

B0 and B1 are different because they are new types created by distinct type definitions; func(int, float64) *B0 and func(x int, y float64) *[]string are different because B0 is different from []string; and P1 and P2 are different because they are different type parameters. D0[int, string] and struct{ x int; y string } are different because the former is an instantiated defined type while the latter is a type literal (but they are still assignable).

Assignability

A value x of type V is *assignable* to a variable of type T ("x is assignable to T") if one of the following conditions applies:

- V and T are identical.
- V and T have identical underlying types but are not type parameters and at least one of V or T is not a named type.
- V and T are channel types with identical element types, V is a bidirectional channel, and at least one of V or T is not a named type.
- T is an interface type, but not a type parameter, and x implements T.
- x is the predeclared identifier nil and T is a pointer, function, slice, map, channel, or interface type, but not a type parameter.
- x is an untyped constant representable by a value of type T.

Additionally, if x's type V or T are type parameters, x is assignable to a variable of type T if one of the following conditions applies:

- x is the predeclared identifier nil, T is a type parameter, and x is assignable to each type in T's type set.
- V is not a named type, T is a type parameter, and x is assignable to each type in T's type set.
- V is a type parameter and T is not a named type, and values of each type in V's type set are assignable to T.

Representability

A constant x is *representable* by a value of type T, where T is not a type parameter, if one of the following conditions applies:

- x is in the set of values determined by T.

- `T` is a floating-point type and `x` can be rounded to `T`'s precision without overflow. Rounding uses IEEE 754 round-to-even rules but with an IEEE negative zero further simplified to an unsigned zero. Note that constant values never result in an IEEE negative zero, NaN, or infinity.
- `T` is a complex type, and `x`'s components `real(x)` and `imag(x)` are representable by values of `T`'s component type (`float32` or `float64`).

If `T` is a type parameter, `x` is representable by a value of type `T` if `x` is representable by a value of each type in `T`'s type set.

```
x                   T           x is representable by a value of T because

'a'                 byte        97 is in the set of byte values
97                  rune        rune is an alias for int32, and 97 is in the set of 32-bit integers
"foo"               string      "foo" is in the set of string values
1024                int16       1024 is in the set of 16-bit integers
42.0                byte        42 is in the set of unsigned 8-bit integers
1e10                uint64      10000000000 is in the set of unsigned 64-bit integers
2.718281828459045   float32     2.718281828459045 rounds to 2.7182817 which is in the set of float32 values
-1e-1000            float64     -1e-1000 rounds to IEEE -0.0 which is further simplified to 0.0
0i                  int         0 is an integer value
(42 + 0i)           float32     42.0 (with zero imaginary part) is in the set of float32 values
```

```
x                   T           x is not representable by a value of T because

0                   bool        0 is not in the set of boolean values
'a'                 string      'a' is a rune, it is not in the set of string values
1024                byte        1024 is not in the set of unsigned 8-bit integers
-1                  uint16      -1 is not in the set of unsigned 16-bit integers
1.1                 int         1.1 is not an integer value
42i                 float32     (0 + 42i) is not in the set of float32 values
1e1000              float64     1e1000 overflows to IEEE +Inf after rounding
```

Method sets

The *method set* of a type determines the methods that can be called on an operand of that type. Every type has a (possibly empty) method set associated with it:

- The method set of a defined type T consists of all methods declared with receiver type T.
- The method set of a pointer to a defined type T (where T is neither a pointer nor an interface) is the set of all methods declared with receiver *T or T.
- The method set of an interface type is the intersection of the method sets of each type in the interface's type set (the resulting method set is usually just the set of declared methods in the interface).

Further rules apply to structs (and pointer to structs) containing embedded fields, as described in the section on struct types. Any other type has an empty method set.

In a method set, each method must have a unique non-blank method name.

Blocks

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

```
Block = "{" StatementList "}" .
StatementList = { 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, type parameter, variable, function, label, or package. Every identifier in a program must be declared. No identifier may be declared twice in the same block, and no identifier may be declared in both the file and package block.

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

```
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, label, or package.

Go is lexically scoped using blocks:

1. The scope of a predeclared identifier is the universe block.
2. The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at top level (outside any function) is the package block.
3. The scope of the package name of an imported package is the file block of the file containing the import declaration.
4. The scope of an identifier denoting a method receiver, function parameter, or result variable is the function body.
5. The scope of an identifier denoting a type parameter of a function or declared by a method receiver begins after the name of the function and ends at the end of the function body.
6. The scope of an identifier denoting a type parameter of a type begins after the name of the type and ends at the end of the TypeSpec.
7. The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec (ShortVarDecl for short variable declarations) and ends at the end of the innermost containing block.
8. 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. It is illegal to define a label that is never used. 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.

Blank identifier

The *blank identifier* is represented by the underscore character _. It serves as an anonymous placeholder instead of a regular (non-blank) identifier and has special meaning in declarations, as an operand, and in assignment statements.

Predeclared identifiers

The following identifiers are implicitly declared in the universe block:

```
Types:
	any bool byte comparable
	complex64 complex128 error float32 float64
	int int8 int16 int32 int64 rune string
	uint uint8 uint16 uint32 uint64 uintptr

Constants:
	true false iota

Zero value:
	nil

Functions:
	append cap clear close complex copy delete imag len
	make max min new panic print println real recover
```

Exported identifiers

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

1. the first character of the identifier's name is a Unicode uppercase letter (Unicode character category Lu); and
2. the identifier is declared in the package block or it is a field name or method name.

All other identifiers are not exported.

Uniqueness of identifiers

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

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 ] "=" ExpressionList ] .

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, which must not be a type parameter. 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 constant
```

```
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 float32 = 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 ConstSpec. 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. Its value is the index of the respective ConstSpec in that constant declaration, starting at zero. It can be used to construct a set of related constants:

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

```
const (
	a = 1 << iota  // a == 1  (iota == 0)
	b = 1 << iota  // b == 2  (iota == 1)
	c = 3          // c == 3  (iota == 2, unused)
	d = 1 << iota  // d == 8  (iota == 3)
)

const (
	u         = iota * 42  // u == 0     (untyped integer constant)
	v float64 = iota * 42  // v == 42.0  (float64 constant)
	w         = iota * 42  // w == 84    (untyped integer constant)
)

const x = iota  // x == 0
const y = iota  // y == 0
```

By definition, multiple uses of `iota` in the same ConstSpec all have the same value:

```
const (
	bit0, mask0 = 1 << iota, 1<<iota - 1  // bit0 == 1, mask0 == 0  (iota == 0)
	bit1, mask1                           // bit1 == 2, mask1 == 1  (iota == 1)
	_, _                                  //                        (iota == 2, unused)
	bit3, mask3                           // bit3 == 8, mask3 == 7  (iota == 3)
)
```

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 type. Type declarations come in two forms: alias declarations and type definitions.

```
TypeDecl = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec = AliasDecl | TypeDef .
```

Alias declarations

An alias declaration binds an identifier to the given type.

```
AliasDecl = identifier "=" Type .
```

Within the scope of the identifier, it serves as an *alias* for the type.

```
type (
	nodeList = []*Node  // nodeList and []*Node are identical types
	Polar    = polar    // Polar and polar denote identical types
)
```

Type definitions

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

```
TypeDef = identifier [ TypeParameters ] Type .
```

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

```
type (
	Point struct{ x, y float64 }  // Point and struct{ x, y float64 } are different types
	polar Point                   // polar and Point denote different types
)

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

type Block interface {
	BlockSize() int
```

```
		Encrypt(src, dst []byte)
		Decrypt(src, dst []byte)
	}
```

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

```
	// A Mutex is a data type with two methods, Lock and Unlock.
	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 method set is empty.
	type NewMutex Mutex

	// The method set of PtrMutex's underlying type *Mutex remains unchanged,
	// but the method set of PtrMutex is empty.
	type PtrMutex *Mutex

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

	// MyBlock is an interface type that has the same method set as Block.
	type MyBlock Block
```

Type definitions may be used to define different boolean, numeric, or string types and associate methods with them:

```
	type TimeZone int

	const (
		EST TimeZone = -(5 + iota)
		CST
```

```
	MST
	PST
)

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

If the type definition specifies type parameters, the type name denotes a *generic type*. Generic types must be instantiated when they are used.

```
type List[T any] struct {
	next  *List[T]
	value T
}
```

In a type definition the given type cannot be a type parameter.

```
type T[P any] P    // illegal: P is a type parameter

func f[T any]() {
	type L T   // illegal: T is a type parameter declared by the enclosing function
}
```

A generic type may also have methods associated with it. In this case, the method receivers must declare the same number of type parameters as present in the generic type definition.

```
// The method Len returns the number of elements in the linked list l.
func (l *List[T]) Len() int  { … }
```

Type parameter declarations

A type parameter list declares the *type parameters* of a generic function or type declaration. The type parameter list looks like an ordinary function parameter list except that the type parameter names must all be present and the list is enclosed in square brackets rather than parentheses.

```
TypeParameters  = "[" TypeParamList [ "," ] "]" .
TypeParamList   = TypeParamDecl { "," TypeParamDecl } .
TypeParamDecl   = IdentifierList TypeConstraint .
```

All non-blank names in the list must be unique. Each name declares a type parameter, which is a new and different named type that acts as a placeholder for an (as of yet) unknown type in the declaration. The type parameter is replaced with a *type argument* upon instantiation of the generic function or type.

```
[P any]
[S interface{ ~[]byte|string }]
[S ~[]E, E any]
[P Constraint[int]]
[_ any]
```

Just as each ordinary function parameter has a parameter type, each type parameter has a corresponding (meta-)type which is called its *type constraint*.

A parsing ambiguity arises when the type parameter list for a generic type declares a single type parameter `P` with a constraint `C` such that the text `P C` forms a valid expression:

```
type T[P *C] …
type T[P (C)] …
type T[P *C|Q] …
…
```

In these rare cases, the type parameter list is indistinguishable from an expression and the type declaration is parsed as an array type declaration. To resolve the ambiguity, embed the constraint in an interface or use a trailing comma:

```
type T[P interface{*C}] …
type T[P *C,] …
```

Type parameters may also be declared by the receiver specification of a method declaration associated with a generic type.

Within a type parameter list of a generic type `T`, a type constraint may not (directly, or indirectly through the type parameter list of another generic type) refer to `T`.

```
type T1[P T1[P]] …                    // illegal: T1 refers to itself
type T2[P interface{ T2[int] }] …     // illegal: T2 refers to itself
type T3[P interface{ m(T3[int])}] …   // illegal: T3 refers to itself
type T4[P T5[P]] …                    // illegal: T4 refers to T5 and
type T5[P T4[P]] …                    //          T5 refers to T4

type T6[P int] struct{ f *T6[P] }     // ok: reference to T6 is not in type parameter list
```

Type constraints

A *type constraint* is an interface that defines the set of permissible type arguments for the respective type parameter and controls the operations supported by values of that type parameter.

```
TypeConstraint = TypeElem .
```

If the constraint is an interface literal of the form `interface{E}` where `E` is an embedded type element (not a method), in a type parameter list the enclosing `interface{ … }` may be omitted for convenience:

```
[T []P]                      // = [T interface{[]P}]
[T ~int]                     // = [T interface{~int}]
[T int|string]               // = [T interface{int|string}]
type Constraint ~int         // illegal: ~int is not in a type parameter list
```

The predeclared interface type `comparable` denotes the set of all non-interface types that are strictly comparable.

Even though interfaces that are not type parameters are comparable, they are not strictly comparable and therefore they do not implement `comparable`. However, they satisfy `comparable`.

```
int                          // implements comparable (int is strictly comparable)
[]byte                       // does not implement comparable (slices cannot be compared)
interface{}                  // does not implement comparable (see above)
interface{ ~int | ~string }  // type parameter only: implements comparable (int, string types are strictly comparable)
interface{ comparable }      // type parameter only: implements comparable (comparable implements itself)
interface{ ~int | ~[]byte }  // type parameter only: does not implement comparable (slices are not comparable)
interface{ ~struct{ any } }  // type parameter only: does not implement comparable (field any is not strictly comparable)
```

The `comparable` interface and interfaces that (directly or indirectly) embed `comparable` may only be used as type constraints. They cannot be the types of values or variables, or components of other, non-interface types.

Satisfying a type constraint

A type argument `T` *satisfies* a type constraint `C` if `T` is an element of the type set defined by `C`; i.e., if `T` implements `C`. As an exception, a strictly comparable type constraint may also be satisfied by a comparable (not necessarily strictly comparable) type argument. More precisely:

A type T *satisfies* a constraint `C` if

- `T` implements `C`; or
- `C` can be written in the form `interface{ comparable; E }`, where `E` is a basic interface and `T` is comparable and implements `E`.

```
type argument      type constraint                // constraint satisfaction

int                interface{ ~int }              // satisfied: int implements interface{ ~int }
string             comparable                     // satisfied: string implements comparable (string is strictly comparable)
[]byte             comparable                     // not satisfied: slices are not comparable
any                interface{ comparable; int }   // not satisfied: any does not implement interface{ int }
any                comparable                     // satisfied: any is comparable and implements the basic interface any
struct{f any}      comparable                     // satisfied: struct{f any} is comparable and implements the basic interface any
any                interface{ comparable; m() }   // not satisfied: any does not implement the basic interface interface{ m() }
```

```
interface{ m() }   interface{ comparable; m() }   // satisfied: interface{ m() } is comparable and implements the basic interface :
```

Because of the exception in the constraint satisfaction rule, comparing operands of type parameter type may panic at run-time (even though comparable type parameters are always strictly comparable).

Variable declarations

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

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

```
var i int
var U, V, W float64
var k = 0
var x, y float32 = -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 interested in "found"
```

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

If a type is present, each variable is given that type. Otherwise, each variable is given the type of the corresponding initialization value in the assignment. If that value is an untyped constant, it is first implicitly converted to its default type; if it is an untyped boolean value, it is first implicitly converted to type `bool`. The predeclared value `nil` cannot be used to initialize a variable with no explicit type.

```
var d = math.Sin(0.5)  // d is float64
var i = 42             // i is int
var t, ok = x.(T)      // t is T, ok is bool
var n = nil            // illegal
```

Implementation restriction: A compiler may make it illegal to declare a variable inside a function body if the variable is never used.

Short variable declarations

A *short variable declaration* uses the syntax:

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

It is 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()  // os.Pipe() returns a connected pair of Files and an error, if any
_, y, _ := coord(p)   // coord() returns three values; only interested in y coordinate
```

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

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // redeclares offset
x, y, x := 1, 2, 3                        // illegal: x repeated on left side of :=
```

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.

Function declarations

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

```
FunctionDecl = "func" FunctionName [ TypeParameters ] Signature [ FunctionBody ] .
FunctionName = identifier .
FunctionBody = Block .
```

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

```
func IndexRune(s string, r rune) int {
	for i, c := range s {
		if c == r {
			return i
		}
	}
	// invalid: missing return statement
}
```

If the function declaration specifies type parameters, the function name denotes a *generic function*. A generic function must be instantiated before it can be called or used as a value.

```
func min[T ~int|~float64](x, y T) T {
	if x < y {
		return x
	}
	return y
}
```

A function declaration without type parameters may omit the body. Such a declaration provides the signature for a function implemented outside Go, such as an assembly routine.

```
func flushICache(begin, end uintptr)  // implemented externally
```

Method declarations

A method is a function with a *receiver*. A method declaration binds an identifier, the *method name*, to a method, and associates the method with the receiver's *base type*.

```
MethodDecl = "func" Receiver MethodName Signature [ FunctionBody ] .
Receiver   = Parameters .
```

The receiver is specified via an extra parameter section preceding the method name. That parameter section must declare a single non-variadic parameter, the receiver. Its type must be a defined type `T` or a pointer to a defined type `T`, possibly followed by a list of type parameter names `[P1, P2, …]` enclosed in square brackets. `T` is called the receiver *base type*. A receiver base type cannot be a pointer or interface type and it must be defined in the same package as the method. The method is said to be *bound* to its receiver base type and the method name is visible only within selectors for type `T` or `*T`.

A non-blank receiver identifier must be unique in the method signature. 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.

For a base type, the non-blank names of methods bound to it must be unique. If the base type is a struct type, the non-blank method and field names must be distinct.

Given defined type `Point` the declarations

```
func (p *Point) Length() float64 {
	return math.Sqrt(p.x * p.x + p.y * p.y)
}

func (p *Point) Scale(factor float64) {
	p.x *= factor
	p.y *= factor
}
```

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

If the receiver base type is a generic type, the receiver specification must declare corresponding type parameters for the method to use. This makes the receiver type parameters available to the method. Syntactically, this type parameter declaration looks like an instantiation of the receiver base type: the type arguments must be identifiers denoting the type parameters being declared, one for each type parameter of the receiver base type. The type parameter names do not need to match their corresponding parameter names in the receiver base type definition, and all non-blank parameter names must be unique in the receiver parameter section and the method signature. The receiver type parameter constraints are implied by the receiver base type definition: corresponding type parameters have corresponding constraints.

```
type Pair[A, B any] struct {
	a A
	b B
}

func (p Pair[A, B]) Swap() Pair[B, A]  { … }  // receiver declares A, B
func (p Pair[First, _]) First() First  { … }  // receiver declares First, corresponds to A in Pair
```

Expressions

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

Operands

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

```
Operand     = Literal | OperandName [ TypeArgs ] | "(" Expression ")" .
Literal     = BasicLit | CompositeLit | FunctionLit .
BasicLit    = int_lit | float_lit | imaginary_lit | rune_lit | string_lit .
OperandName = identifier | QualifiedIdent .
```

An operand name denoting a generic function may be followed by a list of type arguments; the resulting operand is an instantiated function.

The blank identifier may appear as an operand only on the left-hand side of an assignment statement.

Implementation restriction: A compiler need not report an error if an operand's type is a type parameter with an empty type set. Functions with such type parameters cannot be instantiated; any attempt will lead to an error at the instantiation site.

Qualified identifiers

A *qualified identifier* is an identifier qualified with a package name prefix. Both the package name and the identifier must not be blank.

```
QualifiedIdent = PackageName "." identifier .
```

A qualified identifier accesses an identifier in a different package, which must be imported. The identifier must be exported and declared in the package block of that package.

```
math.Sin // denotes the Sin function in package math
```

Composite literals

Composite literals construct new composite values each time they are evaluated. They consist of the type of the literal followed by a brace-bound list of elements. Each element may optionally be preceded by a corresponding key.

```
CompositeLit  = LiteralType LiteralValue .
LiteralType   = StructType | ArrayType | "[" "..." "]" ElementType |
                SliceType | MapType | TypeName [ TypeArgs ] .
LiteralValue  = "{" [ ElementList [ "," ] ] "}" .
ElementList   = KeyedElement { "," KeyedElement } .
KeyedElement  = [ Key ":" ] Element .
Key           = FieldName | Expression | LiteralValue .
FieldName     = identifier .
Element       = Expression | LiteralValue .
```

The LiteralType's core type `T` must be a struct, array, slice, or map type (the syntax enforces this constraint except when the type is given as a TypeName). The types of the elements and keys must be assignable to the respective field, element, and key types of type `T`; there is no additional conversion. The key is interpreted as a field name for struct literals, an index 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 non-constant map keys, see the section on evaluation order.

For struct literals the following rules apply:

- A key must be a field name declared in the struct type.
- An element list 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.
- An element list 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 Point3D struct { x, y, z float64 }
type Line struct { p, q Point3D }
```

one may write

```
origin := Point3D{}                            // zero value for Point3D
line := Line{origin, Point3D{y: -4, z: 12.3}}  // zero value 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 non-negative constant representable by a value of type `int`; and if it is typed it must be of integer type.
- 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 generates a pointer to a unique variable initialized with the literal's value.

```
var pointer *Point3D = &Point3D{y: 1000}
```

Note that the zero value for a slice or map type is not the same as an initialized but empty value of the same type. Consequently, taking the address of an empty slice or map composite literal does not have the same effect as allocating a new slice or map value with new.

```
p1 := &[]int{}    // p1 points to an initialized, empty slice with value []int{} and length 0
p2 := new([]int)  // p2 points to an uninitialized slice with value nil and length 0
```

The length of an array literal is the length specified in the literal type. 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 shorthand for a slice operation applied to an array:

```
tmp := [n]T{x1, x2, … xn}
tmp[0 : n]
```

Within a composite literal of array, slice, or map type `T`, elements or map keys that are themselves composite literals may elide the respective literal type if it is identical to the element or key type of `T`. Similarly, elements or keys that are addresses of composite literals may elide the `&T` when the element or key type is `*T`.

```
[...]Point{{1.5, -3.5}, {0, 0}}     // same as [...]Point{Point{1.5, -3.5}, Point{0, 0}}
[][]int{{1, 2, 3}, {4, 5}}          // same as [][]int{[]int{1, 2, 3}, []int{4, 5}}
[][]Point{{{0, 1}, {1, 2}}}         // same as [][]Point{[]Point{Point{0, 1}, Point{1, 2}}}
map[string]Point{"orig": {0, 0}}    // same as map[string]Point{"orig": Point{0, 0}}
map[Point]string{{0, 0}: "orig"}    // same as map[Point]string{Point{0, 0}: "orig"}

type PPoint *Point
[2]*Point{{1.5, -3.5}, {}}          // same as [2]*Point{&Point{1.5, -3.5}, &Point{}}
[2]PPoint{{1.5, -3.5}, {}}          // same as [2]PPoint{PPoint(&Point{1.5, -3.5}), PPoint(&Point{})}
```

A parsing ambiguity arises when a composite literal using the TypeName form of the LiteralType appears as an operand between the keyword and the opening brace of the block of an "if", "for", or "switch" statement, and the composite literal is not enclosed in parentheses, square brackets, or curly braces. In this rare case, the opening brace of the literal is erroneously parsed as the one introducing the block of statements. To resolve the ambiguity, 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, 2147483647}

// 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]float32{-1, 0, 0, 0, -0.1, -0.1, 0, 0, 0, -1}
filter := [10]float32{-1, 4: -0.1, -0.1, 9: -1}

// frequencies in Hz for equal-tempered scale (A4 = 440Hz)
noteFrequency := map[string]float32{
	"C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
	"G0": 24.50, "A0": 27.50, "B0": 30.87,
}
```

Function literals

A function literal represents an anonymous function. Function literals cannot declare type parameters.

```
FunctionLit = "func" Signature FunctionBody .
```

```
func(a, b int, z float64) 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 }(replyChan)
```

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 |
        MethodExpr |
        PrimaryExpr Selector |
        PrimaryExpr Index |
        PrimaryExpr Slice |
        PrimaryExpr TypeAssertion |
        PrimaryExpr Arguments .

Selector       = "." identifier .
Index          = "[" Expression [ "," ] "]" .
Slice          = "[" [ Expression ] ":" [ Expression ] "]" |
                 "[" [ Expression ] ":" Expression ":" Expression "]" .
```

```
TypeAssertion  = "." "(" Type ")" .
Arguments      = "(" [ ( ExpressionList | Type [ "," ExpressionList ] ) [ "..." ] [ "," ] ] ")" .
```

```
x
2
(s + ".txt")
f(3.1415, true)
Point{1, 2}
m["foo"]
s[i : j + 1]
obj.color
f.p[i].x()
```

Selectors

For a primary expression `x` that is not a package name, the *selector expression*

```
x.f
```

denotes the field or method `f` of the value `x` (or sometimes `*x`; see below). The identifier `f` is called the (field or method) *selector*; it must not be the blank identifier. The type of the selector expression is the type of `f`. If `x` is a package name, see the section on qualified identifiers.

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 embedded field of `T`. The number of embedded 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 embedded 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 a pointer or 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 value `x` of type `I` where `I` is an interface type, `x.f` denotes the actual method with name `f` of the dynamic value of `x`. If there is no method with name `f` in the method set of `I`, the selector expression is illegal.

3. As an exception, if the type of x is a defined pointer type and `(*x).f` is a valid selector expression denoting a field (but not a method), `x.f` is shorthand for `(*x).f`.
4. In all other cases, `x.f` is illegal.
5. If x is of pointer type and has the value `nil` and `x.f` denotes a struct field, assigning to or evaluating `x.f` causes a run-time panic.
6. If x is of interface type and has the value `nil`, calling or evaluating the method `x.f` causes a run-time panic.

For example, given the declarations:

```
type T0 struct {
        x int
}

func (*T0) M0()

type T1 struct {
        y int
}

func (T1) M1()

type T2 struct {
        z int
        T1
        *T0
}

func (*T2) M2()

type Q *T2

var t T2     // with t.T0 != nil
var p *T2    // with p != nil and (*p).T0 != nil
var q Q = p
```

one may write:

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

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

q.x          // (*(*q).T0).x        (*q).x is a valid field selector

p.M0()       // ((*p).T0).M0()      M0 expects *T0 receiver
p.M1()       // ((*p).T1).M1()      M1 expects T1 receiver
p.M2()       // p.M2()              M2 expects *T2 receiver
t.M2()       // (&t).M2()           M2 expects *T2 receiver, see section on Calls
```

but the following is invalid:

```
q.M0()       // (*q).M0 is valid but not a field selector
```

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  = Type .
```

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 receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer receiver
```

```
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 five invocations are equivalent:

```
t.Mv(7)
T.Mv(t, 7)
(T).Mv(t, 7)
f1 := T.Mv; f1(t, 7)
f2 := (T).Mv; f2(t, 7)
```

Similarly, the expression

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

yields a function value representing `Mp` with signature

```
func(tp *T, f float32) float32
```

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 function literal or method value.

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.

Method values

If the expression `x` has static type `T` and `M` is in the method set of type `T`, `x.M` is called a *method value*. The method value `x.M` is a function value that is callable with the same arguments as a method call of `x.M`. The expression `x` is evaluated and saved during the evaluation of the method value; the saved copy is then used as the receiver in any calls, which may be executed later.

```
type S struct { *T }
type T int
func (t T) M() { print(t) }

t := new(T)
s := S{T: t}
f := t.M                    // receiver *t is evaluated and stored in f
g := s.M                    // receiver *(s.T) is evaluated and stored in g
*t = 42                     // does not affect stored receivers in f and g
```

The type `T` may be an interface or non-interface type.

As in the discussion of method expressions above, 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 receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer receiver

var t T
var pt *T
func makeT() T
```

The expression

```
t.Mv
```

yields a function value of type

```
func(int) int
```

These two invocations are equivalent:

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

Similarly, the expression

```
pt.Mp
```

yields a function value of type

```
func(float32) float32
```

As with selectors, a reference to a non-interface method with a value receiver using a pointer will automatically dereference that pointer: `pt.Mv` is equivalent to `(*pt).Mv`.

As with method calls, a reference to a non-interface method with a pointer receiver using an addressable value will automatically take the address of that value: `t.Mp` is equivalent to `(&t).Mp`.

```
f := t.Mv; f(7)   // like t.Mv(7)
f := pt.Mp; f(7)  // like pt.Mp(7)
f := pt.Mv; f(7)  // like (*pt).Mv(7)
f := t.Mp; f(7)   // like (&t).Mp(7)
f := makeT().Mp   // invalid: result of makeT() is not addressable
```

Although the examples above use non-interface types, it is also legal to create a method value from a value of interface type.

```
var i interface { M(int) } = myVal
f := i.M; f(7)  // like i.M(7)
```

Index expressions

A primary expression of the form

```
a[x]
```

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

If `a` is neither a map nor a type parameter:

- the index `x` must be an untyped constant or its core type must be an integer
- a constant index must be non-negative and representable by a value of type `int`
- a constant index that is untyped is given type `int`
- the index `x` is *in range* if `0 <= x < len(a)`, otherwise it is *out of range*

For a of array type A:

- a constant index must be in range
- if x is out of range at run time, a run-time panic occurs
- a[x] is the array element at index x and the type of a[x] is the element type of A

For a of pointer to array type:

- a[x] is shorthand for (*a)[x]

For a of slice type S:

- if x is out of range at run time, a run-time panic occurs
- a[x] is the slice element at index x and the type of a[x] is the element type of S

For a of string type:

- a constant index must be in range if the string a is also constant
- if x is out of range at run time, a run-time panic occurs
- a[x] is the non-constant byte value at index x and the type of a[x] is byte
- a[x] may not be assigned to

For a of map type M:

- 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 element with key x and the type of a[x] is the element type of M
- if the map is nil or does not contain such an entry, a[x] is the zero value for the element type of M

For a of type parameter type P:

- The index expression a[x] must be valid for values of all types in P's type set.
- The element types of all types in P's type set must be identical. In this context, the element type of a string type is byte.
- If there is a map type in the type set of P, all types in that type set must be map types, and the respective key types must be all

identical.
- `a[x]` is the array, slice, or string element at index `x`, or the map element with key `x` of the type argument that `P` is instantiated with, and the type of `a[x]` is the type of the (identical) element types.
- `a[x]` may not be assigned to if `P`'s type set includes string types.

Otherwise `a[x]` is illegal.

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

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

yields an additional untyped boolean value. The value of ok is `true` if the key `x` is present in the map, and `false` otherwise.

Assigning to an element of a `nil` map causes a run-time panic.

Slice expressions

Slice expressions construct a substring or slice from a string, array, pointer to array, or slice. There are two variants: a simple form that specifies a low and high bound, and a full form that also specifies a bound on the capacity.

Simple slice expressions

The primary expression

```
a[low : high]
```

constructs a substring or slice. The core type of a must be a string, array, pointer to array, slice, or a `bytestring`. The *indices* `low` and `high` select which elements of operand a appear in the result. The result has indices starting at 0 and length equal to `high` - `low`. 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, any of the indices may be omitted. A missing `low` index defaults to zero; a missing `high` index defaults to the length of the sliced operand:

```
a[2:]  // same as a[2 : len(a)]
a[:3]  // same as a[0 : 3]
a[:]   // same as a[0 : len(a)]
```

If a is a pointer to an array, `a[low : high]` is shorthand for `(*a)[low : high]`.

For arrays or strings, the indices are *in range* if `0 <= low <= high <= len(a)`, otherwise they are *out of range*. For slices, the upper index bound is the slice capacity `cap(a)` rather than the length. A constant index must be non-negative and representable by a value of type `int`; for arrays or constant strings, constant indices must also be in range. If both indices are constant, they must satisfy `low <= high`. If the indices are out of range at run time, a run-time panic occurs.

Except for untyped strings, if the sliced operand is a string or slice, the result of the slice operation is a non-constant value of the same type as the operand. For untyped string operands the result is a non-constant value of type `string`. If the sliced operand is an array, it must be addressable and the result of the slice operation is a slice with the same element type as the array.

If the sliced operand of a valid slice expression is a `nil` slice, the result is a `nil` slice. Otherwise, if the result is a slice, it shares its underlying array with the operand.

```
var a [10]int
s1 := a[3:7]   // underlying array of s1 is array a; &s1[2] == &a[5]
```

```
s2 := s1[1:4]  // underlying array of s2 is underlying array of s1 which is array a; &s2[1] == &a[5]
s2[1] = 42     // s2[1] == s1[2] == a[5] == 42; they all refer to the same underlying array element

var s []int
s3 := s[:0]    // s3 == nil
```

Full slice expressions

The primary expression

```
a[low : high : max]
```

constructs a slice of the same type, and with the same length and elements as the simple slice expression `a[low : high]`. Additionally, it controls the resulting slice's capacity by setting it to `max - low`. Only the first index may be omitted; it defaults to 0. The core type of a must be an array, pointer to array, or slice (but not a string). After slicing the array a

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

the slice `t` has type `[]int`, length 2, capacity 4, and elements

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

As for simple slice expressions, if a is a pointer to an array, `a[low : high : max]` is shorthand for `(*a)[low : high : max]`. If the sliced operand is an array, it must be addressable.

The indices are *in range* if `0 <= low <= high <= max <= cap(a)`, otherwise they are *out of range*. A constant index must be non-negative and representable by a value of type `int`; for arrays, constant indices must also be in range. If multiple indices are constant, the constants that are present must be in range relative to each other. If the indices are out of range at run time, a run-time panic occurs.

Type assertions

For an expression x of interface type, but not a type parameter, 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. In this case, T must implement the (interface) type of x; otherwise the type assertion is invalid since it is not possible for x to store a value of type T. If T is an interface type, x.(T) asserts that the dynamic type of x implements the interface T.

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.

```
var x interface{} = 7          // x has dynamic type int and value 7
i := x.(int)                   // i has type int and value 7

type I interface { m() }

func f(y I) {
	s := y.(string)        // illegal: string does not implement I (missing method m)
	r := y.(io.Reader)     // r has type io.Reader and the dynamic type of y must implement both I and io.Reader
	…
}
```

A type assertion used in an assignment statement or initialization of the special form

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
var v, ok interface{} = x.(T) // dynamic types of v and ok are T and bool
```

yields an additional untyped boolean value. The value of ok is true if the assertion holds. Otherwise it is false and the value of v is the zero value for type `T`. No run-time panic occurs in this case.

Calls

Given an expression `f` with a core type `F` of function type,

```
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
```

If `f` denotes a generic function, it must be instantiated before it can be called or used as a function value.

In a function call, the function value and arguments are evaluated in the usual order. After they are evaluated, the parameters of the call are passed by value to the function and the called function begins execution. The return parameters of the function are passed by value back to the caller when the function returns.

Calling a `nil` function value causes a run-time panic.

As a special case, if the return values 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, and g must have at least one return value. 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.Panic("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 a final parameter `p` of type `...T`, then within `f` the type of `p` is equivalent to type `[]T`. If `f` is invoked with no actual arguments for `p`, the value passed to `p` is `nil`. Otherwise, the value passed is a new slice of type `[]T` with a new underlying array whose successive elements are the actual arguments, which all must be assignable to `T`. The length and capacity of the slice is therefore the number of arguments bound to `p` and may differ for each call site.

Given the function and calls

```
func Greeting(prefix string, who ...string)
Greeting("nobody")
Greeting("hello:", "Joe", "Anna", "Eileen")
```

within `Greeting`, `who` will have the value `nil` in the first call, and `[]string{"Joe", "Anna", "Eileen"}` in the second.

If the final argument is assignable to a slice type `[]T` and is followed by `...`, it is passed unchanged as the value for a `...T` parameter. In this case no new slice is created.

Given the slice s and call

```
s := []string{"James", "Jasmine"}
Greeting("goodbye:", s...)
```

within Greeting, who will have the same value as s with the same underlying array.

Instantiations

A generic function or type is *instantiated* by substituting *type arguments* for the type parameters. Instantiation proceeds in two steps:

1. Each type argument is substituted for its corresponding type parameter in the generic declaration. This substitution happens across the entire function or type declaration, including the type parameter list itself and any types in that list.
2. After substitution, each type argument must satisfy the constraint (instantiated, if necessary) of the corresponding type parameter. Otherwise instantiation fails.

Instantiating a type results in a new non-generic named type; instantiating a function produces a new non-generic function.

```
type parameter list    type arguments    after substitution

[P any]                int               int satisfies any
[S ~[]E, E any]        []int, int        []int satisfies ~[]int, int satisfies any
[P io.Writer]          string            illegal: string doesn't satisfy io.Writer
[P comparable]         any               any satisfies (but does not implement) comparable
```

When using a generic function, type arguments may be provided explicitly, or they may be partially or completely inferred from the context in which the function is used. Provided that they can be inferred, type argument lists may be omitted entirely if the function is:

- called with ordinary arguments,
- assigned to a variable with a known type
- passed as an argument to another function, or

- returned as a result.

In all other cases, a (possibly partial) type argument list must be present. If a type argument list is absent or partial, all missing type arguments must be inferrable from the context in which the function is used.

```
// sum returns the sum (concatenation, for strings) of its arguments.
func sum[T ~int | ~float64 | ~string](x... T) T { … }

x := sum                       // illegal: the type of x is unknown
intSum := sum[int]             // intSum has type func(x... int) int
a := intSum(2, 3)              // a has value 5 of type int
b := sum[float64](2.0, 3)      // b has value 5.0 of type float64
c := sum(b, -1)                // c has value 4.0 of type float64

type sumFunc func(x... string) string
var f sumFunc = sum            // same as var f sumFunc = sum[string]
f = sum                        // same as f = sum[string]
```

A partial type argument list cannot be empty; at least the first argument must be present. The list is a prefix of the full list of type arguments, leaving the remaining arguments to be inferred. Loosely speaking, type arguments may be omitted from "right to left".

```
func apply[S ~[]E, E any](s S, f func(E) E) S { … }

f0 := apply[]                  // illegal: type argument list cannot be empty
f1 := apply[[]int]             // type argument for S explicitly provided, type argument for E inferred
f2 := apply[[]string, string]  // both type arguments explicitly provided

var bytes []byte
r := apply(bytes, func(byte) byte { … })  // both type arguments inferred from the function arguments
```

For a generic type, all type arguments must always be provided explicitly.

Type inference

A use of a generic function may omit some or all type arguments if they can be *inferred* from the context within which the function is used, including the constraints of the function's type parameters. Type inference succeeds if it can infer the missing type arguments and instantiation succeeds with the inferred type arguments. Otherwise, type inference fails and the program is invalid.

Type inference uses the type relationships between pairs of types for inference: For instance, a function argument must be assignable to its respective function parameter; this establishes a relationship between the type of the argument and the type of the parameter. If either of these two types contains type parameters, type inference looks for the type arguments to substitute the type parameters with such that the assignability relationship is satisfied. Similarly, type inference uses the fact that a type argument must satisfy the constraint of its respective type parameter.

Each such pair of matched types corresponds to a *type equation* containing one or multiple type parameters, from one or possibly multiple generic functions. Inferring the missing type arguments means solving the resulting set of type equations for the respective type parameters.

For example, given

```
// dedup returns a copy of the argument slice with any duplicate entries removed.
func dedup[S ~[]E, E comparable](S) S { … }

type Slice []int
var s Slice
s = dedup(s)   // same as s = dedup[Slice, int](s)
```

the variable `s` of type `Slice` must be assignable to the function parameter type `S` for the program to be valid. To reduce complexity, type inference ignores the directionality of assignments, so the type relationship between `Slice` and `S` can be expressed via the (symmetric) type equation `Slice` $\equiv_A$ `S` (or `S` $\equiv_A$ `Slice` for that matter), where the $_A$ in $\equiv_A$ indicates that the LHS and RHS types must match per assignability rules (see the section on type unification for details). Similarly, the type parameter `S` must satisfy its constraint `~[]E`. This can be expressed as `S` $\equiv_C$ `~[]E` where `X` $\equiv_C$ `Y` stands for "`X` satisfies constraint `Y`". These observations lead to a set of two equations

$$
\begin{array}{lll}
\texttt{Slice} & \equiv_A \texttt{S} & (1) \\
\texttt{S} & \equiv_C \texttt{\textasciitilde[]E} & (2)
\end{array}
$$

which now can be solved for the type parameters `S` and `E`. From (1) a compiler can infer that the type argument for `S` is `Slice`. Similarly, because the underlying type of `Slice` is `[]int` and `[]int` must match `[]E` of the constraint, a compiler can infer that `E` must be `int`. Thus, for these two equations, type inference infers

```
S ➞ Slice
E ➞ int
```

Given a set of type equations, the type parameters to solve for are the type parameters of the functions that need to be instantiated and for which no explicit type arguments is provided. These type parameters are called *bound* type parameters. For instance, in the `dedup` example above, the type parameters `P` and `E` are bound to `dedup`. An argument to a generic function call may be a generic function itself. The type parameters of that function are included in the set of bound type parameters. The types of function arguments may contain type parameters from other functions (such as a generic function enclosing a function call). Those type parameters may also appear in type equations but they are not bound in that context. Type equations are always solved for the bound type parameters only.

Type inference supports calls of generic functions and assignments of generic functions to (explicitly function-typed) variables. This includes passing generic functions as arguments to other (possibly also generic) functions, and returning generic functions as results. Type inference operates on a set of equations specific to each of these cases. The equations are as follows (type argument lists are omitted for clarity):

- For a function call $\texttt{f(a}_0\texttt{, a}_1\texttt{, …)}$ where `f` or a function argument $\texttt{a}_\texttt{i}$ is a generic function:
  Each pair $\texttt{(a}_\texttt{i}\texttt{, p}_\texttt{i}\texttt{)}$ of corresponding function arguments and parameters where $\texttt{a}_\texttt{i}$ is not an untyped constant yields an equation $\texttt{typeof(p}_\texttt{i}\texttt{)} \equiv_A \texttt{typeof(a}_\texttt{i}\texttt{)}$.
  If $\texttt{a}_\texttt{i}$ is an untyped constant $\texttt{c}_\texttt{j}$, and $\texttt{typeof(p}_\texttt{i}\texttt{)}$ is a bound type parameter $\texttt{P}_k$, the pair $\texttt{(c}_\texttt{j}\texttt{, P}_k\texttt{)}$ is collected separately from the type equations.

- For an assignment `v = f` of a generic function `f` to a (non-generic) variable `v` of function type:
  $\texttt{typeof(v)} \equiv_A \texttt{typeof(f)}$.

- For a return statement `return …, f, …` where `f` is a generic function returned as a result to a (non-generic) result variable `r` of function type:
  $\texttt{typeof(r)} \equiv_A \texttt{typeof(f)}$.

Additionally, each type parameter P_k and corresponding type constraint C_k yields the type equation $P_k \equiv_C C_k$.

Type inference gives precedence to type information obtained from typed operands before considering untyped constants. Therefore, inference proceeds in two phases:

1. The type equations are solved for the bound type parameters using type unification. If unification fails, type inference fails.
2. For each bound type parameter P_k for which no type argument has been inferred yet and for which one or more pairs (c_j, P_k) with that same type parameter were collected, determine the constant kind of the constants c_j in all those pairs the same way as for constant expressions. The type argument for P_k is the default type for the determined constant kind. If a constant kind cannot be determined due to conflicting constant kinds, type inference fails.

If not all type arguments have been found after these two phases, type inference fails.

If the two phases are successful, type inference determined a type argument for each bound type parameter:

$$P_k \rightarrow A_k$$

A type argument A_k may be a composite type, containing other bound type parameters P_k as element types (or even be just another bound type parameter). In a process of repeated simplification, the bound type parameters in each type argument are substituted with the respective type arguments for those type parameters until each type argument is free of bound type parameters.

If type arguments contain cyclic references to themselves through bound type parameters, simplification and thus type inference fails. Otherwise, type inference succeeds.

Type unification

Type inference solves type equations through *type unification*. Type unification recursively compares the LHS and RHS types of an equation, where either or both types may be or contain bound type parameters, and looks for type arguments for those type parameters such that the LHS and RHS match (become identical or assignment-compatible, depending on context). To that effect, type inference maintains a map of bound type parameters to inferred type arguments; this map is consulted and updated during type unification. Initially, the bound type parameters are known but the map is empty. During type unification, if a new type argument A is inferred, the respective mapping `P ➞ A` from type parameter to argument is added to the map. Conversely, when comparing types, a known type argument (a type argument for which a map entry already exists) takes the place of its corresponding type parameter. As type inference progresses, the map is populated more and more until all equations have been considered, or until unification fails. Type inference succeeds if no unification step fails and the map has an entry for each type parameter.

For example, given the type equation with the bound type parameter `P`

```
[10]struct{ elem P, list []P } ≡A [10]struct{ elem string; list []string }
```

type inference starts with an empty map. Unification first compares the top-level structure of the LHS and RHS types. Both are arrays of the same length; they unify if the element types unify. Both element types are structs; they unify if they have the same number of fields with the same names and if the field types unify. The type argument for `P` is not known yet (there is no map entry), so unifying `P` with `string` adds the mapping `P ➞ string` to the map. Unifying the types of the `list` field requires unifying `[]P` and `[]string` and thus `P` and `string`. Since the type argument for `P` is known at this point (there is a map entry for `P`), its type argument `string` takes the place of `P`. And since `string` is identical to `string`, this unification step succeeds as well. Unification of the LHS and RHS of the equation is now finished. Type inference succeeds because there is only one type equation, no unification step failed, and the map is fully populated.

Unification uses a combination of *exact* and *loose* unification depending on whether two types have to be identical, assignment-compatible, or only structurally equal. The respective type unification rules are spelled out in detail in the Appendix.

For an equation of the form $X \equiv_A Y$, where `X` and `Y` are types involved in an assignment (including parameter passing and return statements), the top-level type structures may unify loosely but element types must unify exactly, matching the rules for assignments.

For an equation of the form $P \equiv_C C$, where `P` is a type parameter and `C` its corresponding constraint, the unification rules are bit more complicated:

- If `C` has a core type `core(C)` and `P` has a known type argument `A`, `core(C)` and `A` must unify loosely. If `P` does not have a known type argument and `C` contains exactly one type term `T` that is not an underlying (tilde) type, unification adds the mapping `P ➞ T` to the map.
- If `C` does not have a core type and `P` has a known type argument `A`, `A` must have all methods of `C`, if any, and corresponding method types must unify exactly.

When solving type equations from type constraints, solving one equation may infer additional type arguments, which in turn may enable solving other equations that depend on those type arguments. Type inference repeats type unification as long as new type arguments are inferred.

Operators

Operators combine operands into expressions.

```
Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .

binary_op  = "||" | "&&" | rel_op | add_op | mul_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 shifts or untyped constants. For operations involving constants only, see the section on constant expressions.

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

The right operand in a shift expression must have integer type or be an untyped constant representable by a value of type `uint`. If the left operand of a non-constant shift expression is an untyped constant, it is first implicitly converted to the type it would assume if the shift expression were replaced by its left operand alone.

```
var a [1024]byte
var s uint = 33

// The results of the following examples are given for 64-bit ints.
var i = 1<<s                   // 1 has type int
var j int32 = 1<<s             // 1 has type int32; j == 0
```

```
var k = uint64(1<<s)          // 1 has type uint64; k == 1<<33
var m int = 1.0<<s            // 1.0 has type int; m == 1<<33
var n = 1.0<<s == j           // 1.0 has type int32; n == true
var o = 1<<s == 2<<s          // 1 and 2 have type int; o == false
var p = 1<<s == 1<<33         // 1 has type int; p == true
var u = 1.0<<s                // illegal: 1.0 has type float64, cannot shift
var u1 = 1.0<<s != 0          // illegal: 1.0 has type float64, cannot shift
var u2 = 1<<s != 1.0          // illegal: 1 has type float64, cannot shift
var v1 float32 = 1<<s         // illegal: 1 has type float32, cannot shift
var v2 = string(1<<s)         // illegal: 1 is converted to a string, cannot shift
var w int64 = 1.0<<33         // 1.0<<33 is a constant shift expression; w == 1<<33
var x = a[1.0<<s]             // panics: 1.0 has type int, but 1<<33 overflows array bounds
var b = make([]byte, 1.0<<s)  // 1.0 has type int; len(b) == 1<<33

// The results of the following examples are given for 32-bit ints,
// which means the shifts will overflow.
var mm int = 1.0<<s           // 1.0 has type int; mm == 0
var oo = 1<<s == 2<<s         // 1 and 2 have type int; oo == true
var pp = 1<<s == 1<<33        // illegal: 1 has type int, but 1<<33 overflows int
var xx = a[1.0<<s]            // 1.0 has type int; xx == a[0]
var bb = make([]byte, 1.0<<s) // 1.0 has type int; len(bb) == 0
```

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 five precedence levels for binary operators. Multiplication operators bind strongest, followed by addition operators, comparison operators, `&&` (logical AND), and finally `||` (logical OR):

```
Precedence    Operator
    5             *  /  %  <<  >>  &  &^
    4             +  -  |  ^
    3             ==  !=  <  <=  >  >=
    2             &&
    1             ||
```

Binary operators of the same precedence associate from left to right. For instance, `x / y * z` is the same as `(x / y) * z`.

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y+1 && <-chanInt > 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. The bitwise logical and shift operators apply to integers only.

```
+    sum                    integers, floats, complex values, strings
-    difference             integers, floats, complex values
*    product                integers, floats, complex values
/    quotient               integers, floats, complex values
%    remainder              integers

&    bitwise AND            integers
|    bitwise OR             integers
^    bitwise XOR            integers
&^   bit clear (AND NOT)    integers

<<   left shift             integer << integer >= 0
```

```
>>   right shift            integer >> integer >= 0
```

If the operand type is a type parameter, the operator must apply to each type in that type set. The operands are represented as values of the type argument that the type parameter is instantiated with, and the operation is computed with the precision of that type argument. For example, given the function:

```
func dotProduct[F ~float32|~float64](v1, v2 []F) F {
	var s F
	for i, x := range v1 {
		y := v2[i]
		s += x * y
	}
	return s
}
```

the product `x * y` and the addition `s += x * y` are computed with `float32` or `float64` precision, respectively, depending on the type argument for `F`.

Integer operators

For two integer values `x` and `y`, the integer quotient `q = x / y` and remainder `r = x % y` satisfy the following relationships:

```
x = q*y + r  and  |r| < |y|
```

with `x / y` truncated towards zero ("truncated division").

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

The one exception to this rule is that if the dividend `x` is the most negative value for the int type of `x`, the quotient `q = x / -1` is equal to `x` (and `r = 0`) due to two's-complement integer overflow:

```
                         x, q
int8                     -128
int16                  -32768
int32             -2147483648
int64    -9223372036854775808
```

If the divisor is a constant, it must not be zero. If the divisor is zero at run time, a run-time panic occurs. If the dividend is non-negative 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, which must be non-negative. If the shift count is negative at run time, a run-time panic occurs. The shift operators implement arithmetic shifts if the left operand is a signed integer and logical shifts if it is 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 signed x
```

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. 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. Overflow does not cause a run-time panic. 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.

Floating-point operators

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

An implementation may combine multiple floating-point operations into a single fused operation, possibly across statements, and produce a result that differs from the value obtained by executing and rounding the instructions individually. An explicit floating-point type conversion rounds to the precision of the target type, preventing fusion that would discard that rounding.

For instance, some architectures provide a "fused multiply and add" (FMA) instruction that computes x*y + z without rounding the intermediate result x*y. These examples show when a Go implementation can use that instruction:

```
// FMA allowed for computing r, because x*y is not explicitly rounded:
r  = x*y + z
r  = z;   r += x*y
t  = x*y; r = t + z
*p = x*y; r = *p + z
r  = x*y + float64(z)

// FMA disallowed for computing r, because it would omit rounding of x*y:
r  = float64(x*y) + z
r  = z; r += float64(x*y)
t  = float64(x*y); r = t + z
```

String concatenation

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.

Comparison operators

Comparison operators compare two operands and yield an untyped boolean value.

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

In any comparison, the first operand must be assignable to the type of the second operand, or vice versa.

The equality operators `==` and `!=` apply to operands of *comparable* types. The ordering operators `<`, `<=`, `>`, and `>=` apply to operands of *ordered* types. These terms and the result of the comparisons are defined as follows:

- Boolean types are comparable. Two boolean values are equal if they are either both `true` or both `false`.
- Integer types are comparable and ordered. Two integer values are compared in the usual way.
- Floating-point types are comparable and ordered. Two floating-point values are compared as defined by the IEEE-754 standard.
- Complex types are comparable. Two complex values `u` and `v` are equal if both `real(u) == real(v)` and `imag(u) == imag(v)`.
- String types are comparable and ordered. Two string values are compared lexically byte-wise.
- Pointer types are comparable. Two pointer values are equal if they point to the same variable or if both have value `nil`. Pointers to distinct zero-size variables may or may not be equal.
- Channel types are comparable. Two channel values are equal if they were created by the same call to `make` or if both have value `nil`.
- Interface types that are not type parameters are comparable. Two interface values are equal if they have identical dynamic types and equal dynamic values or if both have value `nil`.
- A value `x` of non-interface type `X` and a value `t` of interface type `T` can be compared if type `X` is comparable and `X` implements `T`.

They are equal if `t`'s dynamic type is identical to `X` and `t`'s dynamic value is equal to `x`.

- Struct types are comparable if all their field types are comparable. Two struct values are equal if their corresponding non-blank field values are equal. The fields are compared in source order, and comparison stops as soon as two field values differ (or all fields have been compared).
- Array types are comparable if their array element types are comparable. Two array values are equal if their corresponding element values are equal. The elements are compared in ascending index order, and comparison stops as soon as two element values differ (or all elements have been compared).
- Type parameters are comparable if they are strictly comparable (see below).

A comparison of two interface values with identical dynamic types causes a run-time panic if that type is not comparable. This behavior applies not only to direct interface value comparisons but also when comparing arrays of interface values or structs with interface-valued fields.

Slice, map, and function types are not comparable. However, as a special case, a slice, map, or function value may be compared to the predeclared identifier `nil`. Comparison of pointer, channel, and interface values to `nil` is also allowed and follows from the general rules above.

```
const c = 3 < 4            // c is the untyped boolean constant true

type MyBool bool
var x, y int
var (
	// The result of a comparison is an untyped boolean.
	// The usual assignment rules apply.
	b3        = x == y // b3 has type bool
	b4 bool   = x == y // b4 has type bool
	b5 MyBool = x == y // b5 has type MyBool
)
```

A type is *strictly comparable* if it is comparable and not an interface type nor composed of interface types. Specifically:

- Boolean, numeric, string, pointer, and channel types are strictly comparable.
- Struct types are strictly comparable if all their field types are strictly comparable.

- Array types are strictly comparable if their array element types are strictly comparable.
- Type parameters are strictly comparable if all types in their type set are strictly comparable.

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 false"
||    conditional OR     p || q  is  "if p then true else q"
!     NOT                !p      is  "not p"
```

Address operators

For an operand x of type T, the address operation &x generates a pointer of type *T to x. The operand 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. As an exception to the addressability requirement, x may also be a (possibly parenthesized) composite literal. If the evaluation of x would cause a run-time panic, then the evaluation of &x does too.

For an operand x of pointer type *T, the pointer indirection *x denotes the variable of type T pointed to by x. If x is nil, an attempt to evaluate *x will cause a run-time panic.

```
&x
&a[f(2)]
&Point{2, 3}
*p
*pf(x)

var x *int = nil
*x   // causes a run-time panic
&*x  // causes a run-time panic
```

Receive operator

For an operand `ch` whose core type is a channel, the value of the receive operation `<-ch` is the value received from the channel `ch`. The channel direction must permit receive operations, and the type of the receive operation is the element type of the channel. The expression blocks until a value is available. Receiving from a `nil` channel blocks forever. A receive operation on a closed channel can always proceed immediately, yielding the element type's zero value after any previously sent values have been received.

```
v1 := <-ch
v2 = <-ch
f(<-ch)
<-strobe  // wait until clock pulse and discard received value
```

A receive expression used in an assignment statement or initialization of the special form

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

yields an additional untyped boolean result reporting whether the communication succeeded. The value of `ok` is `true` if the value received was delivered by a successful send operation to the channel, or `false` if it is a zero value generated because the channel is closed and empty.

Conversions

A conversion changes the type of an expression to the type specified by the conversion. A conversion may appear literally in the source, or it may be *implied* by the context in which an expression appears.

An *explicit* conversion is an expression 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 the operator `*` or `<-`, or if the type starts with the keyword `func` and has no result list, it must be parenthesized when necessary to avoid ambiguity:

```
*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
func()(x)        // function signature func() x
(func())(x)      // x is converted to func()
(func() int)(x)  // x is converted to func() int
func() int(x)    // x is converted to func() int (unambiguous)
```

A constant value x can be converted to type T if x is representable by a value of T. As a special case, an integer constant x can be explicitly converted to a string type using the same rule as for non-constant x.

Converting a constant to a type that is not a type parameter yields a typed constant.

```
uint(iota)               // iota value of type uint
float32(2.718281828)     // 2.718281828 of type float32
complex128(1)            // 1.0 + 0.0i of type complex128
float32(0.49999999)      // 0.5 of type float32
float64(-1e-1000)        // 0.0 of type float64
string('x')              // "x" of type string
string(0x266c)           // "♬" of type string
myString("foo" + "bar")  // "foobar" of type myString
string([]byte{'a'})      // not a constant: []byte{'a'} is not a constant
(*int)(nil)              // not a constant: nil is not a constant, *int is not a boolean, numeric, or string type
int(1.2)                 // illegal: 1.2 cannot be represented as an int
string(65.0)             // illegal: 65.0 is not an integer constant
```

Converting a constant to a type parameter yields a *non-constant* value of that type, with the value represented as a value of the type argument that the type parameter is instantiated with. For example, given the function:

```
func f[P ~float32|~float64]() {
        … P(1.1) …
}
```

the conversion `P(1.1)` results in a non-constant value of type `P` and the value `1.1` is represented as a `float32` or a `float64` depending on the type argument for `f`. Accordingly, if `f` is instantiated with a `float32` type, the numeric value of the expression `P(1.1) + 1.2` will be computed with the same precision as the corresponding non-constant `float32` addition.

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

- `x` is assignable to `T`.
- ignoring struct tags (see below), `x`'s type and `T` are not type parameters but have identical underlying types.
- ignoring struct tags (see below), `x`'s type and `T` are pointer types that are not named types, and their pointer base types are not type parameters but 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 a slice of bytes or runes and `T` is a string type.
- `x` is a string and `T` is a slice of bytes or runes.
- `x` is a slice, `T` is an array or a pointer to an array, and the slice and array types have identical element types.

Additionally, if `T` or `x`'s type `V` are type parameters, `x` can also be converted to type `T` if one of the following conditions applies:

- Both `V` and `T` are type parameters and a value of each type in `V`'s type set can be converted to each type in `T`'s type set.
- Only `V` is a type parameter and a value of each type in `V`'s type set can be converted to `T`.
- Only `T` is a type parameter and `x` can be converted to each type in `T`'s type set.

Struct tags are ignored when comparing struct types for identity for the purpose of conversion:

```
type Person struct {
	Name    string
	Address *struct {
		Street string
		City   string
	}
}

var data *struct {
```

```
		Name    string `json:"name"`
		Address *struct {
			Street string `json:"street"`
			City   string `json:"city"`
		} `json:"address"`
	}

	var person = (*Person)(data)  // ignoring tags, the underlying types are identical
```

Specific rules apply to (non-constant) 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`.

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

Conversions between numeric types

For the conversion of non-constant numeric values, the following rules apply:

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 non-constant 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 slice of bytes to a string type yields a string whose successive bytes are the elements of the slice.

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

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

type myByte byte
string([]myByte{'w', 'o', 'r', 'l', 'd', '!'})       // "world!"
myString([]myByte{'\xf0', '\x9f', '\x8c', '\x8d'})   // "🌍"
```

2. Converting a slice of runes to a string type yields a string that is the concatenation of the individual rune values converted to strings.

```
string([]rune{0x767d, 0x9d6c, 0x7fd4})   // "\u767d\u9d6c\u7fd4" == "白鵬翔"
string([]rune{})                         // ""
string([]rune(nil))                      // ""

type runes []rune
string(runes{0x767d, 0x9d6c, 0x7fd4})    // "\u767d\u9d6c\u7fd4" == "白鵬翔"

type myRune rune
string([]myRune{0x266b, 0x266c})         // "\u266b\u266c" == "♫♬"
myString([]myRune{0x1f30e})              // "\U0001f30e" == "🌎"
```

3. Converting a value of a string type to a slice of bytes type yields a slice whose successive elements are the bytes of the string.

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

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

[]myByte("world!")          // []myByte{'w', 'o', 'r', 'l', 'd', '!'}
[]myByte(myString("🌏"))    // []myByte{'\xf0', '\x9f', '\x8c', '\x8f'}
```

4. Converting a value of a string type to a slice of runes type yields a slice containing the individual Unicode code points of the string.

```
[]rune(myString("白鵬翔"))   // []rune{0x767d, 0x9d6c, 0x7fd4}
[]rune("")                  // []rune{}

runes("白鵬翔")              // []rune{0x767d, 0x9d6c, 0x7fd4}

[]myRune("♫♬")              // []myRune{0x266b, 0x266c}
[]myRune(myString("🌐"))    // []myRune{0x1f310}
```

5. Finally, for historical reasons, an integer value may be converted to a string type. This form of conversion yields a string containing the (possibly multi-byte) UTF-8 representation of the Unicode code point with the given integer value. Values outside the range of valid Unicode code points are converted to `"\uFFFD"`.

```
string('a')          // "a"
string(65)           // "A"
string('\xf8')       // "\u00f8" == "ø" == "\xc3\xb8"
string(-1)           // "\ufffd" == "\xef\xbf\xbd"

type myString string
myString('\u65e5')   // "\u65e5" == "日" == "\xe6\x97\xa5"
```

Note: This form of conversion may eventually be removed from the language. The go `vet` tool flags certain integer-to-string conversions as potential errors. Library functions such as `utf8.AppendRune` or `utf8.EncodeRune` should be used instead.

Conversions from slice to array or array pointer

Converting a slice to an array yields an array containing the elements of the underlying array of the slice. Similarly, converting a slice to an array pointer yields a pointer to the underlying array of the slice. In both cases, if the length of the slice is less than the length of the array, a run-time panic occurs.

```
s := make([]byte, 2, 4)

a0 := [0]byte(s)
a1 := [1]byte(s[1:])     // a1[0] == s[1]
a2 := [2]byte(s)         // a2[0] == s[0]
```

```
a4 := [4]byte(s)         // panics: len([4]byte) > len(s)

s0 := (*[0]byte)(s)      // s0 != nil
s1 := (*[1]byte)(s[1:])  // &s1[0] == &s[1]
s2 := (*[2]byte)(s)      // &s2[0] == &s[0]
s4 := (*[4]byte)(s)      // panics: len([4]byte) > len(s)

var t []string
t0 := [0]string(t)       // ok for nil slice t
t1 := (*[0]string)(t)    // t1 == nil
t2 := (*[1]string)(t)    // panics: len([1]string) > len(t)

u := make([]byte, 0)
u0 := (*[0]byte)(u)      // u0 != nil
```

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.

A constant comparison always yields an untyped boolean constant. If the left operand of a constant shift expression is an untyped constant, the result is an integer constant; otherwise it is a constant of the same type as the left operand, which must be of integer type.

Any other operation on untyped constants results in an untyped constant of the same kind; that is, a boolean, integer, floating-point, complex, or string constant. If the untyped operands of a binary operation (other than a shift) are of different kinds, the result is of the operand's kind that appears later in this list: integer, rune, floating-point, complex. For example, an untyped integer constant divided by an untyped complex constant yields an untyped complex constant.

```
const a = 2 + 3.0          // a == 5.0   (untyped floating-point constant)
const b = 15 / 4           // b == 3     (untyped integer constant)
const c = 15 / 4.0         // c == 3.75  (untyped floating-point constant)
const Θ float64 = 3/2      // Θ == 1.0   (type float64, 3/2 is integer division)
const Π float64 = 3/2.     // Π == 1.5   (type float64, 3/2. is float division)
```

```
const d = 1 << 3.0         // d == 8     (untyped integer constant)
const e = 1.0 << 3         // e == 8     (untyped integer constant)
const f = int32(1) << 33   // illegal    (constant 8589934592 overflows int32)
const g = float64(2) >> 1  // illegal    (float64(2) is a typed floating-point constant)
const h = "foo" > "bar"    // h == true  (untyped boolean constant)
const j = true             // j == true  (untyped boolean constant)
const k = 'w' + 1          // k == 'x'   (untyped rune constant)
const l = "hi"             // l == "hi"  (untyped string constant)
const m = string(k)        // m == "x"   (type string)
const Σ = 1 - 0.707i       //            (untyped complex constant)
const Δ = Σ + 2.0e-4       //            (untyped complex constant)
const Φ = iota*1i - 1/1i   //            (untyped complex constant)
```

Applying the built-in function `complex` to untyped integer, rune, or floating-point constants yields an untyped complex constant.

```
const ic = complex(0, c)   // ic == 3.75i  (untyped complex constant)
const iΘ = complex(0, Θ)   // iΘ == 1i     (type complex128)
```

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         // Huge == 1267650600228229401496703205376  (untyped integer constant)
const Four int8 = Huge >> 98  // Four == 4                                (type int8)
```

The divisor of a constant division or remainder operation must not be zero:

```
3.14 / 0.0   // illegal: division by zero
```

The values of *typed* constants must always be accurately representable by 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)  // 1267650600228229401496703205376 cannot be represented as an int64
Four * 300   // operand 300 cannot be represented as an int8 (type of Four)
Four * 100   // product 400 cannot be represented as an int8 (type of Four)
```

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)  // illegal: same as uint8(-2), -2 cannot be represented as a uint8
^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
```

Implementation restriction: A compiler may use rounding while computing untyped floating-point or complex constant expressions; see the implementation restriction in the section on constants. This rounding may cause a floating-point constant expression to be invalid in an integer context, even if it would be integral when calculated using infinite precision, and vice versa.

Order of evaluation

At package level, initialization dependencies determine the evaluation order of individual initialization expressions in variable declarations. Otherwise, when evaluating the operands of an expression, assignment, or return statement, all function calls, method calls, and communication operations are evaluated in lexical left-to-right order.

For example, in the (function-local) 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.

```
a := 1
f := func() int { a++; return a }
x := []int{a, f()}            // x may be [1, 2] or [2, 2]: evaluation order between a and f() is not specified
```

```
m := map[int]int{a: 1, a: 2}  // m may be {2: 1} or {2: 2}: evaluation order between the two map assignments is not specified
n := map[int]int{a: f()}      // n may be {2: 3} or {3: 3}: evaluation order between the key and the value is not specified
```

At package level, initialization dependencies override the left-to-right rule for individual initialization expressions, but not for operands within each expression:

```
var a, b, c = f() + v(), g(), sqr(u()) + v()

func f() int        { return c }
func g() int        { return a }
func sqr(x int) int { return x*x }

// functions u and v are independent of all other variables and functions
```

The function calls happen in the order `u()`, `sqr()`, `v()`, `f()`, `v()`, and `g()`.

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 | SelectStmt | ForStmt |
	DeferStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt | Assignment | ShortVarDecl .
```

Terminating statements

A *terminating statement* interrupts the regular flow of control in a block. The following statements are terminating:

1. A "return" or "goto" statement.
2. A call to the built-in function `panic`.
3. A block in which the statement list ends in a terminating statement.
4. An "if" statement in which:
   - the "else" branch is present, and
   - both branches are terminating statements.
5. A "for" statement in which:
   - there are no "break" statements referring to the "for" statement, and
   - the loop condition is absent, and
   - the "for" statement does not use a range clause.
6. A "switch" statement in which:
   - there are no "break" statements referring to the "switch" statement,
   - there is a default case, and
   - the statement lists in each case, including the default, end in a terminating statement, or a possibly labeled "fallthrough" statement.
7. A "select" statement in which:
   - there are no "break" statements referring to the "select" statement, and
   - the statement lists in each case, including the default if present, end in a terminating statement.
8. A labeled statement labeling a terminating statement.

All other statements are not terminating.

A statement list ends in a terminating statement if the list is not empty and its final non-empty statement is terminating.

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.Panic("error encountered")
```

Expression statements

With the exception of specific built-in functions, function and method calls and receive operations can appear in statement context. Such statements may be parenthesized.

```
ExpressionStmt = Expression .
```

The following built-in functions are not permitted in statement context:

```
append cap complex imag len make new real
unsafe.Add unsafe.Alignof unsafe.Offsetof unsafe.Sizeof unsafe.Slice unsafe.SliceData unsafe.String unsafe.StringData
```

```
h(x+y)
f.Close()
<-ch
(<-ch)
len("foo")  // illegal if len is the built-in function
```

Send statements

A send statement sends a value on a channel. The channel expression's core type must be a channel, the channel direction must permit send operations, and the type of the value to be sent must be assignable to the channel's element type.

```
SendStmt = Channel "<-" Expression .
Channel  = Expression .
```

Both the channel and the value expression are evaluated before communication begins. Communication blocks until the send can proceed. 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. A send on a closed channel proceeds by causing a run-time panic. A send on a `nil` channel blocks forever.

```
ch <- 3  // send value 3 to channel ch
```

IncDec statements

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

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

The following assignment statements are semantically equivalent:

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

Assignment statements

An *assignment* replaces the current value stored in a variable with a new value specified by an expression. An assignment statement may assign a single value to a single variable, or multiple values to a matching number of variables.

```
Assignment = ExpressionList assign_op ExpressionList .

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

Each left-hand side operand must be addressable, a map index expression, or (for = assignments only) the blank identifier. Operands may be parenthesized.

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

An *assignment operation* x *op*= y where *op* is a binary arithmetic operator 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, and the left-hand expression must not be the blank identifier.

```
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 call, a 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. 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:

```
one, two, three = '一', '二', '三'
```

The blank identifier provides a way to ignore right-hand side values in an assignment:

```
_ = x       // evaluate x but ignore it
x, _ = f()  // evaluate f() but ignore second result value
```

The assignment proceeds in two phases. First, the operands of index expressions and pointer indirections (including implicit pointer indirections in selectors) on the left and the expressions on the right are all evaluated in the usual order. Second, the assignments are carried out in left-to-right order.

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

x := []int{1, 2, 3}
i := 0
i, x[i] = 1, 2  // set i = 1, x[0] = 2

i = 0
x[i], i = 2, 1  // set x[0] = 2, i = 1

x[0], x[0] = 1, 2  // set x[0] = 1, then x[0] = 2 (so x[0] == 2 at end)

x[1], x[3] = 4, 5  // set x[1] = 4, then panic setting x[3] = 5.

type Point struct { x, y int }
var p *Point
x[2], p.x = 6, 7  // set x[2] = 6, then panic setting p.x = 7

i = 2
x = []int{3, 5, 7}
for i, x[i] = range x {  // set i, x[2] = 0, x[0]
        break
}
// after this loop, i == 0 and x is []int{3, 5, 3}
```

In assignments, each value must be assignable to the type of the operand to which it is assigned, with the following special cases:

1. Any typed value may be assigned to the blank identifier.
2. If an untyped constant is assigned to a variable of interface type or the blank identifier, the constant is first implicitly converted to its default type.
3. If an untyped boolean value is assigned to a variable of interface type or the blank identifier, it is first implicitly converted to type `bool`.

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.

```
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ] .
```

```
if x > max {
	x = max
}
```

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 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. The switch expression is evaluated exactly once in a switch statement.

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 boolean value `true`.

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

If the switch expression evaluates to an untyped constant, it is first implicitly converted to its default type. The predeclared untyped value `nil` cannot be used as a switch expression. The switch expression type must be comparable.

If a case expression is untyped, it is first implicitly converted to the type of the switch expression. For each (possibly converted) case expression `x` and the value `t` of the switch expression, `x == t` must be a valid comparison.

In other words, the switch expression is treated as if it were used to declare and initialize a temporary variable `t` without explicit type; it is that value of `t` against which each case expression `x` is tested for equality.

In a case or default clause, the last non-empty statement may be a (possibly labeled) "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. A "fallthrough" statement may appear as the last statement of all but the last clause of an expression switch.

The switch 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 "true"
case x < 0: return -x
default: return x
}

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

Implementation restriction: A compiler may disallow multiple case expressions evaluating to the same constant. For instance, the current compilers disallow duplicate integer, floating point, or string constants in case expressions.

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 keyword `type` rather than an actual type:

```
switch x.(type) {
// cases
}
```

Cases then match actual types `T` against the dynamic type of the expression `x`. As with type assertions, `x` must be of interface type, but not a type parameter, and each non-interface type `T` listed in a case must implement the type of `x`. The types listed in the cases of a type switch must all be different.

```
TypeSwitchStmt  = "switch" [ SimpleStmt ";" ] TypeSwitchGuard "{" { TypeCaseClause } "}" .
```

```
TypeSwitchGuard = [ identifier ":=" ] PrimaryExpr "." "(" "type" ")" .
TypeCaseClause  = TypeSwitchCase ":" StatementList .
TypeSwitchCase  = "case" TypeList | "default" .
```

The TypeSwitchGuard may include a short variable declaration. When that form is used, the variable is declared at the end of the TypeSwitchCase in the implicit block of 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.

Instead of a type, a case may use the predeclared identifier `nil`; that case is selected when the expression in the TypeSwitchGuard is a `nil` interface value. There may be at most one `nil` case.

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

```
switch i := x.(type) {
case nil:
	printString("x is nil")                // type of i is type of x (interface{})
case int:
	printInt(i)                            // type of i is int
case float64:
	printFloat64(i)                        // type of i is float64
case func(int) float64:
	printFunction(i)                       // type of i is func(int) float64
case bool, string:
	printString("type is bool or string")  // type of i is type of x (interface{})
default:
	printString("don't know the type")     // type of i is type of x (interface{})
}
```

could be rewritten:

```
v := x  // x is evaluated exactly once
if v == nil {
	i := v                                 // type of i is type of x (interface{})
	printString("x is nil")
} else if i, isInt := v.(int); isInt {
```

```
        printInt(i)                            // type of i is int
} else if i, isFloat64 := v.(float64); isFloat64 {
        printFloat64(i)                        // type of i is float64
} else if i, isFunc := v.(func(int) float64); isFunc {
        printFunction(i)                       // type of i is func(int) float64
} else {
        _, isBool := v.(bool)
        _, isString := v.(string)
        if isBool || isString {
                i := v                         // type of i is type of x (interface{})
                printString("type is bool or string")
        } else {
                i := v                         // type of i is type of x (interface{})
                printString("don't know the type")
        }
}
```

A type parameter or a generic type may be used as a type in a case. If upon instantiation that type turns out to duplicate another entry in the switch, the first matching case is chosen.

```
func f[P any](x any) int {
        switch x.(type) {
        case P:
                return 0
        case string:
                return 1
        case []P:
                return 2
        case []byte:
                return 3
        default:
                return 4
        }
}

var v1 = f[string]("foo")   // v1 == 0
```

```
var v2 = f[byte]([]byte{})  // v2 == 2
```

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. There are three forms: The iteration may be controlled by a single condition, a "for" clause, or a "range" clause.

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

For statements with single condition

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 the boolean value `true`.

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

For statements with `for` clause

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. Variables declared by the init statement are re-used in each iteration.

```
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 the boolean value `true`.

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

For statements with `range` clause

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 assigns *iteration values* to corresponding *iteration variables* if present and then executes the block.

```
RangeClause = [ ExpressionList "=" | IdentifierList ":=" ] "range" Expression .
```

The expression on the right in the "range" clause is called the *range expression*, its core type must be an array, pointer to an array, slice, string, map, or channel permitting receive operations. As with an assignment, if present the operands on the left must be addressable or map index expressions; they denote the iteration variables. If the range expression is a channel, at most one iteration variable is permitted, otherwise there may be up to two. If the last iteration variable is the blank identifier, the range clause is equivalent to the same clause without that identifier.

The range expression `x` is evaluated once before beginning the loop, with one exception: if at most one iteration variable is present and `len(x)` is constant, the range expression is not evaluated.

Function calls on the left are evaluated once per iteration. For each iteration, iteration values are produced as follows if the respective iteration variables are present:

```
Range expression                          1st value          2nd value
```

```
array or slice  a  [n]E, *[n]E, or []E    index    i  int    a[i]       E
string          s  string type            index    i  int    see below  rune
map             m  map[K]V                key      k  K      m[k]       V
channel         c  chan E, <-chan E       element  e  E
```

1. For an array, pointer to array, or slice value `a`, the index iteration values are produced in increasing order, starting at element index 0. If at most one iteration variable is present, the range loop produces iteration values from 0 up to `len(a)-1` and does not index into the array or slice itself. For a `nil` slice, the number of iterations is 0.
2. For a string value, the "range" clause iterates over the Unicode code points in the string starting at byte index 0. On successive iterations, the index value will be the index of the first byte of successive UTF-8-encoded code points in the string, and the second value, of type `rune`, will be the value of the corresponding code point. If the iteration encounters an invalid UTF-8 sequence, the second value will be `0xFFFD`, the Unicode replacement character, and the next iteration will advance a single byte in the string.
3. The iteration order over maps is not specified and is not guaranteed to be the same from one iteration to the next. If a map entry that has not yet been reached is removed during iteration, the corresponding iteration value will not be produced. If a map entry is created during iteration, that entry may be produced during the iteration or may be skipped. The choice may vary for each entry created and from one iteration to the next. If the map is `nil`, the number of iterations is 0.
4. For channels, the iteration values produced are the successive values sent on the channel until the channel is closed. If the channel is `nil`, the range expression blocks forever.

The iteration values are assigned to the respective iteration variables as in an assignment statement.

The iteration variables may be declared by the "range" clause using a form of short variable declaration (`:=`). In this case their types are set to the types of the respective iteration values and their scope is the block of the "for" statement; they are re-used in each iteration. If the iteration variables are declared outside the "for" statement, after execution their values will be those of the last iteration.

```
var testdata *struct {
	a *[7]int
}
for i, _ := range testdata.a {
	// testdata.a is never evaluated; len(testdata.a) is constant
	// i ranges from 0 to 6
	f(i)
}
```

```
var a [10]string
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{}  // element type of m is assignable to val
m := map[string]int{"mon":0, "tue":1, "wed":2, "thu":3, "fri":4, "sat":5, "sun":6}
for key, val = range m {
	h(key, val)
}
// key == last map key encountered in iteration
// val == map[key]

var ch chan Work = producer()
for w := range ch {
	doWork(w)
}

// empty a channel
for range ch {}
```

Go statements

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

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

The expression must be a function or method call; it cannot be parenthesized. Calls of built-in functions are restricted as for expression statements.

The function value and parameters are evaluated as usual in the calling goroutine, but unlike with a regular call, program execution does not wait for the invoked function to complete. Instead, the function begins executing independently in a new goroutine. When the function terminates, its goroutine also terminates. If the function has any return values, they are discarded when the function completes.

```
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 send or receive operations will proceed. It looks similar to a "switch" statement but with the cases all referring to communication operations.

```
SelectStmt = "select" "{" { CommClause } "}" .
CommClause = CommCase ":" StatementList .
CommCase   = "case" ( SendStmt | RecvStmt ) | "default" .
RecvStmt   = [ ExpressionList "=" | IdentifierList ":=" ] RecvExpr .
RecvExpr   = Expression .
```

A case with a RecvStmt may assign the result of a RecvExpr to one or two variables, which may be declared using a short variable declaration. The RecvExpr must be a (possibly parenthesized) receive operation. There can be at most one default case and it may appear anywhere in the list of cases.

Execution of a "select" statement proceeds in several steps:

1. For all the cases in the statement, the channel operands of receive operations and the channel and right-hand-side expressions of send statements are evaluated exactly once, in source order, upon entering the "select" statement. The result is a set of channels to receive from or send to, and the corresponding values to send. Any side effects in that evaluation will occur irrespective of which (if any) communication operation is selected to proceed. Expressions on the left-hand side of a RecvStmt with a short variable declaration or assignment are not yet evaluated.
2. If one or more of the communications can proceed, a single one that can proceed is chosen via a uniform pseudo-random selection. Otherwise, if there is a default case, that case is chosen. If there is no default case, the "select" statement blocks until at least one of the communications can proceed.

3. Unless the selected case is the default case, the respective communication operation is executed.
4. If the selected case is a RecvStmt with a short variable declaration or an assignment, the left-hand side expressions are evaluated and the received value (or values) are assigned.
5. The statement list of the selected case is executed.

Since communication on `nil` channels can never proceed, a select with only `nil` channels and no default case blocks forever.

```
var a []int
var c, c1, c2, c3, c4 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")
case i3, ok := (<-c3):  // same as: i3, ok := <-c3
	if ok {
		print("received ", i3, " from c3\n")
	} else {
		print("c3 is closed\n")
	}
case a[f()] = <-c4:
	// same as:
	// case t := <-c4
	//	a[f()] = t
default:
	print("no communication\n")
}

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

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

Return statements

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

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

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

```
func noResult() {
	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 simpleF() int {
   	return 2
   }

   func complexF1() (re float64, im float64) {
   	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 complexF2() (re float64, im float64) {
   ```

```
		return complexF1()
	}
```

3. The expression list may be empty if the function's result type specifies names for its result parameters. 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 complexF3() (re float64, im float64) {
		re = 7.0
		im = 4.0
		return
	}

	func (devnull) Write(p []byte) (n int, _ error) {
		n = len(p)
		return
	}
```

Regardless of how they are declared, all the result values are initialized to the zero values for their type upon entry to the function. A "return" statement that specifies results sets the result parameters before any deferred functions are executed.

Implementation restriction: A compiler may disallow an empty expression list in a "return" statement if a different entity (constant, type, or variable) with the same name as a result parameter is in scope at the place of the return.

```
func f(n int) (res int, err error) {
	if _, err := f(n-1); err != nil {
		return  // invalid return statement: err is shadowed
	}
	return
}
```

Break statements

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

```
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.

```
OuterLoop:
	for i = 0; i < n; i++ {
		for j = 0; j < m; j++ {
			switch a[i][j] {
			case nil:
				state = Error
				break OuterLoop
			case item:
				state = Found
				break OuterLoop
			}
		}
	}
```

Continue statements

A "continue" statement begins the next iteration of the innermost enclosing "for" loop by advancing control to the end of the loop block. The "for" loop must be within the same function.

```
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.

```
RowLoop:
	for y, row := range rows {
		for x, data := range row {
			if data == endOfRow {
				continue RowLoop
			}
			row[x] = data + bias(x, y)
```

```
		}
	}
```

Goto statements

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

```
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`.

A "goto" statement outside a block cannot jump to a label inside that block. For instance, this example:

```
if n%2 == 1 {
	goto L1
}
for n > 0 {
	f()
	n--
L1:
	f()
	n--
}
```

is erroneous because the label `L1` is inside the "for" statement's block but the `goto` is not.

Fallthrough statements

A "fallthrough" statement transfers control to the first statement of the next case clause in an expression "switch" statement. It may be used only as the final non-empty statement in such a clause.

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

Defer statements

A "defer" statement invokes a function whose execution is deferred to the moment the surrounding function returns, either because the surrounding function executed a return statement, reached the end of its function body, or because the corresponding goroutine is panicking.

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

The expression must be a function or method call; it cannot be parenthesized. Calls of built-in functions are restricted as for expression statements.

Each time a "defer" statement executes, the function value and parameters to the call are evaluated as usual and saved anew but the actual function is not invoked. Instead, deferred functions are invoked immediately before the surrounding function returns, in the reverse order they were deferred. That is, if the surrounding function returns through an explicit return statement, deferred functions are executed *after* any result parameters are set by that return statement but *before* the function returns to its caller. If a deferred function value evaluates to `nil`, execution panics when the function is invoked, not when the "defer" statement is executed.

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. If the deferred function has any return values, they are discarded when the function completes. (See also the section on handling panics.)

```
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 42
func f() (result int) {
	defer func() {
		// result is accessed after it was set to 6 by the return statement
		result *= 7
	}()
	return 6
}
```

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.

Appending to and copying slices

The built-in functions `append` and `copy` assist in common slice operations. For both functions, the result is independent of whether the memory referenced by the arguments overlaps.

The variadic function `append` appends zero or more values `x` to a slice `s` and returns the resulting slice of the same type as `s`. The core type of `s` must be a slice of type `[]E`. The values `x` are passed to a parameter of type `...E` and the respective parameter passing rules apply. As a special case, if the core type of `s` is `[]byte`, `append` also accepts a second argument with core type `bytestring` followed by `...`. This form appends the bytes of the byte slice or string.

```
append(s S, x ...E) S  // core type of S is []E
```

If the capacity of s is not large enough to fit the additional values, append allocates a new, sufficiently large underlying array that fits both the existing slice elements and the additional values. Otherwise, append re-uses the underlying array.

```
s0 := []int{0, 0}
s1 := append(s0, 2)                // append a single element     s1 is []int{0, 0, 2}
s2 := append(s1, 3, 5, 7)          // append multiple elements    s2 is []int{0, 0, 2, 3, 5, 7}
s3 := append(s2, s0...)            // append a slice              s3 is []int{0, 0, 2, 3, 5, 7, 0, 0}
s4 := append(s3[3:6], s3[2:]...)   // append overlapping slice    s4 is []int{3, 5, 7, 2, 3, 5, 7, 0, 0}

var t []interface{}
t = append(t, 42, 3.1415, "foo")   //                             t is []interface{}{42, 3.1415, "foo"}

var b []byte
b = append(b, "bar"...)            // append string contents      b is []byte{'b', 'a', 'r' }
```

The function copy copies slice elements from a source `src` to a destination `dst` and returns the number of elements copied. The core types of both arguments must be slices with identical element type. The number of elements copied is the minimum of `len(src)` and `len(dst)`. As a special case, if the destination's core type is `[]byte`, copy also accepts a source argument with core type `bytestring`. This form copies the bytes from the byte slice or string into the byte slice.

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

Examples:

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
var b = make([]byte, 5)
n1 := copy(s, a[0:])            // n1 == 6, s is []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])            // n2 == 4, s is []int{2, 3, 4, 5, 4, 5}
n3 := copy(b, "Hello, World!")  // n3 == 5, b is []byte("Hello")
```

Clear

The built-in function `clear` takes an argument of map, slice, or type parameter type, and deletes or zeroes out all elements.

```
Call        Argument type     Result

clear(m)    map[K]T           deletes all entries, resulting in an
                              empty map (len(m) == 0)

clear(s)    []T               sets all elements up to the length of
                              s to the zero value of T

clear(t)    type parameter    see below
```

If the type of the argument to `clear` is a type parameter, all types in its type set must be maps or slices, and `clear` performs the operation corresponding to the actual type argument.

If the map or slice is `nil`, `clear` is a no-op.

Close

For an argument `ch` with a core type that is a channel, the built-in function `close` records that no more values will be sent on the channel. It is an error if `ch` is a receive-only channel. Sending to or closing a closed channel causes a run-time panic. Closing the nil channel also causes a run-time panic. 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. The multi-valued receive operation returns a received value along with an indication of whether the channel is closed.

Manipulating complex numbers

Three functions assemble and disassemble complex numbers. The built-in function `complex` 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.

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

The type of the arguments and return value correspond. For `complex`, 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: `complex64` for `float32` arguments, and `complex128` for `float64` arguments. If one of the arguments evaluates to an untyped constant, it is first implicitly converted to the type of the other argument. If both arguments evaluate to untyped constants, they must be non-complex numbers or their imaginary parts must be zero, and the return value of the function is an untyped complex constant.

For `real` and `imag`, the argument must be of complex type, and the return type is the corresponding floating-point type: `float32` for a `complex64` argument, and `float64` for a `complex128` argument. If the argument evaluates to an untyped constant, it must be a number, and the return value of the function is an untyped floating-point constant.

The `real` and `imag` functions together form the inverse of `complex`, so for a value `z` of a complex type `Z`, `z == Z(complex(real(z), imag(z)))`.

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

```
var a = complex(2, -2)             // complex128
const b = complex(1.0, -1.4)       // untyped complex constant 1 - 1.4i
x := float32(math.Cos(math.Pi/2))  // float32
var c64 = complex(5, -x)           // complex64
var s int = complex(1, 0)          // untyped complex constant 1 + 0i can be converted to int
_ = complex(1, 2<<s)               // illegal: 2 assumes floating-point type, cannot shift
var rl = real(c64)                 // float32
var im = imag(a)                   // float64
const c = imag(b)                  // untyped constant -1.4
_ = imag(3 << s)                   // illegal: 3 assumes complex type, cannot shift
```

Arguments of type parameter type are not permitted.

Deletion of map elements

The built-in function `delete` removes the element with key `k` from a map `m`. The value `k` must be assignable to the key type of `m`.

```
delete(m, k)  // remove element m[k] from map m
```

If the type of `m` is a type parameter, all types in that type set must be maps, and they must all have identical key types.

If the map `m` is `nil` or the element `m[k]` does not exist, `delete` is a no-op.

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 defined keys)
          chan T           number of elements queued in channel buffer
          type parameter   see below

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

If the argument type is a type parameter `P`, the call `len(e)` (or `cap(e)` respectively) must be valid for each type in `P`'s type set. The result is the length (or capacity, respectively) of the argument whose type corresponds to the type argument with which `P` was instantiated.

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 of a `nil` slice, map or channel is 0. The capacity of a `nil` slice or channel is 0.

The expression `len(s)` is constant if `s` is a string constant. The expressions `len(s)` and `cap(s)` are constants if the type of `s` is an array or pointer to an array and the expression `s` does not contain channel receives or (non-constant) function calls; in this case `s` is not evaluated. Otherwise, invocations of `len` and `cap` are not constant and `s` is evaluated.

```
const (
	c1 = imag(2i)                    // imag(2i) = 2.0 is a constant
	c2 = len([10]float64{2})         // [10]float64{2} contains no function calls
	c3 = len([10]float64{c1})        // [10]float64{c1} contains no function calls
	c4 = len([10]float64{imag(2i)})  // imag(2i) is a constant and no function call is issued
	c5 = len([10]float64{imag(z)})   // invalid: imag(z) is a (non-constant) function call
)
var z complex128
```

Making slices, maps and channels

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

```
Call             Core type    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 space for approximately n elements

make(T)          channel      unbuffered channel of type T
make(T, n)       channel      buffered channel of type T, buffer size n
```

Each of the size arguments `n` and `m` must be of integer type, have a type set containing only integer types, or be an untyped constant. A constant size argument must be non-negative and representable by a value of type `int`; if it is an untyped constant it is given type `int`. If both `n` and `m` are provided and are constant, then `n` must be no larger than `m`. For slices and channels, if `n` is negative or larger than `m` at run time, a run-time panic occurs.

```
s := make([]int, 10, 100)       // slice with len(s) == 10, cap(s) == 100
s := make([]int, 1e3)           // slice with len(s) == cap(s) == 1000
s := make([]int, 1<<63)         // illegal: len(s) is not representable by a value of type int
s := make([]int, 10, 0)         // illegal: len(s) > cap(s)
c := make(chan int, 10)         // channel with a buffer size of 10
m := make(map[string]int, 100)  // map with initial space for approximately 100 elements
```

Calling `make` with a map type and size hint `n` will create a map with initial space to hold `n` map elements. The precise behavior is implementation-dependent.

Min and max

The built-in functions `min` and `max` compute the smallest—or largest, respectively—value of a fixed number of arguments of ordered types. There must be at least one argument.

The same type rules as for operators apply: for ordered arguments `x` and `y`, `min(x, y)` is valid if `x + y` is valid, and the type of `min(x, y)` is the type of `x + y` (and similarly for `max`). If all arguments are constant, the result is constant.

```
var x, y int
m := min(x)                 // m == x
m := min(x, y)              // m is the smaller of x and y
m := max(x, y, 10)          // m is the larger of x and y but at least 10
c := max(1, 2.0, 10)        // c == 10.0 (floating-point kind)
f := max(0, float32(x))     // type of f is float32
var s []string
_ = min(s...)               // invalid: slice arguments are not permitted
t := max("", "foo", "bar")  // t == "foo" (string kind)
```

For numeric arguments, assuming all NaNs are equal, `min` and `max` are commutative and associative:

```
min(x, y)    == min(y, x)
min(x, y, z) == min(min(x, y), z) == min(x, min(y, z))
```

For floating-point arguments negative zero, NaN, and infinity the following rules apply:

```
   x        y    min(x, y)    max(x, y)

  -0.0    0.0         -0.0          0.0    // negative zero is smaller than (non-negative) zero
  -Inf      y         -Inf            y    // negative infinity is smaller than any other number
  +Inf      y            y         +Inf    // positive infinity is larger than any other number
   NaN      y          NaN          NaN    // if any argument is a NaN, the result is a NaN
```

For string arguments the result for `min` is the first argument with the smallest (or for `max`, largest) value, compared lexically byte-wise:

```
min(x, y)    == if x <= y then x else y
min(x, y, z) == min(min(x, y), z)
```

Allocation

The built-in function `new` takes a type `T`, allocates storage for a variable of that type at run time, and returns a value of type `*T` pointing to it. The variable is initialized as described in the section on initial values.

```
new(T)
```

For instance

```
type S struct { a int; b float64 }
new(S)
```

allocates storage 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 location.

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{}
```

While executing a function `F`, an explicit call to `panic` or a run-time panic terminates the execution of `F`. Any functions deferred by `F` are then executed as usual. Next, any deferred functions run by `F`'s caller are run, and so on up to any deferred by the top-level function in the executing goroutine. 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*.

```
panic(42)
panic("unreachable")
panic(Error("cannot parse"))
```

The `recover` function allows a program to manage behavior of a panicking goroutine. Suppose a function `G` defers a function `D` that calls `recover` and a panic occurs in a function on the same goroutine in which `G` is executing. When the running of deferred functions reaches `D`, the return value of `D`'s call to `recover` will be the value passed to the call of `panic`. If `D` returns normally, without starting a new `panic`, the panicking sequence stops. In that case, the state of functions called between `G` and the call to `panic` is discarded, and normal execution resumes. Any functions deferred by `G` before `D` are then run and `G`'s execution terminates by returning to its caller.

The return value of `recover` is `nil` when the goroutine is not panicking or `recover` was not called directly by a deferred function. Conversely, if a goroutine is panicking and `recover` was called directly by a deferred function, the return value of `recover` is guaranteed not to be `nil`. To ensure this, calling `panic` with a `nil` interface value (or an untyped `nil`) causes a run-time panic.

The `protect` function in the example below invokes the function argument `g` and protects callers from run-time panics raised by `g`.

```
func protect(g func()) {
	defer func() {
		log.Println("done")  // Println executes normally even if there is a panic
		if x := recover(); x != nil {
			log.Printf("run time panic: %v", x)
		}
	}()
	log.Println("start")
```

```
        g()
}
```

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
```

Implementation restriction: `print` and `println` need not accept arbitrary argument types, but printing of boolean, numeric, and string types must be supported.

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 depends on functionality of the *imported* package (§Program initialization and execution) and enables access to exported identifiers of that package. The import names an identifier (PackageName) to be used for access and an ImportPath that specifies the package to be imported.

```
ImportDecl       = "import" ( ImportSpec | "(" { ImportSpec ";" } ")" ) .
ImportSpec       = [ "." | PackageName ] ImportPath .
ImportPath       = string_lit .
```

The PackageName is used in qualified identifiers to access 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 declared in that package's package block will be declared in the importing source file's file block and must 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.

Implementation restriction: A compiler may restrict ImportPaths to non-empty strings using only characters belonging to Unicode's L, M, N, P, and S general categories (the Graphic characters without spaces) and may also exclude the characters `!"#$%&'()*,:;<=>?[\]^`{|}`

and the Unicode replacement character U+FFFD.

Consider a 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` is 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, directly or indirectly, or to directly 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 received 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 subprocess.
	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 storage is allocated for a variable, either through a declaration or a call of `new`, or when a new value is created, either through a composite literal or a call of `make`, and no explicit initialization is provided, the variable or value is given a default value. Each element of such a variable or value is set to the *zero value* for its type: `false` for booleans, `0` for numeric types, `""` 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 float64; 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
```

Package initialization

Within a package, package-level variable initialization proceeds stepwise, with each step selecting the variable earliest in *declaration order* which has no dependencies on uninitialized variables.

More precisely, a package-level variable is considered *ready for initialization* if it is not yet initialized and either has no initialization expression or its initialization expression has no *dependencies* on uninitialized variables. Initialization proceeds by repeatedly initializing the next package-level variable that is earliest in declaration order and ready for initialization, until there are no variables ready for initialization.

If any variables are still uninitialized when this process ends, those variables are part of one or more initialization cycles, and the program is not valid.

Multiple variables on the left-hand side of a variable declaration initialized by single (multi-valued) expression on the right-hand side are initialized together: If any of the variables on the left-hand side is initialized, all those variables are initialized in the same step.

```
var x = a
var a, b = f() // a and b are initialized together, before x is initialized
```

For the purpose of package initialization, blank variables are treated like any other variables in declarations.

The declaration order of variables declared in multiple files is determined by the order in which the files are presented to the compiler: Variables declared in the first file are declared before any of the variables declared in the second file, and so on. To ensure reproducible initialization behavior, build systems are encouraged to present multiple files belonging to the same package in lexical file name order to a compiler.

Dependency analysis does not rely on the actual values of the variables, only on lexical *references* to them in the source, analyzed transitively. For instance, if a variable `x`'s initialization expression refers to a function whose body refers to variable `y` then `x` depends on `y`. Specifically:

- A reference to a variable or function is an identifier denoting that variable or function.
- A reference to a method `m` is a method value or method expression of the form `t.m`, where the (static) type of `t` is not an interface type, and the method `m` is in the method set of `t`. It is immaterial whether the resulting function value `t.m` is invoked.
- A variable, function, or method `x` depends on a variable `y` if `x`'s initialization expression or body (for functions and methods) contains a reference to `y` or to a function or method that depends on `y`.

For example, given the declarations

```
var (
	a = c + b  // == 9
	b = f()    // == 4
	c = f()    // == 5
	d = 3      // == 5 after initialization has finished
)

func f() int {
```

```
        d++
        return d
}
```

the initialization order is d, b, c, a. Note that the order of subexpressions in initialization expressions is irrelevant: a = c + b and a = b + c result in the same initialization order in this example.

Dependency analysis is performed per package; only references referring to variables, functions, and (non-interface) methods declared in the current package are considered. If other, hidden, data dependencies exists between variables, the initialization order between those variables is unspecified.

For instance, given the declarations

```
var x = I(T{}).ab()   // x has an undetected, hidden dependency on a and b
var _ = sideEffect()  // unrelated to x, a, or b
var a = b
var b = 42

type I interface      { ab() []int }
type T struct{}
func (T) ab() []int   { return []int{a, b} }
```

the variable a will be initialized after b but whether x is initialized before b, between b and a, or after a, and thus also the moment at which `sideEffect()` is called (before or after x is initialized) is not specified.

Variables may also be initialized using functions named `init` declared in the package block, with no arguments and no result parameters.

```
func init() { … }
```

Multiple such functions may be defined per package, even within a single source file. In the package block, the `init` identifier can be used only to declare `init` functions, yet the identifier itself is not declared. Thus `init` functions cannot be referred to from anywhere in a program.

The entire package is initialized by assigning initial values to all its package-level variables followed by calling all `init` functions in the order they appear in the source, possibly in multiple files, as presented to the compiler.

Program initialization

The packages of a complete program are initialized stepwise, one package at a time. If a package has imports, the imported packages are initialized before initializing the package itself. If multiple packages import a package, the imported package will be initialized only once. The importing of packages, by construction, guarantees that there can be no cyclic initialization dependencies. More precisely:

Given the list of all packages, sorted by import path, in each step the first uninitialized package in the list for which all imported packages (if any) are already initialized is initialized. This step is repeated until all packages are initialized.

Package initialization—variable initialization and the invocation of `init` functions—happens in a single goroutine, sequentially, one package at a time. An `init` function may launch other goroutines, which can run concurrently with the initialization code. However, initialization always sequences the `init` functions: it will not invoke the next one until the previous one has returned.

Program execution

A complete program is created by linking a single, unimported package called the *main package* with all the packages it imports, transitively. The main package must have package name `main` and declare a function `main` that takes no arguments and returns no value.

```
func main() { … }
```

Program execution begins by initializing the program and then invoking the function `main` in package `main`. When that function invocation returns, the program exits. It does not wait for other (non-`main`) goroutines to complete.

Errors

The predeclared type `error` is defined as

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

```
}
```

It is the conventional interface for representing an error condition, with the nil value representing no error. For instance, a function to read data from a file might be defined:

```
func Read(f *File, b []byte) (n int, err error)
```

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 satisfies the predeclared interface type `error`. The exact error values that represent distinct run-time error conditions are unspecified.

```
package runtime

type Error interface {
	error
	// and perhaps other methods
}
```

System considerations

Package `unsafe`

The built-in package `unsafe`, known to the compiler and accessible through the import path `"unsafe"`, provides facilities for low-level programming including operations that violate the type system. A package using `unsafe` must be vetted manually for type safety and may not be portable. 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) uintptr
func Offsetof(selector ArbitraryType) uintptr
func Sizeof(variable ArbitraryType) uintptr

type IntegerType int  // shorthand for an integer type; it is not a real type
func Add(ptr Pointer, len IntegerType) Pointer
func Slice(ptr *ArbitraryType, len IntegerType) []ArbitraryType
func SliceData(slice []ArbitraryType) *ArbitraryType
func String(ptr *byte, len IntegerType) string
func StringData(str string) *byte
```

A `Pointer` is a pointer type but a `Pointer` value may not be dereferenced. Any pointer or value of underlying type `uintptr` can be converted to a type of underlying type `Pointer` and vice versa. The effect of converting between `Pointer` and `uintptr` is implementation-defined.

```
var f float64
bits = *(*uint64)(unsafe.Pointer(&f))

type ptr unsafe.Pointer
bits = *(*uint64)(ptr(&f))

var p ptr = nil
```

The functions `Alignof` and `Sizeof` take an expression `x` of any type and return the alignment or size, respectively, of a hypothetical variable `v` as if `v` was declared via `var v = x`.

The function `Offsetof` takes a (possibly parenthesized) selector `s.f`, denoting a field `f` of the struct denoted by `s` or `*s`, and returns the field offset in bytes relative to the struct's address. If `f` is an embedded field, it must be reachable without pointer indirections through fields of the struct. For a struct `s` with field `f`:

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

A (variable of) type `T` has *variable size* if `T` is a type parameter, or if it is an array or struct type containing elements or fields of variable size. Otherwise the size is *constant*. Calls to `Alignof`, `Offsetof`, and `Sizeof` are compile-time constant expressions of type `uintptr` if their arguments (or the struct `s` in the selector expression `s.f` for `Offsetof`) are types of constant size.

The function `Add` adds `len` to `ptr` and returns the updated pointer `unsafe.Pointer(uintptr(ptr) + uintptr(len))`. The `len` argument must be of integer type or an untyped constant. A constant `len` argument must be representable by a value of type `int`; if it is an untyped constant it is given type `int`. The rules for valid uses of `Pointer` still apply.

The function `Slice` returns a slice whose underlying array starts at `ptr` and whose length and capacity are `len`. `Slice(ptr, len)` is equivalent to

```
(*[len]ArbitraryType)(unsafe.Pointer(ptr))[:]
```

except that, as a special case, if `ptr` is `nil` and `len` is zero, `Slice` returns `nil`.

The `len` argument must be of integer type or an untyped constant. A constant `len` argument must be non-negative and representable by a value of type `int`; if it is an untyped constant it is given type `int`. At run time, if `len` is negative, or if `ptr` is `nil` and `len` is not zero, a run-time panic occurs.

The function `SliceData` returns a pointer to the underlying array of the `slice` argument. If the slice's capacity `cap(slice)` is not zero, that pointer is `&slice[:1][0]`. If `slice` is `nil`, the result is `nil`. Otherwise it is a non-`nil` pointer to an unspecified memory address.

The function `String` returns a `string` value whose underlying bytes start at `ptr` and whose length is `len`. The same requirements apply to the `ptr` and `len` argument as in the function `Slice`. If `len` is zero, the result is the empty string `""`. Since Go strings are immutable, the bytes passed to `String` must not be modified afterwards.

The function `StringData` returns a pointer to the underlying bytes of the `str` argument. For an empty string the return value is unspecified, and may be `nil`. Since Go strings are immutable, the bytes returned by `StringData` must not be modified.

Size and alignment guarantees

For the 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, complex64     8
complex128                           16
```

The following minimal alignment properties are guaranteed:

1. For a variable `x` of any type: `unsafe.Alignof(x)` is at least 1.
2. 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.
3. For a variable `x` of array type: `unsafe.Alignof(x)` is the same as the alignment of a variable of the array's element type.

A struct or array type has size zero if it contains no fields (or elements, respectively) that have a size greater than zero. Two distinct zero-size variables may have the same address in memory.

Appendix

Type unification rules

The type unification rules describe if and how two types unify. The precise details are relevant for Go implementations, affect the specifics of error messages (such as whether a compiler reports a type inference or other error), and may explain why type inference fails in unusual code situations. But by and large these rules can be ignored when writing Go code: type inference is designed to mostly "work as expected", and the unification rules are fine-tuned accordingly.

Type unification is controlled by a *matching mode*, which may be *exact* or *loose*. As unification recursively descends a composite type structure, the matching mode used for elements of the type, the *element matching mode*, remains the same as the matching mode except when two types are unified for assignability ($\equiv_A$): in this case, the matching mode is *loose* at the top level but then changes to *exact* for element types, reflecting the fact that types don't have to be identical to be assignable.

Two types that are not bound type parameters unify exactly if any of following conditions is true:

- Both types are identical.
- Both types have identical structure and their element types unify exactly.
- Exactly one type is an unbound type parameter with a core type, and that core type unifies with the other type per the unification rules for $\equiv_A$ (loose unification at the top level and exact unification for element types).

If both types are bound type parameters, they unify per the given matching modes if:

- Both type parameters are identical.
- At most one of the type parameters has a known type argument. In this case, the type parameters are *joined*: they both stand for the same type argument. If neither type parameter has a known type argument yet, a future type argument inferred for one the type parameters is simultaneously inferred for both of them.
- Both type parameters have a known type argument and the type arguments unify per the given matching modes.

A single bound type parameter `P` and another type `T` unify per the given matching modes if:

- `P` doesn't have a known type argument. In this case, `T` is inferred as the type argument for `P`.
- `P` does have a known type argument `A`, `A` and `T` unify per the given matching modes, and one of the following conditions is true:
    - Both `A` and `T` are interface types: In this case, if both `A` and `T` are also defined types, they must be identical. Otherwise, if neither of them is a defined type, they must have the same number of methods (unification of `A` and `T` already established that the methods match).
    - Neither `A` nor `T` are interface types: In this case, if `T` is a defined type, `T` replaces `A` as the inferred type argument for `P`.

Finally, two types that are not bound type parameters unify loosely (and per the element matching mode) if:

- Both types unify exactly.
- One type is a defined type, the other type is a type literal, but not an interface, and their underlying types unify per the element matching mode.
- Both types are interfaces (but not type parameters) with identical type terms, both or neither embed the predeclared type comparable, corresponding method types unify per the element matching mode, and the method set of one of the interfaces is a subset of the method set of the other interface.
- Only one type is an interface (but not a type parameter), corresponding methods of the two types unify per the element matching mode, and the method set of the interface is a subset of the method set of the other type.
- Both types have the same structure and their element types unify per the element matching mode.