The Go Programming Language Specification V1.25 (Vietnamese by Jang - Ver 2)

Introduction

Đây là tài liệu tham khảo cho ngôn ngữ lập trình Go. Để biết thêm thông tin và các tài liệu khác, hãy xem go.dev. Go là một ngôn ngữ đa mục đích được thiết kế với mục tiêu lập trình hệ thống. Ngôn ngữ này có kiểu dữ liệu mạnh, thu gom rác tự động và hỗ trợ rõ ràng cho lập trình đồng thời. Các chương trình được xây dựng từ *packages* (gói), với các thuộc tính cho phép quản lý phụ thuộc hiệu quả. Cú pháp của Go ngắn gọn và đơn giản để phân tích, giúp các công cụ tự động như môi trường phát triển tích hợp (IDE) dễ dàng xử lý.

Notation

Cú pháp được mô tả bằng một biến thể của 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 "}" .
```

Các production là các biểu thức được xây dựng từ các thành phần và các toán tử sau, theo thứ tự ưu tiên tăng dần:

```
|   alternation
()  grouping
[]  option (0 hoặc 1 lần)
{}  repetition (0 đến n lần)
```

Tên production viết thường dùng để xác định token từ vựng (lexical/terminal). Non-terminals viết CamelCase. Token từ vựng được đặt trong dấu nháy kép `""` hoặc dấu back quote `` ` ``.

Dạng `a … b` biểu diễn tập hợp các ký tự từ `a` đến `b` như các lựa chọn thay thế. Dấu ba chấm ngang … cũng được sử dụng ở các phần khác trong đặc tả để chỉ các liệt kê hoặc đoạn mã không được mô tả chi tiết. Ký tự … (khác với ba ký tự `...`) không phải là token của ngôn ngữ Go.

Liên kết dạng [Go 1.xx] cho biết một tính năng ngôn ngữ (hoặc một khía cạnh của nó) đã được thay đổi hoặc thêm vào từ phiên bản Go 1.xx và do đó yêu cầu ít nhất phiên bản đó để biên dịch. Xem chi tiết tại phần liên kết trong phụ lục.

Source code representation

Mã nguồn là văn bản Unicode được mã hóa bằng UTF-8. Văn bản không được chuẩn hóa, vì vậy một ký tự có dấu có thể khác với cùng ký tự được tạo từ việc kết hợp dấu và chữ cái; chúng được coi là hai mã ký tự khác nhau. Để đơn giản, tài liệu này sẽ dùng thuật ngữ *character* để chỉ một mã ký tự Unicode trong văn bản nguồn.

Mỗi mã ký tự là riêng biệt; ví dụ, chữ hoa và chữ thường là các ký tự khác nhau.

Giới hạn triển khai: Để tương thích với các công cụ khác, trình biên dịch có thể không cho phép ký tự NUL (U+0000) trong văn bản nguồn.

Giới hạn triển khai: Để tương thích với các công cụ khác, trình biên dịch có thể bỏ qua byte order mark (U+FEFF) được mã hóa UTF-8 nếu nó là mã ký tự Unicode đầu tiên trong văn bản nguồn. Byte order mark có thể không được phép xuất hiện ở bất kỳ vị trí nào khác trong nguồn.

Characters

Các thuật ngữ sau được dùng để chỉ các loại ký tự Unicode cụ thể:

```
newline        = /* mã ký tự Unicode U+000A */ .
unicode_char   = /* một mã ký tự Unicode bất kỳ trừ
newline */ .
unicode_letter = /* một mã ký tự Unicode được phân
loại là "Letter" */ .
unicode_digit  = /* một mã ký tự Unicode được phân
loại là "Number, decimal digit" */ .
```

Trong The Unicode Standard 8.0, Mục 4.5 "General Category" định nghĩa một tập hợp các loại ký tự. Go coi tất cả các ký tự thuộc các loại Letter: Lu, Ll, Lt, Lm, hoặc Lo là unicode_letter, và các ký tự thuộc loại Number: Nd là unicode_digit.

Letters and digits

Ký tự gạch dưới _ (U+005F) được coi là một chữ cái viết thường.

```
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

Chú thích dùng để tài liệu hóa chương trình. Có hai dạng:

1. *Chú thích dòng* bắt đầu với chuỗi ký tự `//` và kết thúc ở cuối dòng.
2. *Chú thích tổng quát* bắt đầu với chuỗi ký tự `/*` và kết thúc với chuỗi ký tự `*/` đầu tiên sau đó.

Chú thích không thể bắt đầu bên trong rune hoặc string literal, hoặc bên trong một chú thích khác. Một chú thích tổng quát không chứa ký tự xuống dòng sẽ được coi như một khoảng trắng. Các chú thích khác được coi như một ký tự xuống dòng.

Tokens

Token tạo thành từ vựng của ngôn ngữ Go. Có bốn loại: *identifiers*, *keywords*, *operators and punctuation*, và *literals*. *White space* (khoảng trắng), bao gồm dấu cách (U+0020), tab ngang (U+0009), xuống dòng (U+000A), và trả về đầu dòng (U+000D), sẽ bị bỏ qua trừ khi nó phân tách các token mà nếu không sẽ bị gộp thành một token duy nhất. Ngoài ra, một ký tự xuống dòng hoặc kết thúc tệp có thể kích hoạt việc chèn dấu chấm phẩy. Khi phân tách đầu vào thành các token, token tiếp theo là chuỗi ký tự dài nhất tạo thành một token hợp lệ.

Semicolons

Cú pháp chính thức sử dụng dấu chấm phẩy `";"` làm ký tự kết thúc trong một số production. Các chương trình Go có thể bỏ qua hầu hết

các dấu chấm phẩy này nhờ hai quy tắc sau:

1. Khi đầu vào được phân tách thành các token, một dấu chấm phẩy sẽ tự động được chèn vào luồng token ngay sau token cuối cùng của dòng nếu token đó là
   - một identifier
   - một integer, floating-point, imaginary, rune, hoặc string literal
   - một trong các keywords `break`, `continue`, `fallthrough`, hoặc `return`
   - một trong các operators and punctuation `++`, `--`, `)`, `]`, hoặc `}`
2. Để cho phép các câu lệnh phức tạp nằm trên một dòng, có thể bỏ qua dấu chấm phẩy trước dấu đóng `")"` hoặc `"}"`.

Để phản ánh cách sử dụng theo phong cách Go, các ví dụ mã trong tài liệu này sẽ lược bỏ dấu chấm phẩy theo các quy tắc trên.

Identifiers

Identifier dùng để đặt tên cho các thực thể chương trình như biến và kiểu dữ liệu. Một identifier là một chuỗi gồm một hoặc nhiều chữ cái và chữ số. Ký tự đầu tiên phải là một chữ cái.

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

```
a
_x9
ThisVariableIsExported
αβ
```

Một số identifier được định nghĩa sẵn.

Keywords

Các từ khóa sau được dành riêng và không được sử dụng làm identifier.

```
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

Các chuỗi ký tự sau đại diện cho toán tử (bao gồm toán tử gán) và dấu câu [Go 1.18]:

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

Integer literals

Một integer literal là một chuỗi các chữ số biểu diễn một hằng số số nguyên. Một tiền tố tùy chọn xác định cơ số không phải thập phân: `0b` hoặc `0B` cho nhị phân, `0`, `0o`, hoặc `0O` cho bát phân, và `0x` hoặc `0X` cho thập lục phân [Go 1.13]. Một `0` đơn lẻ được coi là số không thập phân. Trong các literal thập lục phân, các chữ cái `a` đến `f` và `A` đến `F` đại diện cho giá trị từ 10 đến 15.

Để dễ đọc, ký tự gạch dưới _ có thể xuất hiện sau tiền tố cơ số hoặc giữa các chữ số liên tiếp; các dấu gạch dưới này không làm thay đổi giá trị của literal.

```
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       // ký tự thứ hai là chữ 'O' in hoa
0xBadFace
0xBad_Face
0x_67_7a_2f_cc_40_c6
170141183460469231731687303715884105727
170_141183_460469_231731_687303_715884_105727

_42         // một identifier, không phải integer
literal
42_         // không hợp lệ: _ phải nằm giữa các
chữ số liên tiếp
4__2        // không hợp lệ: chỉ được phép một _
liên tiếp
0_xBadFace  // không hợp lệ: _ phải nằm giữa các
chữ số liên tiếp
```

Floating-point literals

Một floating-point literal là biểu diễn thập phân hoặc thập lục phân của một hằng số số thực.

Floating-point literal thập phân gồm phần nguyên (chữ số thập phân), dấu chấm thập phân, phần thập phân (chữ số thập phân), và phần mũ (`e` hoặc `E` theo sau là dấu tùy chọn và chữ số thập phân). Một trong hai phần nguyên hoặc phần thập phân có thể bị lược bỏ; một trong hai dấu chấm thập phân hoặc phần mũ có thể bị lược bỏ. Giá trị mũ exp sẽ nhân mantissa (phần nguyên và phần thập phân) với 10^exp.

Floating-point literal thập lục phân gồm tiền tố `0x` hoặc `0X`, phần nguyên (chữ số thập lục phân), dấu chấm cơ số, phần thập phân (chữ số thập lục phân), và phần mũ (`p` hoặc `P` theo sau là dấu tùy chọn và chữ số thập phân). Một trong hai phần nguyên hoặc phần thập phân có thể bị lược bỏ; dấu chấm cơ số cũng có thể bị lược bỏ, nhưng phần mũ

là bắt buộc. (Cú pháp này giống với IEEE 754-2008 §5.12.3.) Giá trị mũ exp sẽ nhân mantissa với 2^exp [Go 1.13].

Để dễ đọc, ký tự gạch dưới _ có thể xuất hiện sau tiền tố cơ số hoặc giữa các chữ số liên tiếp; các dấu gạch dưới này không làm thay đổi giá trị literal.

```
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 (phép trừ số nguyên)

0x.p1         // không hợp lệ: mantissa không có chữ
số
1p-2          // không hợp lệ: p exponent yêu cầu
mantissa thập lục phân
0x1.5e-2      // không hợp lệ: mantissa thập lục
phân yêu cầu p exponent
1_.5          // không hợp lệ: _ phải nằm giữa các
chữ số liên tiếp
1._5          // không hợp lệ: _ phải nằm giữa các
chữ số liên tiếp
1.5_e1        // không hợp lệ: _ phải nằm giữa các
chữ số liên tiếp
1.5e_1        // không hợp lệ: _ phải nằm giữa các
chữ số liên tiếp
1.5e1_        // không hợp lệ: _ phải nằm giữa các
chữ số liên tiếp
```

Imaginary literals

Một imaginary literal biểu diễn phần ảo của một hằng số phức. Nó gồm một integer hoặc floating-point literal theo sau là chữ cái thường `i`. Giá trị của imaginary literal là giá trị của literal tương ứng nhân với đơn vị ảo *i* [Go 1.13]

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

Để tương thích ngược, phần nguyên của imaginary literal chỉ gồm các chữ số thập phân (và có thể có dấu gạch dưới) sẽ được coi là số nguyên thập phân, ngay cả khi bắt đầu bằng số 0.

```
0i
0123i         // == 123i để tương thích ngược
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

Một rune literal biểu diễn một hằng số rune, là một giá trị số nguyên xác định một mã ký tự Unicode. Rune literal được biểu diễn bằng một hoặc nhiều ký tự đặt trong dấu nháy đơn, như `'x'` hoặc `'n'`. Bên trong dấu nháy, bất kỳ ký tự nào cũng có thể xuất hiện trừ ký tự xuống dòng và dấu nháy đơn chưa được escape. Một ký tự đơn trong dấu nháy biểu diễn giá trị Unicode của chính ký tự đó, trong khi các chuỗi nhiều ký tự bắt đầu bằng dấu gạch chéo ngược encode giá trị ở các định dạng khác nhau.

Dạng đơn giản nhất biểu diễn ký tự duy nhất trong dấu nháy; vì mã nguồn Go là các ký tự Unicode được mã hóa UTF-8, nhiều byte UTF-8 có thể biểu diễn một giá trị số nguyên duy nhất. Ví dụ, literal `'a'` chứa một byte duy nhất biểu diễn ký tự `a`, Unicode U+0061, giá trị `0x61`,

trong khi `'ä'` chứa hai byte (`0xc3 0xa4`) biểu diễn ký tự a-dieresis, U+00E4, giá trị `0xe4`.

Một số escape với dấu gạch chéo ngược cho phép mã hóa giá trị bất kỳ dưới dạng văn bản ASCII. Có bốn cách để biểu diễn giá trị số nguyên dưới dạng hằng số số học: `x` theo sau là đúng hai chữ số thập lục phân; `u` theo sau là đúng bốn chữ số thập lục phân; `U` theo sau là đúng tám chữ số thập lục phân, và một dấu gạch chéo ngược theo sau là đúng ba chữ số bát phân. Trong mỗi trường hợp, giá trị của literal là giá trị được biểu diễn bởi các chữ số ở cơ số tương ứng.

Mặc dù các biểu diễn này đều cho ra một số nguyên, chúng có các phạm vi hợp lệ khác nhau. Escape bát phân phải biểu diễn giá trị từ 0 đến 255. Escape thập lục phân thỏa mãn điều kiện này theo cấu trúc. Escape `u` và `U` biểu diễn mã ký tự Unicode nên một số giá trị là không hợp lệ, đặc biệt là các giá trị trên `0x10FFFF` và các nửa surrogate.

Sau dấu gạch chéo ngược, một số escape một ký tự biểu diễn các giá trị đặc biệt:

```
\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)
```

Một ký tự không nhận diện được sau dấu gạch chéo ngược trong rune literal là không hợp lệ.

```
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

Một string literal biểu diễn một hằng số chuỗi thu được từ việc nối một chuỗi ký tự. Có hai dạng: raw string literal và interpreted string literal.

Raw string literal là chuỗi ký tự nằm giữa hai dấu back quote, như `` `foo` ``. Bên trong dấu back quote, bất kỳ ký tự nào cũng có thể xuất hiện trừ dấu back quote. Giá trị của raw string literal là chuỗi các ký tự không bị xử lý (ngầm định mã hóa UTF-8) giữa hai dấu back quote; đặc biệt, dấu gạch chéo ngược không có ý nghĩa đặc biệt và chuỗi có thể chứa ký tự xuống dòng. Ký tự carriage return ('r') bên trong raw string literal sẽ bị loại bỏ khỏi giá trị chuỗi.

Interpreted string literal là chuỗi ký tự nằm giữa hai dấu nháy kép, như `"bar"`. Bên trong dấu nháy kép, bất kỳ ký tự nào cũng có thể xuất hiện trừ ký tự xuống dòng và dấu nháy kép chưa được escape. Văn bản giữa hai dấu nháy tạo thành giá trị của literal, với các escape được xử lý như trong rune literal (trừ `\'` là không hợp lệ và `\"` là hợp lệ), với các giới hạn tương tự. Escape bát phân ba chữ số (`\`*nnn*) và escape thập lục phân hai chữ số (`\x`*nn*) biểu diễn từng *byte* của chuỗi kết quả; các escape khác biểu diễn (có thể là nhiều byte) mã hóa UTF-8 của từng *ký tự*. Do đó, trong string literal, `\377`và`\xFF`biểu diễn một byte duy nhất có giá trị`0xFF`=255, trong khi `ÿ`, `\u00FF`, `\U000000FF`và`\xc3\xbf`biểu diễn hai byte`0xc3 0xbf` của mã hóa UTF-8 cho ký tự U+00FF.

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

`abc`                // giống như "abc"
`n
n`                   // giống như "nnn"
"n"
"""                  // giống như `"`
"Hello, world!n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
"\uD800"             // không hợp lệ: surrogate
half
"\U00110000"         // không hợp lệ: mã ký tự
Unicode không hợp lệ
```

Các ví dụ sau đều biểu diễn cùng một chuỗi:

```
"日本語"                                 // văn bản
UTF-8
`日本語`                                 // văn bản
UTF-8 dưới dạng raw literal
"\u65e5\u672c\u8a9e"                    // mã ký tự
Unicode rõ ràng
"\U000065e5\U0000672c\U00008a9e"        // mã ký tự
Unicode rõ ràng
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"  // các byte
UTF-8 rõ ràng
```

Nếu mã nguồn biểu diễn một ký tự dưới dạng hai mã ký tự, ví dụ một dạng kết hợp giữa dấu và chữ cái, kết quả sẽ là lỗi nếu đặt trong rune literal (không phải một mã ký tự duy nhất), và sẽ xuất hiện dưới dạng hai mã ký tự nếu đặt trong string literal.

Constants

Có các *hằng boolean*, *hằng rune*, *hằng số nguyên*, *hằng số thực*, *hằng số phức*, và *hằng chuỗi*. Rune, số nguyên, số thực và số phức được gọi chung là *hằng số số học*.

Một giá trị hằng được biểu diễn bởi một rune, số nguyên, số thực, số ảo, hoặc chuỗi literal, một identifier biểu diễn một hằng, một biểu thức hằng, một chuyển đổi với kết quả là một hằng, hoặc giá trị trả về của một số hàm dựng sẵn như `min` hoặc `max` áp dụng cho các đối số hằng, `unsafe.Sizeof` áp dụng cho một số giá trị nhất định, `cap` hoặc `len` áp dụng cho một số biểu thức, `real` và `imag` áp dụng cho hằng số phức và `complex` áp dụng cho các hằng số số học. Giá trị chân lý boolean được biểu diễn bởi các hằng định nghĩa sẵn `true` và `false`. Identifier định nghĩa sẵn iota biểu diễn một hằng số nguyên.

Nói chung, hằng số phức là một dạng của biểu thức hằng và được thảo luận ở phần đó.

Hằng số số học biểu diễn giá trị chính xác với độ chính xác tùy ý và không bị tràn. Do đó, không có hằng số nào biểu diễn các giá trị IEEE 754 như số không âm, vô cực, hoặc không phải số.

Hằng số có thể có kiểu hoặc *không kiểu*. Hằng literal, `true`, `false`, `iota`, và một số biểu thức hằng chỉ chứa các toán hạng hằng không kiểu là không kiểu.

Một hằng có thể được gán kiểu tường minh bởi một khai báo hằng hoặc chuyển đổi, hoặc ngầm định khi sử dụng trong khai báo biến hoặc

câu lệnh gán hoặc như một toán hạng trong biểu thức. Nếu giá trị hằng không thể được biểu diễn như một giá trị của kiểu tương ứng thì sẽ báo lỗi. Nếu kiểu là một tham số kiểu, hằng số sẽ được chuyển thành giá trị không phải hằng của tham số kiểu đó.

Một hằng không kiểu có *kiểu mặc định* là kiểu mà hằng đó sẽ được chuyển ngầm định trong các ngữ cảnh yêu cầu giá trị có kiểu, ví dụ, trong khai báo biến ngắn như `i := 0` khi không có kiểu tường minh. Kiểu mặc định của một hằng không kiểu là `bool`, `rune`, `int`, `float64`, `complex128`, hoặc `string` tương ứng, tùy thuộc vào việc nó là hằng boolean, rune, số nguyên, số thực, phức hay chuỗi.

Giới hạn triển khai: Mặc dù hằng số số học có độ chính xác tùy ý trong ngôn ngữ, trình biên dịch có thể triển khai chúng bằng biểu diễn nội bộ với độ chính xác giới hạn. Tuy nhiên, mọi trình biên dịch phải:

- Biểu diễn hằng số nguyên với ít nhất 256 bit.
- Biểu diễn hằng số thực, bao gồm cả phần của hằng số phức, với mantissa ít nhất 256 bit và số mũ nhị phân có dấu ít nhất 16 bit.
- Báo lỗi nếu không thể biểu diễn chính xác một hằng số nguyên.
- Báo lỗi nếu không thể biểu diễn một hằng số thực hoặc phức do tràn.
- Làm tròn đến hằng số có thể biểu diễn gần nhất nếu không thể biểu diễn một hằng số thực hoặc phức do giới hạn về độ chính xác.

Các yêu cầu này áp dụng cho cả hằng literal và kết quả của việc đánh giá biểu thức hằng.

Variables

Biến là một vị trí lưu trữ để chứa một *giá trị*. Tập hợp các giá trị hợp lệ được xác định bởi *kiểu* của biến.

Một khai báo biến hoặc, với tham số và kết quả hàm, chữ ký của một khai báo hàm hoặc hàm literal sẽ dành bộ nhớ cho một biến có tên. Gọi hàm dựng sẵn `new` hoặc lấy địa chỉ của một literal tổng hợp sẽ cấp phát bộ nhớ cho một biến tại thời gian chạy. Biến ẩn danh như vậy được tham chiếu thông qua một con trỏ gián tiếp (có thể ngầm định).

Biến *cấu trúc* của các kiểu mảng, slice, và struct có các phần tử và trường có thể được truy cập địa chỉ riêng lẻ. Mỗi phần tử như vậy hoạt động như một biến.

Kiểu tĩnh (hoặc chỉ là *kiểu*) của một biến là kiểu được khai báo, kiểu được cung cấp trong lệnh gọi `new` hoặc literal tổng hợp, hoặc kiểu của một phần tử của biến cấu trúc. Biến kiểu interface cũng có *kiểu động* riêng biệt, là kiểu (không phải interface) của giá trị được gán cho biến tại thời gian chạy (trừ khi giá trị là identifier định nghĩa sẵn `nil`, không có kiểu). Kiểu động có thể thay đổi trong quá trình thực thi nhưng các giá trị lưu trong biến interface luôn có thể gán cho kiểu tĩnh của biến.

```
var x interface{} // x là nil và có kiểu tĩnh
interface{}
var v *T // v có giá trị nil, kiểu tĩnh *T
x = 42 // x có giá trị 42 và kiểu động int
x = v // x có giá trị (*T)(nil) và kiểu động *T
```

Giá trị của biến được lấy bằng cách tham chiếu đến biến trong một biểu thức; đó là giá trị gần nhất được gán cho biến. Nếu biến chưa được gán giá trị, giá trị của nó là giá trị zero cho kiểu của nó.

Types

Kiểu xác định một tập hợp giá trị cùng với các phép toán và phương thức cụ thể cho các giá trị đó. Một kiểu có thể được biểu diễn bởi một *tên kiểu*, nếu có, và phải theo sau bởi tham số kiểu nếu kiểu đó là

generic. Một kiểu cũng có thể được chỉ định bằng *literal kiểu*, kết hợp kiểu từ các kiểu có sẵn.

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

Ngôn ngữ định nghĩa sẵn một số tên kiểu. Các tên khác được giới thiệu bằng khai báo kiểu hoặc danh sách tham số kiểu. *Kiểu tổng hợp*—mảng, struct, con trỏ, hàm, interface, slice, map, và channel—có thể được xây dựng bằng literal kiểu.

Kiểu định nghĩa sẵn, kiểu định nghĩa, và tham số kiểu được gọi là *kiểu có tên*. Một alias biểu diễn kiểu có tên nếu kiểu được khai báo trong alias là kiểu có tên.

Boolean types

Một kiểu boolean biểu diễn tập hợp các giá trị chân lý Boolean được biểu diễn bởi các hằng định nghĩa sẵn `true` và `false`. Kiểu boolean định nghĩa sẵn là `bool`; nó là một kiểu định nghĩa.

Numeric types

Một kiểu số nguyên, *số thực*, hoặc *số phức* biểu diễn tập hợp các giá trị số nguyên, số thực, hoặc số phức tương ứng. Chúng được gọi chung là *kiểu số học*. Các kiểu số học định nghĩa sẵn không phụ thuộc kiến trúc là:

uint8 tập hợp tất cả các số nguyên không dấu 8 bit (0 đến 255) uint16 tập hợp tất cả các số nguyên không dấu 16 bit (0 đến 65535) uint32 tập hợp tất cả các số nguyên không dấu 32 bit (0 đến 4294967295) uint64 tập hợp tất cả các số nguyên không dấu 64 bit (0 đến 18446744073709551615)

int8 tập hợp tất cả các số nguyên có dấu 8 bit (-128 đến 127) int16 tập hợp tất cả các số nguyên có dấu 16 bit (-32768 đến 32767) int32 tập hợp tất cả các số nguyên có dấu 32 bit (-2147483648 đến 2147483647) int64 tập hợp tất cả các số nguyên có dấu 64 bit (-9223372036854775808 đến 9223372036854775807)

float32 tập hợp tất cả các số thực dấu phẩy động 32 bit theo IEEE 754 float64 tập hợp tất cả các số thực dấu phẩy động 64 bit theo IEEE 754

complex64 tập hợp tất cả các số phức với phần thực và ảo kiểu float32 complex128 tập hợp tất cả các số phức với phần thực và ảo kiểu float64

byte là alias cho uint8 rune là alias cho int32

Giá trị của một số nguyên *n*-bit có chiều rộng *n* bit và được biểu diễn bằng số bù hai.

Cũng có một tập hợp các kiểu số nguyên định nghĩa sẵn với kích thước phụ thuộc vào triển khai:

uint hoặc 32 hoặc 64 bit int cùng kích thước với uint uintptr một số nguyên không dấu đủ lớn để lưu trữ bit chưa giải thích của giá trị con trỏ

Để tránh các vấn đề về tính di động, tất cả các kiểu số học là kiểu định nghĩa và do đó là riêng biệt, trừ `byte` là alias cho `uint8`, và `rune` là alias cho `int32`. Cần chuyển đổi tường minh khi trộn các kiểu số học khác nhau trong biểu thức hoặc phép gán. Ví dụ, `int32` và `int` không phải là cùng một kiểu dù có thể cùng kích thước trên một kiến trúc cụ thể.

String types

Một kiểu chuỗi biểu diễn tập hợp các giá trị chuỗi. Một giá trị chuỗi là một chuỗi byte (có thể rỗng). Số byte gọi là độ dài chuỗi và không bao giờ âm. Chuỗi là bất biến: một khi đã tạo, không thể thay đổi nội dung chuỗi. Kiểu chuỗi định nghĩa sẵn là `string`; nó là một kiểu định nghĩa.

Độ dài của chuỗi `s` có thể được lấy bằng hàm dựng sẵn `len`. Độ dài là hằng số tại thời gian biên dịch nếu chuỗi là hằng. Các byte của chuỗi có thể được truy cập bằng chỉ số nguyên từ 0 đến `len(s)-1`. Không hợp lệ khi lấy địa chỉ của phần tử như vậy; nếu `s[i]` là byte thứ `i` của chuỗi, thì `&s[i]` là không hợp lệ.

Array types

Một mảng là một chuỗi đánh số các phần tử cùng kiểu, gọi là kiểu phần tử. Số phần tử gọi là độ dài mảng và không bao giờ âm.

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

Độ dài là một phần của kiểu mảng; nó phải đánh giá thành một hằng số không âm có thể biểu diễn bởi giá trị kiểu `int`. Độ dài của mảng `a` có thể lấy bằng hàm dựng sẵn `len`. Các phần tử có thể được truy cập bằng chỉ số nguyên từ 0 đến `len(a)-1`. Kiểu mảng luôn một chiều nhưng có thể kết hợp để tạo kiểu đa chiều.

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

```
[2][2][2]float64 // giống như [2](<[2]
([2]float64)>)
```

Một kiểu mảng `T` không được có phần tử kiểu `T`, hoặc kiểu chứa `T` như một thành phần, trực tiếp hoặc gián tiếp, nếu các kiểu chứa đó chỉ là kiểu mảng hoặc struct.

```
// kiểu mảng không hợp lệ
type (
T1 [10]T1 // phần tử của T1 là T1
T2 [10]struct{ f T2 } // T2 chứa T2 như thành phần
của struct
T3 [10]T4 // T3 chứa T3 như thành phần của struct
trong T4
T4 struct{ f T3 } // T4 chứa T4 như thành phần của
mảng T3 trong struct
)

// kiểu mảng hợp lệ
type (
T5 [10]*T5 // T5 chứa T5 như thành phần của con trỏ
T6 [10]func() T6 // T6 chứa T6 như thành phần của
kiểu hàm
T7 [10]struct{ f []T7 } // T7 chứa T7 như thành
phần của slice trong struct
)
```

Slice types

Slice là một mô tả cho một đoạn liên tục của *mảng nền* và cung cấp truy cập đến một chuỗi đánh số các phần tử từ mảng đó. Kiểu slice biểu diễn tập hợp tất cả các slice của mảng có kiểu phần tử tương ứng. Số

phần tử gọi là độ dài slice và không bao giờ âm. Giá trị của slice chưa khởi tạo là `nil`.

SliceType = "[" "]" ElementType .

Độ dài của slice `s` có thể lấy bằng hàm dựng sẵn `len`; khác với mảng, nó có thể thay đổi trong quá trình thực thi. Các phần tử có thể được truy cập bằng chỉ số nguyên từ 0 đến `len(s)-1`. Chỉ số slice của một phần tử có thể nhỏ hơn chỉ số của phần tử đó trong mảng nền.

Một slice, khi đã khởi tạo, luôn liên kết với một mảng nền chứa các phần tử của nó. Do đó, slice chia sẻ bộ nhớ với mảng và các slice khác của cùng mảng; ngược lại, các mảng khác nhau luôn là bộ nhớ riêng biệt.

Mảng nền của slice có thể kéo dài vượt quá cuối slice. *Dung lượng* là thước đo phạm vi đó: nó là tổng độ dài slice và độ dài mảng vượt quá slice; một slice có độ dài đến dung lượng đó có thể được tạo bằng *slicing* một slice mới từ slice gốc. Dung lượng của slice `a` có thể lấy bằng hàm dựng sẵn `cap(a)`.

Một giá trị slice mới, đã khởi tạo cho kiểu phần tử `T` có thể được tạo bằng hàm dựng sẵn `make`, nhận kiểu slice và các tham số chỉ định độ dài và tùy chọn dung lượng. Slice tạo bằng `make` luôn cấp phát một mảng ẩn mới mà slice trả về tham chiếu đến. Tức là, thực hiện

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

tạo ra slice giống như cấp phát một mảng và slicing nó, nên hai biểu thức sau là tương đương:

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

Giống như mảng, slice luôn một chiều nhưng có thể kết hợp để tạo đối tượng đa chiều. Với mảng của mảng, các mảng bên trong luôn cùng độ dài; tuy nhiên với slice của slice (hoặc mảng của slice), độ dài bên trong có thể thay đổi động. Ngoài ra, các slice bên trong phải được khởi tạo riêng lẻ.

Struct types

Struct là một chuỗi các phần tử có tên, gọi là trường, mỗi trường có tên và kiểu. Tên trường có thể được chỉ định tường minh (IdentifierList) hoặc ngầm định (EmbeddedField). Trong một struct, tên trường không blank phải duy nhất.

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

// Một struct rỗng.
struct {}

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

Trường được khai báo với kiểu nhưng không có tên trường tường minh gọi là *trường nhúng*. Trường nhúng phải được chỉ định là tên kiểu `T` hoặc con trỏ đến tên kiểu không phải interface `*T`, và bản thân `T` không được là kiểu con trỏ hoặc tham số kiểu. Tên kiểu không đủ điều kiện sẽ là tên trường.

```
// Một struct với bốn trường nhúng kiểu T1, *T2,
P.T3 và *P.T4
struct {
T1 // tên trường là T1
*T2 // tên trường là T2
P.T3 // tên trường là T3
*P.T4 // tên trường là T4
x, y int // tên trường là x và y
}
```

Khai báo sau là không hợp lệ vì tên trường phải duy nhất trong một kiểu struct:

```
struct {
T // trùng với trường nhúng *T và *P.T
*T // trùng với trường nhúng T và *P.T
*P.T // trùng với trường nhúng T và *T
}
```

Một trường hoặc phương thức `f` của trường nhúng trong struct `x` được gọi là *được nâng cấp* nếu `x.f` là một selector hợp lệ biểu diễn trường hoặc phương thức đó.

Trường được nâng cấp hoạt động như trường thông thường của struct trừ việc không thể dùng làm tên trường trong literal tổng hợp của struct.

Với kiểu struct `S` và tên kiểu `T`, các phương thức được nâng cấp được đưa vào tập phương thức của struct như sau:

- Nếu `S` chứa trường nhúng `T`, tập phương thức của `S` và `*S` đều bao gồm các phương thức được nâng cấp với receiver `T`. Tập phương thức của `*S` cũng bao gồm các phương thức được nâng cấp với receiver `*T`.
- Nếu `S` chứa trường nhúng `*T`, tập phương thức của `S` và `*S` đều bao gồm các phương thức được nâng cấp với receiver `T` hoặc `*T`.

Khai báo trường có thể theo sau bởi một literal chuỗi *tag* tùy chọn, trở thành thuộc tính cho tất cả các trường trong khai báo trường tương ứng. Chuỗi tag rỗng tương đương với không có tag. Các tag được truy cập qua giao diện reflection và tham gia vào định danh kiểu cho struct nhưng bị bỏ qua ở các trường hợp khác.

```
struct {
x, y float64 "" // chuỗi tag rỗng như không có tag
name string "bất kỳ chuỗi nào cũng được phép làm
tag"
_ [4]byte "ceci n'est pas un champ de structure"
}

// Một struct tương ứng với TimeStamp protocol
buffer.
// Chuỗi tag xác định số trường protocol buffer;
// theo quy ước của gói reflect.
struct {
microsec uint64 `protobuf:"1"`
serverIP6 uint64 `protobuf:"2"`
}
```

Một kiểu struct `T` không được chứa trường kiểu `T`, hoặc kiểu chứa `T` như thành phần, trực tiếp hoặc gián tiếp, nếu các kiểu chứa đó chỉ là kiểu mảng hoặc struct.

```
// kiểu struct không hợp lệ
type (
T1 struct{ T1 } // T1 chứa trường kiểu T1
T2 struct{ f [10]T2 } // T2 chứa T2 như thành phần
của mảng
T3 struct{ T4 } // T3 chứa T3 như thành phần của
mảng trong struct T4
T4 struct{ f [10]T3 } // T4 chứa T4 như thành phần
của struct T3 trong mảng
)

// kiểu struct hợp lệ
type (
T5 struct{ f *T5 } // T5 chứa T5 như thành phần của
con trỏ
T6 struct{ f func() T6 } // T6 chứa T6 như thành
phần của kiểu hàm
T7 struct{ f [10][]T7 } // T7 chứa T7 như thành
phần của slice trong mảng
)
```

Pointer types

Kiểu con trỏ biểu diễn tập hợp tất cả các con trỏ đến biến của một kiểu nhất định, gọi là *kiểu cơ sở* của con trỏ. Giá trị của con trỏ chưa khởi tạo là `nil`.

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

*Point
*[4]int
```

Function types

Kiểu hàm biểu diễn tập hợp tất cả các hàm có cùng kiểu tham số và kết quả. Giá trị của biến kiểu hàm chưa khởi tạo là `nil`.

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

Trong danh sách tham số hoặc kết quả, tên (IdentifierList) hoặc phải có đầy đủ hoặc không có. Nếu có, mỗi tên đại diện cho một phần tử (tham số hoặc kết quả) của kiểu chỉ định và tất cả tên không blank trong chữ ký phải duy nhất. Nếu không có, mỗi kiểu đại diện cho một phần tử của kiểu đó. Danh sách tham số và kết quả luôn đặt trong dấu ngoặc tròn trừ khi chỉ có một kết quả không tên thì có thể viết dưới dạng kiểu không ngoặc.

Tham số cuối cùng trong chữ ký hàm có thể có kiểu bắt đầu bằng `...`. Hàm có tham số như vậy gọi là *variadic* và có thể được gọi với không hoặc nhiều đối số cho tham số đó.

```
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

Kiểu interface định nghĩa một *tập hợp kiểu*. Biến kiểu interface có thể lưu giá trị của bất kỳ kiểu nào thuộc tập hợp kiểu của interface. Kiểu như vậy được gọi là triển khai interface. Giá trị của biến kiểu interface chưa khởi tạo là `nil`.

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

Kiểu interface được chỉ định bằng danh sách *phần tử interface*. Một phần tử interface là *phương thức* hoặc *phần tử kiểu*, trong đó phần tử kiểu là hợp của một hoặc nhiều *type term*. Type term là một kiểu đơn hoặc kiểu nền đơn.

Basic interfaces

Ở dạng cơ bản nhất, interface chỉ định một danh sách (có thể rỗng) các phương thức. Tập hợp kiểu do interface như vậy định nghĩa là tập hợp các kiểu triển khai tất cả các phương thức đó, và tập phương thức tương ứng chỉ gồm các phương thức được chỉ định bởi interface. Interface mà tập hợp kiểu chỉ định hoàn toàn bằng danh sách phương thức gọi là *basic interfaces*.

```
// Một interface File đơn giản.
interface {
Read([]byte) (int, error)
Write([]byte) (int, error)
Close() error
}
```

Tên của mỗi phương thức chỉ định tường minh phải duy nhất và không blank.

```
interface {
String() string
String() string // không hợp lệ: String không duy
nhất
_(x int) // không hợp lệ: phương thức phải có tên
không blank
}
```

Nhiều kiểu có thể triển khai một interface. Ví dụ, nếu hai kiểu `S1` và `S2` có tập phương thức

```
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
```

(trong đó `T` là `S1` hoặc `S2`) thì interface `File` được triển khai bởi cả `S1` và `S2`, bất kể các phương thức khác mà `S1` và `S2` có hoặc chia sẻ.

Mọi kiểu là thành viên của tập hợp kiểu của một interface đều triển khai interface đó. Một kiểu có thể triển khai nhiều interface khác nhau. Ví dụ, mọi kiểu đều triển khai *interface rỗng* đại diện cho tập hợp tất cả các kiểu (không phải interface):

```
interface{}
```

Để thuận tiện, kiểu định nghĩa sẵn `any` là alias cho interface rỗng. [Go 1.18]

Tương tự, xem xét khai báo interface sau, xuất hiện trong khai báo kiểu để định nghĩa interface tên `Locker`:

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

Nếu `S1` và `S2` cũng triển khai

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

chúng cũng triển khai interface `Locker` cũng như interface `File`.

Embedded interfaces

Ở dạng tổng quát hơn, interface `T` có thể sử dụng tên kiểu interface (có thể đủ điều kiện) `E` làm phần tử interface. Đây gọi là *nhúng* interface `E` vào `T` [Go 1.14]. Tập hợp kiểu của `T` là *giao* của các tập hợp kiểu do các phương thức khai báo tường minh của `T` và các tập hợp kiểu của các interface nhúng của `T` xác định. Nói cách khác, tập hợp kiểu của `T` là tập hợp tất cả các kiểu triển khai tất cả các phương thức khai báo tường minh của `T` và tất cả các phương thức của `E` [Go 1.18].

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

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

// Các phương thức của ReadWriter là Read, Write,
và Close.
type ReadWriter interface {
Reader // bao gồm các phương thức của Reader trong
tập phương thức của ReadWriter
Writer // bao gồm các phương thức của Writer trong
tập phương thức của ReadWriter
}
```

Khi nhúng interface, các phương thức có cùng tên phải có chữ ký giống hệt.

```
type ReadCloser interface {
Reader // bao gồm các phương thức của Reader trong
tập phương thức của ReadCloser
Close() // không hợp lệ: chữ ký của Reader.Close và
Close khác nhau
}
```

General interfaces

Ở dạng tổng quát nhất, phần tử interface cũng có thể là một type term tùy ý `T`, hoặc dạng `~T` chỉ định kiểu nền `T`, hoặc hợp của các term `t1|t2|…|tn` [Go 1.18]. Cùng với các phương thức, các phần tử này cho phép định nghĩa chính xác tập hợp kiểu của interface như sau:

- Tập hợp kiểu của interface rỗng là tập hợp tất cả các kiểu không phải interface.
- Tập hợp kiểu của interface không rỗng là giao của các tập hợp kiểu của các phần tử interface.
- Tập hợp kiểu của một phương thức là tập hợp tất cả các kiểu không phải interface có tập phương thức bao gồm phương thức đó.
- Tập hợp kiểu của một type term không phải interface là tập hợp chỉ gồm kiểu đó.
- Tập hợp kiểu của term dạng `~T` là tập hợp tất cả các kiểu có kiểu nền là `T`.
- Tập hợp kiểu của *hợp* các term `t1|t2|…|tn` là hợp của các tập hợp kiểu của các term.

Việc định lượng "tập hợp tất cả các kiểu không phải interface" không chỉ bao gồm tất cả các kiểu (không phải interface) được khai báo trong chương trình hiện tại, mà còn tất cả các kiểu có thể có trong mọi chương trình, do đó là vô hạn. Tương tự, với tập hợp tất cả các kiểu

không phải interface triển khai một phương thức cụ thể, giao của các tập phương thức của các kiểu đó sẽ chỉ chứa phương thức đó, ngay cả khi tất cả các kiểu trong chương trình hiện tại luôn ghép phương thức đó với phương thức khác.

Theo cấu trúc, tập hợp kiểu của một interface không bao giờ chứa kiểu interface.

```
// Một interface chỉ đại diện cho kiểu int.
interface {
int
}

// Một interface đại diện cho tất cả các kiểu có
kiểu nền là int.
interface {
~int
}

// Một interface đại diện cho tất cả các kiểu có
kiểu nền là int và triển khai phương thức String.
interface {
~int
String() string
}

// Một interface đại diện cho tập hợp kiểu rỗng:
không có kiểu nào vừa là int vừa là string.
interface {
int
string
}
```

Trong term dạng `~T`, kiểu nền của `T` phải là chính nó, và `T` không được là interface.

```
type MyInt int

interface {
~[]byte // kiểu nền của []byte là chính nó
~MyInt // không hợp lệ: kiểu nền của MyInt không
phải là MyInt
~error // không hợp lệ: error là interface
}
```

Các phần tử hợp biểu diễn hợp của các tập hợp kiểu:

```
// Interface Float đại diện cho tất cả các kiểu số
thực
// (bao gồm cả các kiểu có tên mà kiểu nền là
// float32 hoặc float64).
type Float interface {
~float32 | ~float64
}
```

Kiểu `T` trong term dạng `T` hoặc `~T` không được là tham số kiểu, và tập hợp kiểu của tất cả các term không phải interface phải đôi một rời nhau (giao của các tập hợp kiểu phải rỗng). Với tham số kiểu `P`:

```
interface {
P // không hợp lệ: P là tham số kiểu
int | ~P // không hợp lệ: P là tham số kiểu
~int | MyInt // không hợp lệ: tập hợp kiểu của ~int
và MyInt không rời nhau (~int bao gồm MyInt)
```

```
float32 | Float // tập hợp kiểu chồng lấp nhưng
Float là interface
}
```

Giới hạn triển khai: Một hợp (có nhiều hơn một term) không được chứa identifier định nghĩa sẵn `comparable` hoặc interface chỉ định phương thức, hoặc nhúng `comparable` hoặc interface chỉ định phương thức.

Interface không phải basic chỉ được dùng làm ràng buộc kiểu, hoặc phần tử của interface khác dùng làm ràng buộc. Chúng không thể là kiểu của giá trị hoặc biến, hoặc thành phần của kiểu khác không phải interface.

```
var x Float // không hợp lệ: Float không phải basic
interface

var x interface{} = Float(nil) // không hợp lệ

type Floatish struct {
f Float // không hợp lệ
}

Một kiểu interface `T` không được nhúng phần tử
kiểu là, chứa, hoặc nhúng `T`, trực tiếp hoặc gián
tiếp.

// không hợp lệ: Bad không được nhúng chính nó
type Bad interface {
Bad
}

// không hợp lệ: Bad1 không được nhúng chính nó qua
Bad2
type Bad1 interface {
Bad2
```

```
}
type Bad2 interface {
Bad1
}

// không hợp lệ: Bad3 không được nhúng hợp chứa
Bad3
type Bad3 interface {
~int | ~string | Bad3
}

// không hợp lệ: Bad4 không được nhúng mảng chứa
Bad4 làm kiểu phần tử
type Bad4 interface {
[10]Bad4
}
```

Implementing an interface

Một kiểu `T` triển khai interface `I` nếu

- `T` không phải interface và là phần tử của tập hợp kiểu của `I`; hoặc
- `T` là interface và tập hợp kiểu của `T` là tập con của tập hợp kiểu của `I`.

Giá trị kiểu `T` triển khai interface nếu `T` triển khai interface đó.

Map types

Map là một nhóm không có thứ tự các phần tử của một kiểu, gọi là kiểu phần tử, được đánh chỉ số bởi một tập hợp *khóa* duy nhất của kiểu khác, gọi là kiểu khóa. Giá trị của map chưa khởi tạo là `nil`.

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

Toán tử so sánh `==` và `!=` phải được định nghĩa đầy đủ cho toán hạng kiểu khóa; do đó kiểu khóa không được là hàm, map, hoặc slice. Nếu kiểu khóa là kiểu interface, các toán tử so sánh này phải được định nghĩa cho giá trị khóa động; nếu không sẽ gây panic tại runtime.

```
map[string]int
map[*T]struct{ x, y float64 }
map[string]interface{}
```

Số phần tử của map gọi là độ dài. Với map `m`, có thể lấy bằng hàm dựng sẵn `len` và có thể thay đổi trong quá trình thực thi. Phần tử có thể được thêm trong quá trình thực thi bằng gán và truy xuất bằng biểu thức chỉ số; có thể xóa bằng hàm dựng sẵn `delete` và `clear`.

Một giá trị map mới, rỗng được tạo bằng hàm dựng sẵn `make`, nhận kiểu map và tùy chọn gợi ý dung lượng làm đối số:

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

Dung lượng ban đầu không giới hạn kích thước: map sẽ tự động mở rộng để chứa số phần tử lưu trữ, trừ map `nil`. Map `nil` tương đương map rỗng trừ việc không thể thêm phần tử.

Channel types

Channel cung cấp cơ chế cho các hàm thực thi đồng thời giao tiếp bằng cách gửi và nhận giá trị của kiểu phần tử chỉ định. Giá trị của channel chưa khởi tạo là `nil`.

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

Toán tử `<-` tùy chọn chỉ định *hướng* channel, *gửi* hoặc *nhận*. Nếu có hướng, channel là *một chiều*, nếu không là *hai chiều*. Channel có thể bị giới hạn chỉ gửi hoặc chỉ nhận bằng gán hoặc chuyển đổi tường minh.

```
chan T // có thể dùng để gửi và nhận giá trị kiểu T
chan<- float64 // chỉ dùng để gửi float64
<-chan int // chỉ dùng để nhận int
```

Toán tử `<-` liên kết với `chan` ngoài cùng bên trái có thể:

```
chan<- chan int // giống như chan<- (chan int)
chan<- <-chan int // giống như chan<- (<-chan int)
<-chan <-chan int // giống như <-chan (<-chan int)
chan (<-chan int)
```

Một giá trị channel mới, đã khởi tạo có thể được tạo bằng hàm dựng sẵn `make`, nhận kiểu channel và tùy chọn *dung lượng* làm đối số:

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

Dung lượng, tính theo số phần tử, xác định kích thước bộ đệm của channel. Nếu dung lượng là 0 hoặc không có, channel là không bộ đệm và giao tiếp chỉ thành công khi cả bên gửi và nhận đều sẵn sàng. Nếu không, channel có bộ đệm và giao tiếp thành công mà không bị chặn

nếu bộ đệm chưa đầy (gửi) hoặc chưa rỗng (nhận). Channel `nil` không bao giờ sẵn sàng giao tiếp.

Channel có thể được đóng bằng hàm dựng sẵn `close`. Dạng gán nhiều giá trị của toán tử nhận báo cáo liệu giá trị nhận được có được gửi trước khi channel bị đóng hay không.

Một channel có thể được sử dụng trong câu lệnh gửi, phép nhận, và gọi các hàm dựng sẵn `cap` và `len` bởi bất kỳ số goroutine nào mà không cần đồng bộ hóa thêm. Channel hoạt động như hàng đợi FIFO. Ví dụ, nếu một goroutine gửi giá trị lên channel và một goroutine khác nhận, các giá trị sẽ được nhận theo thứ tự gửi.

Properties of types and values

Representation of values

Giá trị của các kiểu định nghĩa sẵn (xem bên dưới với interface `any` và `error`), mảng, và struct là tự chứa: Mỗi giá trị như vậy chứa bản sao đầy đủ của tất cả dữ liệu, và biến của các kiểu này lưu trữ toàn bộ giá trị. Ví dụ, biến mảng cung cấp bộ nhớ (các biến) cho tất cả phần tử của mảng. Giá trị zero tương ứng với từng kiểu; chúng không bao giờ là `nil`.

Giá trị con trỏ, hàm, slice, map, và channel khác nil chứa tham chiếu đến dữ liệu nền có thể được chia sẻ bởi nhiều giá trị:

- Giá trị con trỏ là tham chiếu đến biến chứa giá trị kiểu cơ sở con trỏ.
- Giá trị hàm chứa tham chiếu đến hàm (có thể là ẩn danh) và các biến bao quanh.
- Giá trị slice chứa độ dài, dung lượng, và tham chiếu đến mảng nền.
- Giá trị map hoặc channel là tham chiếu đến cấu trúc dữ liệu đặc trưng của map hoặc channel.

Giá trị interface có thể tự chứa hoặc chứa tham chiếu đến dữ liệu nền tùy thuộc vào kiểu động của interface. Identifier định nghĩa sẵn `nil` là giá trị zero cho các kiểu có thể chứa tham chiếu.

Khi nhiều giá trị chia sẻ dữ liệu nền, thay đổi một giá trị có thể làm thay đổi giá trị khác. Ví dụ, thay đổi một phần tử của slice sẽ thay đổi phần tử đó trong mảng nền cho tất cả các slice chia sẻ mảng đó.

Underlying types

Mỗi kiểu `T` có một *kiểu nền*: Nếu `T` là một trong các kiểu boolean, số học, hoặc chuỗi định nghĩa sẵn, hoặc một literal kiểu, thì kiểu nền tương ứng là chính `T`. Nếu không, kiểu nền của `T` là kiểu nền của kiểu mà `T` tham chiếu trong khai báo. Với tham số kiểu, kiểu nền là ràng buộc kiểu, luôn là interface.

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

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

func f[P any](x P) { … }
```

Kiểu nền của `string`, `A1`, `A2`, `B1`, và `B2` là `string`. Kiểu nền của `[]B1`, `B3`, và `B4` là `[]B1`. Kiểu nền của `P` là `interface{}`.

Type identity

Hai kiểu hoặc là *giống hệt* ("giống nhau") hoặc *khác nhau*.

Một kiểu có tên luôn khác với bất kỳ kiểu nào khác. Nếu không, hai kiểu giống hệt nếu literal kiểu nền của chúng tương đương về cấu trúc; tức là, chúng có cấu trúc literal giống nhau và các thành phần tương ứng có kiểu giống hệt. Cụ thể:

- Hai kiểu mảng giống hệt nếu có kiểu phần tử giống hệt và cùng độ dài mảng.
- Hai kiểu slice giống hệt nếu có kiểu phần tử giống hệt.
- Hai kiểu struct giống hệt nếu có cùng chuỗi trường, và các cặp trường tương ứng có cùng tên, kiểu giống hệt, tag giống hệt, và đều là trường nhúng hoặc đều không nhúng. Tên trường không export từ các package khác nhau luôn khác nhau.
- Hai kiểu con trỏ giống hệt nếu có kiểu cơ sở giống hệt.
- Hai kiểu hàm giống hệt nếu có cùng số tham số và giá trị trả về, các kiểu tham số và kết quả tương ứng giống hệt, và hoặc cả hai hàm là variadic hoặc không. Tên tham số và kết quả không cần giống nhau.
- Hai kiểu interface giống hệt nếu định nghĩa cùng tập hợp kiểu.
- Hai kiểu map giống hệt nếu có kiểu khóa và phần tử giống hệt.
- Hai kiểu channel giống hệt nếu có kiểu phần tử và hướng giống hệt.
- Hai kiểu khởi tạo giống hệt nếu kiểu định nghĩa và tất cả tham số kiểu giống hệt.

Với các khai báo

```
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]

)
```

các kiểu sau là giống hệt:

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

B0 và C0
D0[int, string] và E0
[]int và []int
struct{ a, b *B5 } và struct{ a, b *B5 }
func(x int, y float64) *[]string, func(int,
float64) (result *[]string), và A5
```

`B0` và `B1` là khác nhau vì chúng là kiểu mới tạo bởi định nghĩa kiểu khác nhau; `func(int, float64) *B0` và `func(x int, y float64) *[]string` là khác nhau vì `B0` khác với `[]string`; và `P1` và `P2` là khác nhau vì chúng là tham số kiểu khác nhau. `D0[int, string]` và

`struct{ x int; y string }` là khác nhau vì cái trước là kiểu định nghĩa khởi tạo còn cái sau là literal kiểu (nhưng vẫn có thể gán).

Assignability

Giá trị `x` kiểu `V` *có thể gán* cho biến kiểu `T` ("`x` có thể gán cho `T`") nếu một trong các điều kiện sau đúng:

- `V` và `T` giống hệt.
- `V` và `T` có kiểu nền giống hệt nhưng không phải tham số kiểu và ít nhất một trong hai không phải kiểu có tên.
- `V` và `T` là kiểu channel với kiểu phần tử giống hệt, `V` là channel hai chiều, và ít nhất một trong hai không phải kiểu có tên.
- `T` là kiểu interface, không phải tham số kiểu, và `x` triển khai `T`.
- `x` là identifier định nghĩa sẵn `nil` và `T` là con trỏ, hàm, slice, map, channel, hoặc interface, nhưng không phải tham số kiểu.
- `x` là hằng không kiểu có thể biểu diễn bởi giá trị kiểu `T`.

Ngoài ra, nếu kiểu `V` của `x` hoặc `T` là tham số kiểu, `x` có thể gán cho biến kiểu `T` nếu một trong các điều kiện sau đúng:

- `x` là identifier định nghĩa sẵn `nil`, `T` là tham số kiểu, và `x` có thể gán cho mỗi kiểu trong tập hợp kiểu của `T`.
- `V` không phải kiểu có tên, `T` là tham số kiểu, và `x` có thể gán cho mỗi kiểu trong tập hợp kiểu của `T`.
- `V` là tham số kiểu và `T` không phải kiểu có tên, và giá trị của mỗi kiểu trong tập hợp kiểu của `V` có thể gán cho `T`.

Representability

Một hằng `x` *có thể biểu diễn* bởi giá trị kiểu `T`, với `T` không phải tham số kiểu, nếu một trong các điều kiện sau đúng:

- `x` thuộc tập hợp giá trị xác định bởi `T`.

- `T` là kiểu số thực và `x` có thể làm tròn đến độ chính xác của `T` mà không bị tràn. Làm tròn theo quy tắc IEEE 754 round-to-even nhưng với số không âm IEEE được đơn giản hóa thành số không không dấu. Lưu ý rằng giá trị hằng không bao giờ cho ra số không âm, NaN, hoặc vô cực IEEE.
- `T` là kiểu phức, và thành phần `real(x)` và `imag(x)` của `x` có thể biểu diễn bởi giá trị kiểu thành phần của `T` (`float32` hoặc `float64`).

Nếu `T` là tham số kiểu, `x` có thể biểu diễn bởi giá trị kiểu `T` nếu `x` có thể biểu diễn bởi giá trị của mỗi kiểu trong tập hợp kiểu của `T`.

x T x có thể biểu diễn bởi giá trị kiểu T vì

```
'a' byte 97 thuộc tập hợp giá trị byte
97 rune rune là alias cho int32, và 97 thuộc tập
hợp số nguyên 32 bit
"foo" string "foo" thuộc tập hợp giá trị chuỗi
1024 int16 1024 thuộc tập hợp số nguyên 16 bit
42.0 byte 42 thuộc tập hợp số nguyên không dấu 8
bit
1e10 uint64 10000000000 thuộc tập hợp số nguyên
không dấu 64 bit
2.718281828459045 float32 2.718281828459045 làm
tròn thành 2.7182817 thuộc tập hợp giá trị float32
-1e-1000 float64 -1e-1000 làm tròn thành IEEE -0.0
được đơn giản hóa thành 0.0
0i int 0 là giá trị số nguyên
(42 + 0i) float32 42.0 (phần ảo bằng 0) thuộc tập
hợp giá trị float32
```

x T x không thể biểu diễn bởi giá trị kiểu T vì

```
0 bool 0 không thuộc tập hợp giá trị boolean
'a' string 'a' là rune, không thuộc tập hợp giá trị
chuỗi
1024 byte 1024 không thuộc tập hợp số nguyên không
dấu 8 bit
-1 uint16 -1 không thuộc tập hợp số nguyên không
dấu 16 bit
1.1 int 1.1 không phải giá trị số nguyên
42i float32 (0 + 42i) không thuộc tập hợp giá trị
float32
1e1000 float64 1e1000 làm tròn thành IEEE +Inf
(tràn)
```

Method sets

Tập phương thức của một kiểu xác định các phương thức có thể được gọi trên một toán hạng của kiểu đó. Mỗi kiểu có một tập phương thức (có thể rỗng) liên kết với nó:

- Tập phương thức của kiểu định nghĩa `T` gồm tất cả phương thức khai báo với receiver kiểu `T`.
- Tập phương thức của con trỏ đến kiểu định nghĩa `T` (với `T` không phải con trỏ hoặc interface) là tập hợp tất cả phương thức khai báo với receiver `*T` hoặc `T`.
- Tập phương thức của kiểu interface là giao của các tập phương thức của mỗi kiểu trong tập hợp kiểu của interface (tập phương thức kết quả thường chỉ là tập các phương thức khai báo trong interface).

Các quy tắc bổ sung áp dụng cho struct (và con trỏ đến struct) chứa trường nhúng, như mô tả trong phần kiểu struct. Các kiểu khác có tập phương thức rỗng.

Trong một tập phương thức, mỗi phương thức phải có tên phương thức duy nhất và không blank.

Blocks

Một block là một chuỗi (có thể rỗng) các khai báo và câu lệnh nằm trong cặp dấu ngoặc nhọn.

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

Ngoài các block tường minh trong mã nguồn, còn có các block ngầm định:

1. *Universe block* bao gồm toàn bộ mã nguồn Go.
2. Mỗi package có một *package block* chứa toàn bộ mã nguồn của package đó.
3. Mỗi file có một *file block* chứa toàn bộ mã nguồn trong file đó.
4. Mỗi câu lệnh "if", "for", và "switch" được coi là nằm trong block ngầm định riêng.
5. Mỗi clause trong câu lệnh "switch" hoặc "select" hoạt động như một block ngầm định.

Các block lồng nhau và ảnh hưởng đến phạm vi.

Declarations and scope

Một khai báo liên kết một identifier không blank với một hằng số, kiểu, tham số kiểu, biến, hàm, nhãn, hoặc package. Mỗi identifier trong chương trình phải được khai báo. Không được khai báo hai lần cùng một identifier trong cùng một block, và không được khai báo cùng một identifier ở cả file block và package block.

Blank identifier có thể được sử dụng như bất kỳ identifier nào khác trong khai báo, nhưng nó không tạo ra liên kết và do đó không được khai báo. Trong package block, identifier `init` chỉ được dùng cho khai báo hàm `init`, và giống như blank identifier, nó không tạo ra liên kết mới.

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

Phạm vi của một identifier được khai báo là phạm vi mã nguồn mà trong đó identifier đó biểu diễn hằng số, kiểu, biến, hàm, nhãn, hoặc package đã chỉ định.

Go sử dụng phạm vi từ vựng dựa trên block:

1. Phạm vi của identifier định nghĩa sẵn là universe block.
2. Phạm vi của identifier biểu diễn hằng số, kiểu, biến, hoặc hàm (không phải method) khai báo ở top level (ngoài mọi hàm) là package block.
3. Phạm vi của tên package của một package import là file block của file chứa khai báo import đó.
4. Phạm vi của identifier biểu diễn method receiver, tham số hàm, hoặc biến kết quả là thân hàm.
5. Phạm vi của identifier biểu diễn tham số kiểu của hàm hoặc khai báo bởi method receiver bắt đầu sau tên hàm và kết thúc ở cuối thân hàm.
6. Phạm vi của identifier biểu diễn tham số kiểu của kiểu bắt đầu sau tên kiểu và kết thúc ở cuối TypeSpec.
7. Phạm vi của identifier hằng số hoặc biến khai báo trong hàm bắt đầu ở cuối ConstSpec hoặc VarSpec (ShortVarDecl với khai báo biến ngắn) và kết thúc ở cuối block chứa gần nhất.

8. Phạm vi của identifier kiểu khai báo trong hàm bắt đầu tại identifier trong TypeSpec và kết thúc ở cuối block chứa gần nhất.

Identifier khai báo trong một block có thể được khai báo lại trong block con. Khi identifier của khai báo bên trong còn trong phạm vi, nó biểu diễn thực thể được khai báo bởi khai báo bên trong.

Package clause không phải là khai báo; tên package không xuất hiện trong bất kỳ phạm vi nào. Mục đích của nó là xác định các file thuộc cùng một package và chỉ định tên package mặc định cho các khai báo import.

Label scopes

Nhãn được khai báo bởi labeled statements và được sử dụng trong các câu lệnh "break", "continue", và "goto". Việc định nghĩa một nhãn mà không bao giờ được sử dụng là không hợp lệ. Khác với các identifier khác, nhãn không bị giới hạn bởi block và không xung đột với các identifier không phải nhãn. Phạm vi của nhãn là thân hàm mà nó được khai báo và không bao gồm thân của bất kỳ hàm lồng nhau nào.

Blank identifier

Blank identifier được biểu diễn bởi ký tự gạch dưới _. Nó đóng vai trò như một placeholder ẩn danh thay cho identifier thông thường (không blank) và có ý nghĩa đặc biệt trong khai báo, như một toán hạng, và trong câu lệnh gán.

Predeclared identifiers

Các identifier sau được khai báo ngầm định trong universe block [Go 1.18] [Go 1.21]:

```
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

Một identifier có thể được *export* để cho phép truy cập từ package khác. Một identifier được export nếu cả hai điều kiện sau đúng:

1. ký tự đầu tiên của tên identifier là chữ cái hoa Unicode (Unicode category Lu); và
2. identifier được khai báo trong package block hoặc là tên trường hoặc tên method.

Tất cả các identifier khác không được export.

Uniqueness of identifiers

Với một tập hợp identifier, một identifier được gọi là *duy nhất* nếu nó *khác* với mọi identifier khác trong tập. Hai identifier là khác nhau nếu

chúng được viết khác nhau, hoặc nếu chúng xuất hiện ở các package khác nhau và không được export. Nếu không, chúng là giống nhau.

Constant declarations

Khai báo hằng số liên kết một danh sách identifier (tên các hằng số) với giá trị của một danh sách biểu thức hằng. Số lượng identifier phải bằng số lượng biểu thức, và identifier thứ *n* bên trái được liên kết với giá trị của biểu thức thứ *n* bên phải.

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

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

Nếu có kiểu, tất cả các hằng số nhận kiểu chỉ định, và các biểu thức phải có thể gán cho kiểu đó, kiểu này không được là tham số kiểu. Nếu không có kiểu, các hằng số nhận kiểu riêng của từng biểu thức tương ứng. Nếu giá trị biểu thức là hằng không kiểu, các hằng số khai báo vẫn không kiểu và identifier hằng số biểu diễn giá trị hằng số. Ví dụ, nếu biểu thức là literal số thực, identifier hằng số biểu diễn hằng số số thực, ngay cả khi phần thập phân của literal là 0.

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0         // hằng số thực không kiểu
const (
    size int64 = 1024
    eof        = -1  // hằng số nguyên không kiểu
)
```

```
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c =
"foo", hằng số nguyên và chuỗi không kiểu
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

Trong một danh sách khai báo `const` có dấu ngoặc, danh sách biểu thức có thể bị bỏ qua ở bất kỳ ConstSpec nào trừ ConstSpec đầu tiên. Danh sách rỗng như vậy tương đương với việc thay thế văn bản bằng danh sách biểu thức không rỗng gần nhất trước đó và kiểu của nó nếu có. Việc bỏ qua danh sách biểu thức tương đương với việc lặp lại danh sách trước đó. Số lượng identifier phải bằng số lượng biểu thức trong danh sách trước đó. Kết hợp với bộ sinh hằng số `iota`, cơ chế này cho phép khai báo nhanh các giá trị tuần tự:

```
const (
    Sunday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Partyday
    numberOfDays  // hằng số này không được export
)
```

Iota

Trong khai báo hằng số, identifier định nghĩa sẵn `iota` biểu diễn các hằng số nguyên không kiểu liên tiếp. Giá trị của nó là chỉ số của ConstSpec tương ứng trong khai báo hằng số, bắt đầu từ 0. Nó có thể được dùng để tạo một tập hợp các hằng số liên quan:

```
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, không
dùng)
    d = 1 << iota  // d == 8  (iota == 3)
)

const (
    u         = iota * 42  // u == 0     (hằng số
nguyên không kiểu)
    v float64 = iota * 42  // v == 42.0  (hằng số
float64)
    w         = iota * 42  // w == 84    (hằng số
nguyên không kiểu)
)

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

Theo định nghĩa, nhiều lần sử dụng `iota` trong cùng một ConstSpec đều có giá trị giống nhau:

```
const (
    bit0, mask0 = 1 << iota, 1<<iota - 1  // bit0
== 1, mask0 == 0  (iota == 0)
    bit1, mask1                           // bit1
== 2, mask1 == 1  (iota == 1)
```

```
    _, _                                    //
(iota == 2, không dùng)
    bit3, mask3                             // bit3
== 8, mask3 == 7  (iota == 3)
)
```

Ví dụ cuối cùng này tận dụng lặp lại ngầm định của danh sách biểu thức không rỗng cuối cùng.

Type declarations

Khai báo kiểu liên kết một identifier, *tên kiểu*, với một kiểu. Khai báo kiểu có hai dạng: khai báo alias và định nghĩa kiểu.

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

Alias declarations

Khai báo alias liên kết một identifier với kiểu chỉ định [Go 1.9].

AliasDecl = identifier [TypeParameters] "=" Type .

Trong phạm vi của identifier, nó đóng vai trò là *alias* cho kiểu chỉ định.

```
type (
    nodeList = []*Node  // nodeList và []*Node là
cùng một kiểu
    Polar    = polar    // Polar và polar là cùng
một kiểu
)
```

Nếu khai báo alias chỉ định tham số kiểu [Go 1.24], tên kiểu biểu diễn một *alias generic*. Alias generic phải được khởi tạo khi sử dụng.

```
type set[P comparable] = map[P]bool

In an alias declaration the given type cannot be a
type parameter.

type A[P any] = P    // không hợp lệ: P là tham số
kiểu
```

Type definitions

Định nghĩa kiểu tạo ra một kiểu mới, riêng biệt với cùng kiểu nền và các phép toán như kiểu chỉ định và liên kết một identifier, *tên kiểu*, với nó.

TypeDef = identifier [TypeParameters] Type .

Kiểu mới được gọi là *kiểu định nghĩa*. Nó khác với bất kỳ kiểu nào khác, kể cả kiểu mà nó được tạo ra từ đó.

```
type (
    Point struct{ x, y float64 }  // Point và
struct{ x, y float64 } là hai kiểu khác nhau
    polar Point                   // polar và Point
là hai kiểu khác nhau
)

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

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

Kiểu định nghĩa có thể có method liên kết với nó. Nó không kế thừa bất kỳ method nào của kiểu chỉ định, nhưng tập method của kiểu interface hoặc phần tử của kiểu tổng hợp vẫn không đổi:

```
// Mutex là kiểu dữ liệu với hai method, Lock và
Unlock.
type Mutex struct         { /* Mutex fields */ }
func (m *Mutex) Lock()    { /* Lock implementation
*/ }
func (m *Mutex) Unlock()  { /* Unlock
implementation */ }

// NewMutex có cùng cấu trúc với Mutex nhưng tập
method rỗng.
type NewMutex Mutex

// Tập method của kiểu nền PtrMutex là *Mutex không
đổi,
// nhưng tập method của PtrMutex là rỗng.
type PtrMutex *Mutex

// Tập method của *PrintableMutex chứa các method
// Lock và Unlock liên kết với trường nhúng Mutex.
type PrintableMutex struct {
    Mutex
}

// MyBlock là kiểu interface có cùng tập method với
```

```
Block.
type MyBlock Block
```

Định nghĩa kiểu có thể dùng để định nghĩa các kiểu boolean, số học, hoặc chuỗi khác nhau và liên kết method với chúng:

```
type TimeZone int

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

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

Nếu định nghĩa kiểu chỉ định tham số kiểu, tên kiểu biểu diễn một *kiểu generic*. Kiểu generic phải được khởi tạo khi sử dụng.

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

Trong định nghĩa kiểu, kiểu chỉ định không được là tham số kiểu.

```
type T[P any] P    // không hợp lệ: P là tham số
kiểu
```

```
func f[T any]() {
    type L T   // không hợp lệ: T là tham số kiểu
khai báo bởi hàm bao ngoài
}
```

Kiểu generic cũng có thể có method liên kết với nó. Trong trường hợp này, receiver của method phải khai báo cùng số lượng tham số kiểu như trong định nghĩa kiểu generic.

```
// Method Len trả về số phần tử trong linked list
l.
func (l *List[T]) Len() int  { … }
```

Type parameter declarations

Danh sách tham số kiểu khai báo *tham số kiểu* của hàm generic hoặc khai báo kiểu generic. Danh sách tham số kiểu giống như danh sách tham số hàm thông thường ngoại trừ việc tên tham số kiểu phải có đầy đủ và danh sách được đặt trong dấu ngoặc vuông thay vì ngoặc tròn [Go 1.18].

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

Tất cả tên không blank trong danh sách phải duy nhất. Mỗi tên khai báo một tham số kiểu, là một kiểu có tên mới và khác, đóng vai trò

placeholder cho một kiểu chưa biết trong khai báo. Tham số kiểu được thay thế bằng *đối số kiểu* khi khởi tạo hàm hoặc kiểu generic.

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

Cũng như mỗi tham số hàm thông thường có kiểu tham số, mỗi tham số kiểu có (meta-)kiểu tương ứng gọi là *ràng buộc kiểu*.

Có thể xảy ra mơ hồ khi danh sách tham số kiểu cho kiểu generic khai báo một tham số kiểu `P` với ràng buộc `C` sao cho văn bản `P C` tạo thành một biểu thức hợp lệ:

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

Trong các trường hợp hiếm này, danh sách tham số kiểu không phân biệt được với biểu thức và khai báo kiểu được phân tích như khai báo kiểu mảng. Để giải quyết, hãy nhúng ràng buộc vào interface hoặc dùng dấu phẩy cuối:

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

Tham số kiểu cũng có thể được khai báo bởi receiver của method liên kết với kiểu generic.

Trong danh sách tham số kiểu của kiểu generic `T`, ràng buộc kiểu không được (trực tiếp hoặc gián tiếp qua danh sách tham số kiểu của kiểu generic khác) tham chiếu đến `T`.

```
type T1[P T1[P]] …                    // không hợp lệ: T1 tham chiếu chính nó
type T2[P interface{ T2[int] }] …     // không hợp lệ: T2 tham chiếu chính nó
type T3[P interface{ m(T3[int])}] …   // không hợp lệ: T3 tham chiếu chính nó
type T4[P T5[P]] …                    // không hợp lệ: T4 tham chiếu T5 và
type T5[P T4[P]] …                    //
T5 tham chiếu T4

type T6[P int] struct{ f *T6[P] }     // hợp lệ:
tham chiếu T6 không nằm trong danh sách tham số
kiểu
```

Type constraints

Ràng buộc kiểu là một interface xác định tập hợp đối số kiểu hợp lệ cho tham số kiểu tương ứng và kiểm soát các phép toán được hỗ trợ bởi giá trị của tham số kiểu đó [Go 1.18].

TypeConstraint = TypeElem .

Nếu ràng buộc là interface literal dạng `interface{E}` với `E` là type element nhúng (không phải method), trong danh sách tham số kiểu có thể bỏ qua `interface{ … }` cho tiện:

```
[T []P]                         // = [T
interface{[]P}]
[T ~int]                        // = [T
interface{~int}]
[T int|string]                  // = [T
interface{int|string}]
type Constraint ~int            // không hợp lệ: ~int
không nằm trong danh sách tham số kiểu
```

Interface type định nghĩa sẵn `comparable` biểu diễn tập hợp tất cả các kiểu không phải interface mà so sánh nghiêm ngặt được [Go 1.18].

Dù interface không phải tham số kiểu là so sánh được, chúng không phải so sánh nghiêm ngặt và do đó không triển khai `comparable`. Tuy nhiên, chúng thỏa mãn `comparable`.

```
int                             // triển khai
comparable (int so sánh nghiêm ngặt)
[]byte                          // không triển khai
comparable (slice không so sánh được)
interface{}                     // không triển khai
comparable (xem trên)
interface{ ~int | ~string }  // chỉ tham số kiểu:
triển khai comparable (int, string so sánh nghiêm
ngặt)
interface{ comparable }      // chỉ tham số kiểu:
triển khai comparable (comparable tự triển khai)
interface{ ~int | ~[]byte }  // chỉ tham số kiểu:
không triển khai comparable (slice không so sánh
được)
interface{ ~struct{ any } }  // chỉ tham số kiểu:
không triển khai comparable (trường any không so
sánh nghiêm ngặt)
```

Interface `comparable` và các interface (trực tiếp hoặc gián tiếp) nhúng `comparable` chỉ được dùng làm ràng buộc kiểu. Chúng không thể là kiểu của giá trị hoặc biến, hoặc thành phần của kiểu khác không phải interface.

Satisfying a type constraint

Đối số kiểu `T` *thỏa mãn* ràng buộc kiểu `C` nếu `T` là phần tử của tập hợp kiểu do `C` xác định; nói cách khác, nếu `T` triển khai `C`. Ngoại lệ, ràng buộc kiểu so sánh nghiêm ngặt cũng có thể được thỏa mãn bởi đối số kiểu so sánh được (không nhất thiết so sánh nghiêm ngặt) [Go 1.20]. Cụ thể:

Kiểu T *thỏa mãn* ràng buộc `C` nếu

- `T` triển khai `C`; hoặc
- `C` có thể viết dưới dạng `interface{ comparable; E }`, với `E` là basic interface và `T` so sánh được và triển khai `E`.

```
type argument      type constraint
// constraint satisfaction

int               interface{ ~int }
// thỏa mãn: int triển khai interface{ ~int }
string            comparable
// thỏa mãn: string triển khai comparable (string
so sánh nghiêm ngặt)
[]byte            comparable
// không thỏa mãn: slice không so sánh được
any               interface{ comparable; int }
// không thỏa mãn: any không triển khai interface{
int }
any               comparable
// thỏa mãn: any so sánh được và triển khai basic
interface any
```

```
struct{f any}      comparable
// thỏa mãn: struct{f any} so sánh được và triển
khai basic interface any
any                interface{ comparable; m() }
// không thỏa mãn: any không triển khai basic
interface interface{ m() }
interface{ m() }   interface{ comparable; m() }
// thỏa mãn: interface{ m() } so sánh được và triển
khai basic interface interface{ m() }
```

Do ngoại lệ trong quy tắc thỏa mãn ràng buộc, việc so sánh toán hạng kiểu tham số kiểu có thể gây panic tại runtime (dù tham số kiểu comparable luôn so sánh nghiêm ngặt).

Variable declarations

Khai báo biến tạo một hoặc nhiều biến, liên kết các identifier tương ứng với chúng, và gán cho mỗi biến một kiểu và giá trị khởi tạo.

```
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]  // tra cứu map; chỉ
quan tâm đến "found"
```

Nếu có danh sách biểu thức, các biến được khởi tạo với các biểu thức theo quy tắc của câu lệnh gán. Nếu không, mỗi biến được khởi tạo với giá trị zero.

Nếu có kiểu, mỗi biến nhận kiểu đó. Nếu không, mỗi biến nhận kiểu của giá trị khởi tạo tương ứng trong phép gán. Nếu giá trị đó là hằng không kiểu, nó được chuyển đổi ngầm định sang kiểu mặc định; nếu là giá trị boolean không kiểu, nó được chuyển ngầm định sang kiểu `bool`. Identifier định nghĩa sẵn `nil` không thể dùng để khởi tạo biến không có kiểu tường minh.

```
var d = math.Sin(0.5)  // d là float64
var i = 42             // i là int
var t, ok = x.(T)      // t là T, ok là bool
var n = nil            // không hợp lệ
```

Giới hạn triển khai: Trình biên dịch có thể không cho phép khai báo biến trong thân hàm nếu biến đó không bao giờ được sử dụng.

Short variable declarations

Khai báo biến ngắn sử dụng cú pháp:

ShortVarDecl = IdentifierList ":=" ExpressionList .

Nó là dạng rút gọn của khai báo biến thông thường với biểu thức khởi tạo nhưng không có kiểu:

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

i, j := 0, 10
```

```
f := func() int { return 7 }
ch := make(chan int)
r, w, _ := os.Pipe()  // os.Pipe() trả về một cặp
File kết nối và một error, nếu có
_, y, _ := coord(p)   // coord() trả về ba giá trị;
chỉ quan tâm đến tọa độ y
```

Khác với khai báo biến thông thường, khai báo biến ngắn có thể *khai báo lại* biến với điều kiện biến đó đã được khai báo trước đó trong cùng block (hoặc danh sách tham số nếu block là thân hàm) với cùng kiểu, và ít nhất một biến không blank là mới. Do đó, khai báo lại chỉ xuất hiện trong khai báo biến ngắn nhiều biến. Khai báo lại không tạo biến mới; chỉ gán giá trị mới cho biến gốc. Tên biến không blank bên trái `:=` phải duy nhất.

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // khai
báo lại offset
x, y, x := 1, 2, 3                        // không
hợp lệ: x lặp lại bên trái :=
```

Khai báo biến ngắn chỉ xuất hiện bên trong hàm. Trong một số ngữ cảnh như khởi tạo cho câu lệnh "if", "for", hoặc "switch", chúng có thể dùng để khai báo biến tạm cục bộ.

Function declarations

Khai báo hàm liên kết một identifier, *tên hàm*, với một hàm.

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

```
FunctionName = identifier .
FunctionBody = Block .
```

Nếu chữ ký của hàm khai báo tham số kết quả, danh sách câu lệnh trong thân hàm phải kết thúc bằng một câu lệnh kết thúc.

```
func IndexRune(s string, r rune) int {
    for i, c := range s {
        if c == r {
            return i
        }
    }
    // không hợp lệ: thiếu câu lệnh return
}
```

Nếu khai báo hàm chỉ định tham số kiểu, tên hàm biểu diễn một *hàm generic*. Hàm generic phải được khởi tạo trước khi gọi hoặc sử dụng làm giá trị.

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

Khai báo hàm không có tham số kiểu có thể bỏ qua thân hàm. Khai báo như vậy cung cấp chữ ký cho hàm được cài đặt ngoài Go, như routine assembly.

```
func flushICache(begin, end uintptr)  // cài đặt
bên ngoài
```

Method declarations

Method là một hàm có *receiver*. Khai báo method liên kết một identifier, *tên method*, với một method, và liên kết method với *kiểu cơ sở* của receiver.

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

Receiver được chỉ định qua một phần tham số bổ sung trước tên method. Phần tham số này phải khai báo một tham số không variadic duy nhất, là receiver. Kiểu của nó phải là kiểu định nghĩa `T` hoặc con trỏ đến kiểu định nghĩa `T`, có thể theo sau bởi danh sách tên tham số kiểu `[P1, P2, …]` trong dấu ngoặc vuông. `T` gọi là *kiểu cơ sở receiver*. Kiểu cơ sở receiver không được là con trỏ hoặc interface và phải được định nghĩa trong cùng package với method. Method được gọi là *liên kết* với kiểu cơ sở receiver và tên method chỉ hiển thị trong selector cho kiểu `T` hoặc `*T`.

Identifier receiver không blank phải duy nhất trong chữ ký method. Nếu giá trị receiver không được tham chiếu trong thân method, identifier của nó có thể bị bỏ qua trong khai báo. Điều này cũng áp dụng cho tham số của hàm và method nói chung.

Với một kiểu cơ sở, các tên method không blank liên kết với nó phải duy nhất. Nếu kiểu cơ sở là struct, tên method và trường không blank phải khác nhau.

Với kiểu định nghĩa `Point`, các khai báo

```
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
}
```

liên kết các method `Length` và `Scale`, với receiver kiểu `*Point`, với kiểu cơ sở `Point`.

Nếu kiểu cơ sở receiver là kiểu generic, receiver phải khai báo các tham số kiểu tương ứng để method sử dụng. Điều này làm cho các tham số kiểu receiver khả dụng cho method. Về cú pháp, khai báo tham số kiểu này giống như khởi tạo kiểu cơ sở receiver: các đối số kiểu phải là identifier biểu diễn tham số kiểu được khai báo, một cho mỗi tham số kiểu của kiểu cơ sở receiver. Tên tham số kiểu không cần trùng với tên tham số tương ứng trong định nghĩa kiểu cơ sở receiver, và tất cả tên tham số không blank phải duy nhất trong phần tham số receiver và chữ ký method. Ràng buộc tham số kiểu receiver được ngầm định bởi định nghĩa kiểu cơ sở receiver: các tham số kiểu tương ứng có ràng buộc tương ứng.

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

func (p Pair[A, B]) Swap() Pair[B, A]  { … }  //
```

```
receiver khai báo A, B
func (p Pair[First, _]) First() First  { … }  //
receiver khai báo First, tương ứng với A trong Pair
```

Nếu kiểu receiver là (con trỏ đến) một alias, alias đó không được là generic và không được biểu diễn kiểu generic đã khởi tạo, dù trực tiếp hay gián tiếp qua alias khác, bất kể số lần gián tiếp con trỏ.

```
type GPoint[P any] = Point
type HPoint        = *GPoint[int]
type IPair         = Pair[int, int]

func (*GPoint[P]) Draw(P)   { … }  // không hợp lệ:
alias không được là generic
func (HPoint) Draw(P)       { … }  // không hợp lệ:
alias không được biểu diễn kiểu đã khởi tạo
GPoint[int]
func (*IPair) Second() int  { … }  // không hợp lệ:
alias không được biểu diễn kiểu đã khởi tạo
Pair[int, int]
```

Expressions

Một biểu thức xác định phép tính giá trị bằng cách áp dụng các toán tử và hàm lên các toán hạng.

Operands

Toán hạng biểu diễn các giá trị cơ bản trong một biểu thức. Một toán hạng có thể là một literal, một identifier (có thể đủ điều kiện) không blank biểu diễn một hằng số, biến, hoặc hàm, hoặc một biểu thức đặt trong dấu ngoặc.

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

Tên toán hạng biểu diễn một hàm generic có thể theo sau bởi danh sách type arguments; toán hạng kết quả là một hàm đã được khởi tạo.

Blank identifier chỉ có thể xuất hiện như toán hạng ở phía bên trái của câu lệnh gán.

Giới hạn triển khai: Trình biên dịch không bắt buộc phải báo lỗi nếu kiểu của toán hạng là tham số kiểu với type set rỗng. Các hàm có tham số kiểu như vậy không thể được khởi tạo; mọi nỗ lực sẽ gây lỗi tại vị trí khởi tạo.

Qualified identifiers

Một identifier đủ điều kiện là một identifier có tiền tố là tên package. Cả tên package và identifier đều không được blank.

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

Identifier đủ điều kiện truy cập một identifier trong package khác, package này phải được import. Identifier phải được export và khai báo trong package block của package đó.

math.Sin // biểu diễn hàm Sin trong package math

Composite literals

Composite literal tạo giá trị mới cho struct, array, slice, và map mỗi lần được đánh giá. Chúng gồm kiểu của literal theo sau là danh sách phần tử trong dấu ngoặc nhọn. Mỗi phần tử có thể tùy chọn có key tương ứng.

```
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 .
```

Trừ khi LiteralType là tham số kiểu, kiểu nền của nó phải là struct, array, slice, hoặc map (cú pháp đảm bảo điều này trừ khi kiểu được chỉ định là TypeName). Nếu LiteralType là tham số kiểu, tất cả các kiểu trong type set của nó phải có cùng kiểu nền là kiểu hợp lệ cho composite literal. Kiểu của các phần tử và key phải có thể gán cho trường, phần tử, và key tương ứng của kiểu `T`; không có chuyển đổi bổ sung. Key được hiểu là tên trường với struct literal, chỉ số với array và slice literal, và key với map literal. Với map literal, tất cả phần tử phải có key. Việc chỉ định nhiều phần tử với cùng tên trường hoặc giá trị key hằng là lỗi. Với key map không phải hằng, xem phần thứ tự đánh giá.

Với struct literal, áp dụng các quy tắc sau:

- Key phải là tên trường khai báo trong kiểu struct.
- Danh sách phần tử không có key phải liệt kê phần tử cho mỗi trường struct theo thứ tự khai báo.
- Nếu bất kỳ phần tử nào có key, mọi phần tử phải có key.
- Danh sách phần tử có key không cần có phần tử cho mọi trường struct. Trường bị bỏ qua nhận giá trị zero cho trường đó.
- Literal có thể bỏ qua danh sách phần tử; literal như vậy đánh giá thành giá trị zero cho kiểu của nó.
- Việc chỉ định phần tử cho trường không export của struct thuộc package khác là lỗi.

Với các khai báo

```
type Point3D struct { x, y, z float64 }
type Line struct { p, q Point3D }
```

có thể viết

```
origin := Point3D{}                            //
giá trị zero cho Point3D
line := Line{origin, Point3D{y: -4, z: 12.3}}  //
giá trị zero cho line.q.x
```

Với array và slice literal, áp dụng các quy tắc sau:

- Mỗi phần tử có chỉ số nguyên xác định vị trí trong array.
- Phần tử có key dùng key làm chỉ số. Key phải là hằng số không âm có thể biểu diễn bởi giá trị kiểu `int`; nếu có kiểu thì phải là kiểu số nguyên.
- Phần tử không có key dùng chỉ số của phần tử trước cộng một. Nếu phần tử đầu không có key, chỉ số là 0.

Lấy địa chỉ của composite literal tạo ra con trỏ đến một biến duy nhất được khởi tạo với giá trị literal.

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

Lưu ý rằng giá trị zero cho kiểu slice hoặc map không giống với giá trị đã khởi tạo nhưng rỗng của cùng kiểu. Do đó, lấy địa chỉ của composite literal slice hoặc map rỗng không giống với việc cấp phát giá trị slice hoặc map mới bằng new.

```
p1 := &[]int{}    // p1 trỏ đến slice đã khởi tạo,
rỗng với giá trị []int{} và độ dài 0
p2 := new([]int)  // p2 trỏ đến slice chưa khởi tạo
với giá trị nil và độ dài 0
```

Độ dài của array literal là độ dài chỉ định trong kiểu literal. Nếu literal cung cấp ít phần tử hơn độ dài, các phần tử còn thiếu nhận giá trị zero cho kiểu phần tử array. Việc cung cấp phần tử với chỉ số ngoài phạm vi array là lỗi. Ký hiệu ... chỉ định độ dài array bằng chỉ số phần tử lớn nhất cộng một.

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

Slice literal mô tả toàn bộ array literal nền. Do đó, độ dài và dung lượng của slice literal là chỉ số phần tử lớn nhất cộng một. Slice literal có dạng

```
[]T{x1, x2, … xn}
```

và là dạng rút gọn của phép slice áp dụng cho array:

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

Trong composite literal của array, slice, hoặc map kiểu `T`, phần tử hoặc key map là composite literal có thể bỏ qua kiểu literal nếu giống với kiểu phần tử hoặc key của `T`. Tương tự, phần tử hoặc key là địa chỉ của composite literal có thể bỏ qua `&T` khi kiểu phần tử hoặc key là `*T`.

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

type PPoint *Point
[2]*Point{{1.5, -3.5}, {}}          // giống
[2]*Point{&Point{1.5, -3.5}, &Point{}}
[2]PPoint{{1.5, -3.5}, {}}          // giống
```

```
[2]PPoint{PPoint(&Point{1.5, -3.5}),
PPoint(&Point{})}
```

Có thể xảy ra mơ hồ khi composite literal dùng dạng TypeName của LiteralType xuất hiện như toán hạng giữa từ khóa và dấu ngoặc nhọn mở của block trong câu lệnh "if", "for", hoặc "switch", và composite literal không nằm trong dấu ngoặc tròn, vuông, hoặc nhọn. Trong trường hợp hiếm này, dấu ngoặc nhọn mở của literal bị phân tích nhầm là dấu mở block câu lệnh. Để giải quyết, composite literal phải nằm trong dấu ngoặc tròn.

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

Ví dụ array, slice, và map literal hợp lệ:

```
// danh sách số nguyên tố
primes := []int{2, 3, 5, 7, 9, 2147483647}

// vowels[ch] là true nếu ch là nguyên âm
vowels := [128]bool{'a': true, 'e': true, 'i':
true, 'o': true, 'u': true, 'y': true}

// mảng [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}

// tần số Hz cho 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

Function literal biểu diễn một hàm ẩn danh. Function literal không thể khai báo tham số kiểu.

```
FunctionLit = "func" Signature FunctionBody .

func(a, b int, z float64) bool { return a*b <
int(z) }
```

Function literal có thể gán cho biến hoặc gọi trực tiếp.

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

Function literal là *closure*: chúng có thể tham chiếu biến định nghĩa trong hàm bao quanh. Các biến này được chia sẻ giữa hàm bao quanh và function literal, và tồn tại miễn là còn được truy cập.

Primary expressions

Primary expression là toán hạng cho các biểu thức một ngôi và hai ngôi.

```
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

Với một primary expression `x` không phải tên package, *selector expression*

x.f

biểu diễn trường hoặc method `f` của giá trị `x` (hoặc đôi khi `*x`; xem bên dưới). Identifier `f` gọi là *selector* (trường hoặc method); nó không được là blank identifier. Kiểu của selector expression là kiểu của `f`. Nếu `x` là tên package, xem phần qualified identifiers.

Selector `f` có thể biểu diễn trường hoặc method `f` của kiểu `T`, hoặc tham chiếu trường hoặc method `f` của trường nhúng lồng nhau của `T`. Số trường nhúng phải đi qua để đến `f` gọi là *độ sâu* của nó trong `T`. Độ sâu của trường hoặc method `f` khai báo trong `T` là 0. Độ sâu của trường hoặc method `f` khai báo trong trường nhúng `A` của `T` là độ sâu của `f` trong `A` cộng một.

Các quy tắc sau áp dụng cho selector:

1. Với giá trị `x` kiểu `T` hoặc `*T` (T không phải con trỏ hoặc interface), `x.f` biểu diễn trường hoặc method ở độ sâu nhỏ nhất trong `T` có `f`. Nếu không có đúng một `f` ở độ sâu nhỏ nhất, selector expression là không hợp lệ.
2. Với giá trị `x` kiểu `I` (I là interface), `x.f` biểu diễn method thực tế tên `f` của giá trị động của `x`. Nếu không có method tên `f` trong method set của `I`, selector expression là không hợp lệ.
3. Ngoại lệ: nếu kiểu của `x` là kiểu con trỏ định nghĩa và `(*x).f` là selector hợp lệ biểu diễn trường (không phải method), `x.f` là viết tắt cho `(*x).f`.
4. Các trường hợp khác, `x.f` là không hợp lệ.
5. Nếu `x` là con trỏ và có giá trị `nil` và `x.f` là trường struct, gán hoặc đánh giá `x.f` gây panic tại runtime.
6. Nếu `x` là interface và có giá trị `nil`, gọi hoặc đánh giá method `x.f` gây panic tại runtime.

Ví dụ, với các khai báo:

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

có thể viết:

```
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 là
selector trường hợp hợp lệ

p.M0()       // ((*p).T0).M0()      M0 nhận
receiver *T0
p.M1()       // ((*p).T1).M1()      M1 nhận
```

```
receiver T1
p.M2()       // p.M2()              M2 nhận
receiver *T2
t.M2()       // (&t).M2()           M2 nhận
receiver *T2, xem phần Calls
```

nhưng sau đây là không hợp lệ:

```
q.M0()       // (*q).M0 hợp lệ nhưng không phải
selector trường hợp
```

Method expressions

Nếu `M` nằm trong method set của kiểu `T`, `T.M` là một hàm có thể gọi như hàm thông thường với các đối số giống `M` nhưng thêm đối số đầu là receiver của method.

MethodExpr = ReceiverType "." MethodName . ReceiverType = Type .

Xét kiểu struct `T` với hai method, `Mv` (receiver kiểu `T`) và `Mp` (receiver kiểu `*T`).

```
type T struct {
    a int
}
func (tv  T) Mv(a int) int         { return 0 }  //
receiver giá trị
func (tp *T) Mp(f float32) float32 { return 1 }  //
receiver con trỏ

var t T
```

Biểu thức

T.Mv

trả về hàm tương đương với `Mv` nhưng receiver tường minh là đối số đầu; có chữ ký

func(tv T, a int) int

Hàm này có thể gọi bình thường với receiver tường minh, nên năm cách gọi sau là tương đương:

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

Tương tự, biểu thức

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

trả về giá trị hàm biểu diễn `Mp` với chữ ký

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

Với method receiver giá trị, có thể tạo hàm với receiver con trỏ tường minh, nên

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

trả về giá trị hàm biểu diễn `Mv` với chữ ký

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

Hàm như vậy sẽ gián tiếp qua receiver để tạo giá trị truyền vào method; method không ghi đè giá trị có địa chỉ truyền vào.

Trường hợp cuối, hàm receiver giá trị cho method receiver con trỏ là không hợp lệ vì method receiver con trỏ không nằm trong method set của kiểu giá trị.

Giá trị hàm tạo từ method được gọi bằng cú pháp gọi hàm; receiver là đối số đầu của lời gọi. Tức là, với `f := T.Mv`, gọi `f` là `f(t, 7)` chứ không phải `t.f(7)`. Để tạo hàm gắn receiver, dùng function literal hoặc method value.

Có thể tạo giá trị hàm từ method của kiểu interface. Hàm kết quả nhận receiver tường minh kiểu interface đó.

Method values

Nếu biểu thức `x` có kiểu tĩnh `T` và `M` nằm trong method set của `T`, `x.M` gọi là *method value*. Method value `x.M` là giá trị hàm có thể gọi với các đối số giống lời gọi method `x.M`. Biểu thức `x` được đánh giá và lưu lại khi đánh giá method value; bản sao lưu được dùng làm receiver trong mọi lời gọi, có thể thực hiện sau đó.

```
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 được
đánh giá và lưu trong f
g := s.M                    // receiver *(s.T) được
đánh giá và lưu trong g
*t = 42                     // không ảnh hưởng đến
receiver đã lưu trong f và g
```

Kiểu `T` có thể là interface hoặc không phải interface.

Như phần method expressions ở trên, xét kiểu struct `T` với hai method, `Mv` (receiver kiểu `T`) và `Mp` (receiver kiểu `*T`).

```
type T struct {
    a int
}
func (tv  T) Mv(a int) int         { return 0 }  //
receiver giá trị
func (tp *T) Mp(f float32) float32 { return 1 }  //
receiver con trỏ

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

Biểu thức

t.Mv

trả về giá trị hàm kiểu

func(int) int

Hai lời gọi sau là tương đương:

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

Tương tự, biểu thức

pt.Mp

trả về giá trị hàm kiểu

func(float32) float32

Như với selectors, tham chiếu method không phải interface với receiver giá trị dùng con trỏ sẽ tự động dereference: `pt.Mv` tương đương `(*pt).Mv`.

Như với method calls, tham chiếu method không phải interface với receiver con trỏ dùng giá trị có thể lấy địa chỉ sẽ tự động lấy địa chỉ: `t.Mp` tương đương `(&t).Mp`.

```
f := t.Mv; f(7)   // như t.Mv(7)
f := pt.Mp; f(7)  // như pt.Mp(7)
f := pt.Mv; f(7)  // như (*pt).Mv(7)
f := t.Mp; f(7)   // như (&t).Mp(7)
f := makeT().Mp   // không hợp lệ: kết quả makeT()
không thể lấy địa chỉ
```

Dù các ví dụ trên dùng kiểu không phải interface, cũng hợp lệ tạo method value từ giá trị kiểu interface.

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

Index expressions

Primary expression dạng

a[x]

biểu diễn phần tử của array, con trỏ đến array, slice, string hoặc map `a` với chỉ số `x`. Giá trị `x` gọi là *index* hoặc *map key*. Áp dụng các quy tắc sau:

Nếu `a` không phải map cũng không phải tham số kiểu:

- chỉ số `x` phải là hằng không kiểu, hoặc kiểu của nó phải là số nguyên hoặc tham số kiểu với type set chỉ chứa kiểu số nguyên
- chỉ số hằng phải không âm và có thể biểu diễn bởi giá trị kiểu `int`
- chỉ số hằng không kiểu được gán kiểu `int`
- chỉ số `x` *trong phạm vi* nếu `0 <= x < len(a)`, ngược lại *ngoài phạm vi*

Với `a` kiểu array `A`:

- chỉ số hằng phải trong phạm vi
- nếu `x` ngoài phạm vi tại runtime, panic tại runtime
- `a[x]` là phần tử array tại chỉ số `x`, kiểu là kiểu phần tử của `A`

Với `a` là con trỏ đến array:

- `a[x]` là viết tắt cho `(*a)[x]`

Với `a` kiểu slice `S`:

- nếu `x` ngoài phạm vi tại runtime, panic tại runtime
- `a[x]` là phần tử slice tại chỉ số `x`, kiểu là kiểu phần tử của `S`

Với `a` kiểu string:

- chỉ số hằng phải trong phạm vi nếu string `a` cũng là hằng
- nếu `x` ngoài phạm vi tại runtime, panic tại runtime
- `a[x]` là giá trị byte không hằng tại chỉ số `x`, kiểu là `byte`

- không được gán cho `a[x]`

Với `a` kiểu map `M`:

- kiểu của `x` phải có thể gán cho kiểu key của `M`
- nếu map chứa entry với key `x`, `a[x]` là phần tử map với key `x`, kiểu là kiểu phần tử của `M`
- nếu map là `nil` hoặc không có entry như vậy, `a[x]` là giá trị zero cho kiểu phần tử của `M`

Với `a` kiểu tham số kiểu `P`:

- Biểu thức chỉ số `a[x]` phải hợp lệ với mọi kiểu trong type set của `P`.
- Kiểu phần tử của mọi kiểu trong type set của `P` phải giống hệt nhau. Với string, kiểu phần tử là `byte`.
- Nếu có kiểu map trong type set của `P`, mọi kiểu trong type set đó phải là map, và kiểu key phải giống hệt nhau.
- `a[x]` là phần tử array, slice, hoặc string tại chỉ số `x`, hoặc phần tử map với key `x` của đối số kiểu mà `P` được khởi tạo, kiểu của `a[x]` là kiểu phần tử giống hệt.
- Không được gán cho `a[x]` nếu type set của `P` có kiểu string.

Nếu không, `a[x]` là không hợp lệ.

Biểu thức chỉ số trên map `a` kiểu `map[K]V` dùng trong câu lệnh gán hoặc khởi tạo dạng đặc biệt

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

trả về thêm giá trị boolean không kiểu. Giá trị của `ok` là `true` nếu key `x` có trong map, ngược lại là `false`.

Gán cho phần tử của map `nil` gây panic tại runtime.

Slice expressions

Slice expression tạo substring hoặc slice từ toán hạng string, array, con trỏ đến array, hoặc slice. Có hai dạng: dạng đơn giản chỉ định giới hạn thấp và cao, và dạng đầy đủ chỉ định thêm giới hạn dung lượng.

Nếu kiểu toán hạng là tham số kiểu, trừ khi type set chứa kiểu string, mọi kiểu trong type set phải có cùng kiểu nền, và slice expression phải hợp lệ với toán hạng kiểu đó. Nếu type set chứa kiểu string, có thể chứa thêm slice byte với kiểu nền `[]byte`. Khi đó, slice expression phải hợp lệ với toán hạng kiểu `string`.

Simple slice expressions

Với string, array, con trỏ đến array, hoặc slice `a`, primary expression

a[low : high]

tạo substring hoặc slice. *Chỉ số* `low` và `high` chọn phần tử của toán hạng `a` xuất hiện trong kết quả. Kết quả có chỉ số bắt đầu từ 0 và độ dài bằng `high` - `low`. Sau khi slice array `a`:

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

slice `s` có kiểu `[]int`, độ dài 3, dung lượng 4, và phần tử

```
s[0] == 2
s[1] == 3
s[2] == 4
```

Có thể bỏ qua bất kỳ chỉ số nào. Thiếu `low` mặc định là 0; thiếu `high` mặc định là độ dài toán hạng slice:

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

Nếu `a` là con trỏ đến array, `a[low : high]` là viết tắt cho `(*a)[low : high]`.

Với array hoặc string, chỉ số *trong phạm vi* nếu `0 <= low <= high <= len(a)`, ngược lại *ngoài phạm vi*. Với slice, giới hạn trên là dung lượng slice `cap(a)` thay vì độ dài. Chỉ số hằng phải không âm và có thể biểu diễn bởi giá trị kiểu `int`; với array hoặc string hằng, chỉ số hằng cũng phải trong phạm vi. Nếu cả hai chỉ số là hằng, phải thỏa mãn `low <= high`. Nếu chỉ số ngoài phạm vi tại runtime, panic tại runtime.

Trừ string không kiểu, nếu toán hạng slice là string hoặc slice, kết quả phép slice là giá trị không hằng cùng kiểu với toán hạng. Với string không kiểu, kết quả là giá trị không hằng kiểu `string`. Nếu toán hạng slice là array, nó phải có thể lấy địa chỉ và kết quả phép slice là slice cùng kiểu phần tử với array.

Nếu toán hạng slice hợp lệ là slice `nil`, kết quả là slice `nil`. Nếu không, nếu kết quả là slice, nó chia sẻ mảng nền với toán hạng.

```
var a [10]int
s1 := a[3:7]   // mảng nền của s1 là a; &s1[2] ==
&a[5]
s2 := s1[1:4]  // mảng nền của s2 là mảng nền của
s1 là a; &s2[1] == &a[5]
s2[1] = 42     // s2[1] == s1[2] == a[5] == 42; tất
cả tham chiếu cùng phần tử mảng nền

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

Full slice expressions

Với array, con trỏ đến array, hoặc slice `a` (không phải string), primary expression

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

tạo slice cùng kiểu, cùng độ dài và phần tử như slice expression đơn giản `a[low : high]`. Ngoài ra, nó kiểm soát dung lượng slice kết quả bằng cách đặt thành `max - low`. Chỉ chỉ số đầu có thể bỏ qua; mặc định là 0. Sau khi slice array `a`:

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

slice `t` có kiểu `[]int`, độ dài 2, dung lượng 4, và phần tử

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

Như với slice expression đơn giản, nếu `a` là con trỏ đến array, `a[low : high : max]` là viết tắt cho `(*a)[low : high : max]`. Nếu toán hạng slice là array, nó phải có thể lấy địa chỉ.

Chỉ số *trong phạm vi* nếu `0 <= low <= high <= max <= cap(a)`, ngược lại *ngoài phạm vi*. Chỉ số hằng phải không âm và có thể biểu diễn bởi giá trị kiểu `int`; với array, chỉ số hằng cũng phải trong phạm vi. Nếu có nhiều chỉ số hằng, các chỉ số hiện diện phải trong phạm vi lẫn nhau. Nếu chỉ số ngoài phạm vi tại runtime, panic tại runtime.

Type assertions

Với biểu thức `x` kiểu interface, không phải tham số kiểu, và kiểu `T`, primary expression

```
x.(T)
```

khẳng định rằng `x` không phải `nil` và giá trị lưu trong `x` có kiểu `T`. Ký hiệu `x.(T)` gọi là *type assertion*.

Cụ thể, nếu `T` không phải interface, `x.(T)` khẳng định kiểu động của `x` giống hệt kiểu `T`. Khi đó, `T` phải triển khai kiểu (interface) của `x`; nếu không, type assertion không hợp lệ vì không thể lưu giá trị kiểu `T` trong `x`. Nếu `T` là interface, `x.(T)` khẳng định kiểu động của `x` triển khai interface `T`.

Nếu type assertion đúng, giá trị biểu thức là giá trị lưu trong `x` và kiểu là `T`. Nếu sai, panic tại runtime. Nói cách khác, dù kiểu động của `x` chỉ biết

tại runtime, kiểu của `x.(T)` được biết là `T` trong chương trình đúng.

```
var x interface{} = 7          // x có kiểu động
int và giá trị 7
i := x.(int)                   // i có kiểu int và
giá trị 7

type I interface { m() }

func f(y I) {
    s := y.(string)        // không hợp lệ: string
không triển khai I (thiếu method m)
    r := y.(io.Reader)     // r có kiểu io.Reader
và kiểu động của y phải triển khai cả I và
io.Reader
    …
}
```

Type assertion dùng trong câu lệnh gán hoặc khởi tạo dạng đặc biệt

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
var v, ok interface{} = x.(T) // kiểu động của v và
ok là T và bool
```

trả về thêm giá trị boolean không kiểu. Giá trị của `ok` là `true` nếu assertion đúng. Nếu không, là `false` và giá trị của `v` là giá trị zero cho kiểu `T`. Không có panic tại runtime trong trường hợp này.

Calls

Với biểu thức `f` kiểu hàm `F`,

```
f(a1, a2, … an)
```

gọi `f` với các đối số `a1, a2, … an`. Trừ một trường hợp đặc biệt, các đối số phải là biểu thức đơn giá trị có thể gán cho kiểu tham số của `F` và được đánh giá trước khi gọi hàm. Kiểu của biểu thức là kiểu kết quả của `F`. Gọi method tương tự nhưng method được chỉ định là selector trên giá trị kiểu receiver của method.

```
math.Atan2(x, y)  // gọi hàm
var pt *Point
pt.Scale(3.5)     // gọi method với receiver pt
```

Nếu `f` là hàm generic, phải được khởi tạo trước khi gọi hoặc dùng làm giá trị hàm.

Nếu kiểu của `f` là tham số kiểu, mọi kiểu trong type set phải có cùng kiểu nền là kiểu hàm, và lời gọi hàm phải hợp lệ với kiểu đó.

Trong lời gọi hàm, giá trị hàm và đối số được đánh giá theo thứ tự thông thường. Sau khi đánh giá, bộ nhớ mới được cấp phát cho biến của hàm, gồm tham số và kết quả. Sau đó, đối số được *truyền* vào hàm, nghĩa là chúng được gán cho tham số hàm, và hàm bắt đầu thực thi. Tham số trả về được trả lại cho caller khi hàm trả về.

Gọi giá trị hàm `nil` gây panic tại runtime.

Trường hợp đặc biệt, nếu giá trị trả về của hàm hoặc method `g` bằng số lượng và từng giá trị có thể gán cho tham số của hàm hoặc method `f`, thì lời gọi `f(g(_parameters_of_g_))` sẽ gọi `f` sau khi truyền giá trị trả về của `g` cho tham số của `f` theo thứ tự. Lời gọi `f` không có tham số nào ngoài lời gọi `g`, và `g` phải trả về ít nhất một giá trị. Nếu `f` có tham số

cuối `...`, nó nhận các giá trị trả về còn lại của `g` sau khi gán cho tham số thông thường.

```
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")
}
```

Lời gọi method `x.m()` hợp lệ nếu method set của (kiểu của) `x` chứa `m` và danh sách đối số có thể gán cho danh sách tham số của `m`. Nếu `x` có thể lấy địa chỉ và method set của `&x` chứa `m`, `x.m()` là viết tắt cho `(&x).m()`:

var p Point p.Scale(3.5)

Không có kiểu method riêng biệt và không có method literal.

Passing arguments to `...` parameters

Nếu `f` là variadic với tham số cuối `p` kiểu `...T`, thì trong `f` kiểu của `p` tương đương kiểu `[]T`. Nếu gọi `f` không có đối số cho `p`, giá trị truyền cho `p` là `nil`. Nếu không, giá trị truyền là slice mới kiểu `[]T` với mảng nền mới, các phần tử là đối số thực tế, tất cả phải có thể gán cho `T`. Độ dài và dung lượng slice là số đối số gán cho `p` và có thể khác nhau ở mỗi nơi gọi.

Với hàm và lời gọi

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

trong `Greeting`, `who` sẽ có giá trị `nil` ở lần gọi đầu, và `[]string{"Joe", "Anna", "Eileen"}` ở lần gọi thứ hai.

Nếu đối số cuối có thể gán cho kiểu slice `[]T` và theo sau là `...`, nó được truyền nguyên vẹn làm giá trị cho tham số `...T`. Khi đó không tạo slice mới.

Với slice `s` và lời gọi

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

trong `Greeting`, `who` sẽ có cùng giá trị với `s` và cùng mảng nền.

Instantiations

Hàm hoặc kiểu generic được *khởi tạo* bằng cách thay thế *type arguments* cho các tham số kiểu [Go 1.18]. Khởi tạo gồm hai bước:

1. Mỗi type argument được thay cho tham số kiểu tương ứng trong khai báo generic. Việc thay thế này diễn ra trên toàn bộ khai báo hàm hoặc kiểu, bao gồm cả danh sách tham số kiểu và mọi kiểu trong danh sách đó.
2. Sau khi thay thế, mỗi type argument phải thỏa mãn ràng buộc (đã khởi tạo nếu cần) của tham số kiểu tương ứng. Nếu không, khởi tạo thất bại.

Khởi tạo kiểu tạo ra kiểu có tên không generic mới; khởi tạo hàm tạo ra hàm không generic mới.

```
type parameter list    type arguments    after
substitution

[P any]                int               int thỏa
mãn any
[S ~[]E, E any]        []int, int        []int thỏa
mãn ~[]int, int thỏa mãn any
[P io.Writer]          string            không hợp
lệ: string không thỏa mãn io.Writer
[P comparable]         any               any thỏa
mãn (nhưng không triển khai) comparable
```

Khi dùng hàm generic, type argument có thể cung cấp tường minh, hoặc có thể được suy luận một phần hoặc toàn bộ từ ngữ cảnh sử dụng hàm. Nếu có thể suy luận, danh sách type argument có thể bỏ qua hoàn toàn nếu hàm được:

- gọi với đối số thông thường,
- gán cho biến có kiểu đã biết
- truyền làm đối số cho hàm khác, hoặc
- trả về làm kết quả.

Các trường hợp khác, phải có danh sách type argument (có thể một phần). Nếu danh sách type argument thiếu hoặc một phần, mọi type argument thiếu phải suy luận được từ ngữ cảnh sử dụng hàm.

```
// sum trả về tổng (nối chuỗi, với string) của các
đối số.
func sum[T ~int | ~float64 | ~string](x... T) T { …
}
```

```
x := sum                          // không hợp lệ:
kiểu của x không xác định
intSum := sum[int]                // intSum có kiểu
func(x... int) int
a := intSum(2, 3)                 // a có giá trị 5
kiểu int
b := sum[float64](2.0, 3)         // b có giá trị 5.0
kiểu float64
c := sum(b, -1)                   // c có giá trị 4.0
kiểu float64

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

Danh sách type argument một phần không được rỗng; ít nhất phải có đối số đầu. Danh sách là tiền tố của danh sách type argument đầy đủ, các đối số còn lại sẽ được suy luận. Nói chung, type argument có thể bỏ từ "phải sang trái".

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

f0 := apply[]                     // không hợp lệ:
danh sách type argument không được rỗng
f1 := apply[[]int]                // type argument cho
S cung cấp tường minh, cho E suy luận
f2 := apply[[]string, string]     // cả hai type
argument cung cấp tường minh

var bytes []byte
r := apply(bytes, func(byte) byte { … })  // cả hai
type argument suy luận từ đối số hàm
```

Với kiểu generic, luôn phải cung cấp đầy đủ type argument tường minh.

Type inference

Việc sử dụng một hàm generic có thể bỏ qua một phần hoặc toàn bộ type argument nếu chúng có thể được *suy luận* từ ngữ cảnh sử dụng hàm, bao gồm cả các ràng buộc của tham số kiểu của hàm. Suy luận kiểu thành công nếu nó có thể suy luận các type argument còn thiếu và khởi tạo thành công với các type argument đã suy luận. Nếu không, suy luận kiểu thất bại và chương trình không hợp lệ.

Suy luận kiểu sử dụng mối quan hệ kiểu giữa các cặp kiểu để suy luận: Ví dụ, một đối số hàm phải có thể gán cho kiểu tham số hàm tương ứng; điều này thiết lập mối quan hệ giữa kiểu của đối số và kiểu của tham số. Nếu một trong hai kiểu này chứa tham số kiểu, suy luận kiểu sẽ tìm các type argument để thay thế tham số kiểu sao cho mối quan hệ gán được thỏa mãn. Tương tự, suy luận kiểu sử dụng thực tế rằng một type argument phải thỏa mãn ràng buộc của tham số kiểu tương ứng.

Mỗi cặp kiểu khớp như vậy tương ứng với một *phương trình kiểu* chứa một hoặc nhiều tham số kiểu, từ một hoặc nhiều hàm generic. Suy luận các type argument còn thiếu nghĩa là giải hệ phương trình kiểu cho các tham số kiểu tương ứng.

Ví dụ, với

```
// dedup trả về bản sao của slice đối số với các
phần tử trùng lặp bị loại bỏ.
func dedup[S ~[]E, E comparable](S) S { … }

type Slice []int
```

```
var s Slice
s = dedup(s)   // giống như s = dedup[Slice, int]
(s)
```

biến `s` kiểu `Slice` phải có thể gán cho kiểu tham số hàm `S` để chương trình hợp lệ. Để giảm độ phức tạp, suy luận kiểu bỏ qua hướng của phép gán, nên mối quan hệ kiểu giữa `Slice` và `S` có thể biểu diễn bằng phương trình kiểu đối xứng `Slice ≡A S` (hoặc `S ≡A Slice`), trong đó `A` trong `≡A` chỉ rằng kiểu bên trái và phải phải khớp theo quy tắc gán (xem phần type unification để biết chi tiết). Tương tự, tham số kiểu `S` phải thỏa mãn ràng buộc `~[]E`. Điều này có thể biểu diễn là `S ≡C ~[]E` với `X ≡C Y` nghĩa là "`X` thỏa mãn ràng buộc `Y`". Các quan sát này dẫn đến hai phương trình

```
Slice ≡A S      (1)
S     ≡C ~[]E   (2)
```

có thể giải cho các tham số kiểu `S` và `E`. Từ (1) trình biên dịch có thể suy luận type argument cho `S` là `Slice`. Tương tự, vì kiểu nền của `Slice` là `[]int` và `[]int` phải khớp với `[]E` của ràng buộc, trình biên dịch có thể suy luận `E` phải là `int`. Như vậy, với hai phương trình này, suy luận kiểu sẽ suy ra

```
S → Slice
E → int
```

Với một tập hợp phương trình kiểu, các tham số kiểu cần giải là các tham số kiểu của các hàm cần khởi tạo và chưa có type argument tường minh. Các tham số kiểu này gọi là *tham số kiểu bị ràng buộc*. Ví dụ, trong ví dụ `dedup` ở trên, các tham số kiểu `S` và `E` bị ràng buộc với

`dedup`. Một đối số cho lời gọi hàm generic có thể là một hàm generic. Các tham số kiểu của hàm đó cũng được đưa vào tập tham số kiểu bị ràng buộc. Kiểu của đối số hàm có thể chứa tham số kiểu từ hàm khác (như hàm generic bao ngoài lời gọi hàm). Các tham số kiểu đó cũng có thể xuất hiện trong phương trình kiểu nhưng không bị ràng buộc trong ngữ cảnh đó. Phương trình kiểu luôn được giải cho các tham số kiểu bị ràng buộc.

Suy luận kiểu hỗ trợ gọi hàm generic và gán hàm generic cho biến (có kiểu hàm tường minh). Điều này bao gồm truyền hàm generic làm đối số cho hàm khác (có thể cũng là generic), và trả về hàm generic làm kết quả. Suy luận kiểu hoạt động trên một tập phương trình riêng cho từng trường hợp này. Các phương trình như sau (bỏ qua danh sách type argument cho rõ ràng):

- Với lời gọi hàm `f(a0, a1, …)` trong đó `f` hoặc một đối số hàm `ai` là hàm generic:
  Mỗi cặp `(ai, pi)` của đối số hàm và tham số tương ứng mà `ai` không phải hằng không kiểu tạo ra phương trình `typeof(pi) ≡A typeof(ai)`.
  Nếu `ai` là hằng không kiểu `cj`, và `typeof(pi)` là tham số kiểu bị ràng buộc `Pk`, cặp `(cj, Pk)` được thu thập riêng khỏi phương trình kiểu.
- Với phép gán `v = f` của hàm generic `f` cho biến (không generic) `v` kiểu hàm:
  `typeof(v) ≡A typeof(f)`.
- Với câu lệnh return `return …, f, …` trong đó `f` là hàm generic trả về cho biến kết quả (không generic) `r` kiểu hàm:
  `typeof(r) ≡A typeof(f)`.

Ngoài ra, mỗi tham số kiểu `Pk` và ràng buộc kiểu tương ứng `Ck` tạo ra phương trình kiểu `Pk ≡C Ck`.

Suy luận kiểu ưu tiên thông tin kiểu lấy từ toán hạng đã có kiểu trước khi xét đến hằng không kiểu. Do đó, suy luận diễn ra theo hai pha:

1. Phương trình kiểu được giải cho các tham số kiểu bị ràng buộc bằng type unification. Nếu unification thất bại, suy luận kiểu thất bại.
2. Với mỗi tham số kiểu bị ràng buộc `Pk` chưa có type argument suy luận và có một hoặc nhiều cặp `(cj, Pk)` với cùng tham số kiểu đó, xác định loại hằng của các hằng `cj` trong các cặp đó giống như với biểu thức hằng. Type argument cho `Pk` là kiểu mặc định cho loại hằng xác định. Nếu không xác định được loại hằng do xung đột, suy luận kiểu thất bại.

Nếu sau hai pha này vẫn chưa tìm đủ type argument, suy luận kiểu thất bại.

Nếu cả hai pha thành công, suy luận kiểu xác định được type argument cho mỗi tham số kiểu bị ràng buộc:

```
Pk → Ak
```

Type argument `Ak` có thể là kiểu tổng hợp, chứa các tham số kiểu bị ràng buộc khác `Pk` làm kiểu phần tử (hoặc chỉ là một tham số kiểu bị ràng buộc khác). Trong quá trình đơn giản hóa lặp lại, các tham số kiểu bị ràng buộc trong mỗi type argument được thay thế bằng type argument tương ứng cho đến khi mỗi type argument không còn tham số kiểu bị ràng buộc.

Nếu type argument chứa tham chiếu vòng lặp đến chính nó qua tham số kiểu bị ràng buộc, quá trình đơn giản hóa và do đó suy luận kiểu thất bại. Nếu không, suy luận kiểu thành công.

Type unification

Suy luận kiểu giải phương trình kiểu thông qua *type unification*. Type unification so sánh đệ quy kiểu bên trái và phải của phương trình, trong đó một hoặc cả hai kiểu có thể là hoặc chứa tham số kiểu bị ràng buộc, và tìm type argument cho các tham số kiểu đó sao cho hai bên khớp (trở nên giống hệt hoặc tương thích gán, tùy ngữ cảnh). Để làm điều này, suy luận kiểu duy trì một map từ tham số kiểu bị ràng buộc đến type argument đã suy luận; map này được tham chiếu và cập nhật trong quá trình unification. Ban đầu, các tham số kiểu bị ràng buộc đã biết nhưng map rỗng. Trong quá trình unification, nếu suy luận được type argument mới `A`, ánh xạ `P → A` từ tham số kiểu đến argument được thêm vào map. Ngược lại, khi so sánh kiểu, type argument đã biết (đã có trong map) sẽ thay thế tham số kiểu tương ứng. Khi suy luận kiểu tiến triển, map được điền dần cho đến khi xét hết các phương trình, hoặc unification thất bại. Suy luận kiểu thành công nếu không có bước unification nào thất bại và map có entry cho mỗi tham số kiểu.

Ví dụ, với phương trình kiểu có tham số kiểu bị ràng buộc `P`:

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

suy luận kiểu bắt đầu với map rỗng. Unification so sánh cấu trúc cấp cao nhất của hai kiểu. Cả hai là array cùng độ dài; chúng unify nếu kiểu phần tử unify. Cả hai kiểu phần tử là struct; chúng unify nếu có cùng số trường với cùng tên và kiểu trường unify. Type argument cho `P` chưa biết (chưa có entry trong map), nên unify `P` với `string` thêm ánh xạ `P → string` vào map. Unify kiểu trường `list` cần unify `[]P` và `[]string` và do đó `P` và `string`. Vì type argument cho `P` đã biết (có entry trong map), type argument `string` thay thế cho `P`. Và vì `string` giống hệt `string`, bước unify này cũng thành công. Unification của hai vế phương

trình hoàn tất. Suy luận kiểu thành công vì chỉ có một phương trình kiểu, không bước unify nào thất bại, và map đã đầy đủ.

Unification sử dụng kết hợp *unify chính xác* và *unify lỏng* tùy thuộc vào việc hai kiểu phải giống hệt, tương thích gán, hay chỉ cần giống cấu trúc. Quy tắc unify kiểu chi tiết được trình bày trong Phụ lục.

Với phương trình dạng $X \equiv_A Y$, trong đó X và Y là kiểu liên quan đến phép gán (bao gồm truyền tham số và return), cấu trúc kiểu cấp cao nhất có thể unify lỏng nhưng kiểu phần tử phải unify chính xác, theo quy tắc gán.

Với phương trình dạng $P \equiv_C C$, trong đó P là tham số kiểu và C là ràng buộc tương ứng, quy tắc unify phức tạp hơn:

- Nếu tất cả kiểu trong type set của C có cùng kiểu nền U, và P đã có type argument A, U và A phải unify lỏng.
- Tương tự, nếu tất cả kiểu trong type set của C là channel với cùng kiểu phần tử và hướng channel không xung đột, và P đã có type argument A, kiểu channel hạn chế nhất trong type set của C và A phải unify lỏng.
- Nếu P chưa có type argument và C chỉ có đúng một type term T không phải kiểu nền (tilde), unify thêm ánh xạ $P \to T$ vào map.
- Nếu C không có kiểu U như trên và P đã có type argument A, A phải có tất cả method của C (nếu có), và kiểu method tương ứng phải unify chính xác.

Khi giải phương trình kiểu từ ràng buộc kiểu, giải một phương trình có thể suy luận thêm type argument, từ đó cho phép giải các phương trình khác phụ thuộc vào type argument đó. Suy luận kiểu lặp lại unify kiểu miễn là còn suy luận được type argument mới.

Operators

Toán tử kết hợp các toán hạng thành biểu thức.

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

So sánh được thảo luận ở chỗ khác. Với các toán tử nhị phân khác, kiểu toán hạng phải giống hệt trừ khi phép toán liên quan đến shift hoặc hằng không kiểu. Với phép toán chỉ liên quan đến hằng, xem phần biểu thức hằng.

Trừ phép shift, nếu một toán hạng là hằng không kiểu và toán hạng còn lại không phải, hằng được chuyển đổi ngầm định sang kiểu của toán hạng còn lại.

Toán hạng phải của biểu thức shift phải có kiểu số nguyên [Go 1.13] hoặc là hằng không kiểu có thể biểu diễn bởi giá trị kiểu `uint`. Nếu toán hạng trái của biểu thức shift không phải hằng là hằng không kiểu, nó được chuyển ngầm định sang kiểu mà nó sẽ nhận nếu biểu thức shift được thay bằng toán hạng trái.

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

```
// Kết quả các ví dụ sau cho int 64 bit.
var i = 1<<s                    // 1 có kiểu int
var j int32 = 1<<s              // 1 có kiểu int32;
j == 0
var k = uint64(1<<s)            // 1 có kiểu uint64;
k == 1<<33
var m int = 1.0<<s              // 1.0 có kiểu int;
m == 1<<33
var n = 1.0<<s == j             // 1.0 có kiểu
int32; n == true
var o = 1<<s == 2<<s            // 1 và 2 có kiểu
int; o == false
var p = 1<<s == 1<<33           // 1 có kiểu int; p
== true
var u = 1.0<<s                  // không hợp lệ: 1.0
có kiểu float64, không thể shift
var u1 = 1.0<<s != 0            // không hợp lệ: 1.0
có kiểu float64, không thể shift
var u2 = 1<<s != 1.0            // không hợp lệ: 1
có kiểu float64, không thể shift
var v1 float32 = 1<<s           // không hợp lệ: 1
có kiểu float32, không thể shift
var v2 = string(1<<s)           // không hợp lệ: 1
được chuyển sang string, không thể shift
var w int64 = 1.0<<33           // 1.0<<33 là biểu
thức shift hằng; w == 1<<33
var x = a[1.0<<s]               // panic: 1.0 có
kiểu int, nhưng 1<<33 vượt quá giới hạn array
var b = make([]byte, 1.0<<s)    // 1.0 có kiểu int;
len(b) == 1<<33

// Kết quả các ví dụ sau cho int 32 bit, shift sẽ
tràn.
var mm int = 1.0<<s             // 1.0 có kiểu int;
mm == 0
var oo = 1<<s == 2<<s           // 1 và 2 có kiểu
int; oo == true
```

```
var pp = 1<<s == 1<<33          // không hợp lệ: 1
có kiểu int, nhưng 1<<33 tràn int
var xx = a[1.0<<s]              // 1.0 có kiểu int;
xx == a[0]
var bb = make([]byte, 1.0<<s)  // 1.0 có kiểu int;
len(bb) == 0
```

Operator precedence

Toán tử một ngôi có độ ưu tiên cao nhất. Vì toán tử `++` và `--` tạo thành câu lệnh, không phải biểu thức, nên chúng nằm ngoài hệ thống ưu tiên toán tử. Do đó, câu lệnh `*p++` giống như `(*p)++`.

Có năm mức ưu tiên cho toán tử nhị phân. Toán tử nhân mạnh nhất, tiếp theo là toán tử cộng, toán tử so sánh, `&&` (AND logic), và cuối cùng là `||` (OR logic):

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

Toán tử nhị phân cùng ưu tiên kết hợp từ trái sang phải. Ví dụ, `x / y * z` giống như `(x / y) * z`.

```
+x                          // x
42 + a - b                  // (42 + a) - b
23 + 3*x[i]                 // 23 + (3 * x[i])
x <= f()                    // x <= f()
^a >> b                     // (^a) >> b
```

```
f() || g()                  // f() || g()
x == y+1 && <-chanInt > 0   // (x == (y+1)) && ((<-
chanInt) > 0)
```

Arithmetic operators

Toán tử số học áp dụng cho giá trị số và trả về kết quả cùng kiểu với toán hạng đầu. Bốn toán tử số học chuẩn (+, -, *, /) áp dụng cho số nguyên, số thực, và số phức; + cũng áp dụng cho chuỗi. Toán tử logic bit và shift chỉ áp dụng cho số nguyên.

```
+    tổng                    số nguyên, số thực, số
phức, chuỗi
-    hiệu                    số nguyên, số thực, số
phức
*    tích                    số nguyên, số thực, số
phức
/    thương                  số nguyên, số thực, số
phức
%    dư                      số nguyên

&    AND bit                 số nguyên
|    OR bit                  số nguyên
^    XOR bit                 số nguyên
&^   AND NOT bit             số nguyên

<<   shift trái             số nguyên << số nguyên
>= 0
>>   shift phải             số nguyên >> số nguyên
>= 0
```

Nếu kiểu toán hạng là tham số kiểu, toán tử phải áp dụng cho từng kiểu trong type set. Toán hạng được biểu diễn là giá trị của type

argument mà tham số kiểu được khởi tạo, và phép toán được tính với độ chính xác của type argument đó. Ví dụ, với hàm:

```
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
}
```

tích `x * y` và phép cộng `s += x * y` được tính với độ chính xác `float32` hoặc `float64` tùy type argument cho `F`.

Integer operators

Với hai giá trị số nguyên `x` và `y`, thương số nguyên `q = x / y` và dư `r = x % y` thỏa mãn:

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

với `x / y` làm tròn về 0 ("truncated division").

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

Ngoại lệ duy nhất là nếu số bị chia `x` là giá trị âm nhất cho kiểu int của `x`, thương `q = x / -1` bằng `x` (và `r = 0`) do tràn số nguyên hai bù:

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

Nếu số chia là hằng, nó không được là 0. Nếu số chia là 0 tại runtime, panic tại runtime. Nếu số bị chia không âm và số chia là lũy thừa của 2, phép chia có thể thay bằng shift phải, và phép lấy dư có thể thay bằng AND bit:

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

Toán tử shift dịch toán hạng trái theo số lần chỉ định bởi toán hạng phải, phải không âm. Nếu số shift âm tại runtime, panic tại runtime. Toán tử shift thực hiện shift số học nếu toán hạng trái là số nguyên có dấu và shift logic nếu là số nguyên không dấu. Không giới hạn trên cho số shift. Shift như thể toán hạng trái được shift `n` lần bởi 1 với số shift là `n`. Do đó, `x << 1` giống `x*2` và `x >> 1` giống `x/2` nhưng làm tròn về âm vô cùng.

Với số nguyên, toán tử một ngôi `+`, `-`, và `^` được định nghĩa như sau:

```
+x                          là 0 + x
-x    phủ định              là 0 - x
```

```
^x    đảo bit                là m ^ x  với m = "tất
cả bit là 1" cho x không dấu
                                        và  m = -1
cho x có dấu
```

Integer overflow

Với giá trị số nguyên không dấu, các phép `+`, `-`, `*`, và `<<` được tính theo modulo 2*n*, với *n* là số bit của kiểu số nguyên không dấu. Nói chung, các phép toán này loại bỏ bit cao khi tràn, và chương trình có thể dựa vào "quay vòng".

Với số nguyên có dấu, các phép `+`, `-`, `*`, `/`, và `<<` có thể tràn hợp lệ và giá trị kết quả tồn tại, xác định bởi biểu diễn số nguyên có dấu, phép toán và toán hạng. Tràn không gây panic tại runtime. Trình biên dịch không được tối ưu hóa dựa trên giả định không có tràn. Ví dụ, không được giả định `x < x + 1` luôn đúng.

Floating-point operators

Với số thực và số phức, `+x` giống `x`, còn `-x` là phủ định của `x`. Kết quả phép chia số thực hoặc số phức cho 0 không được chỉ định ngoài chuẩn IEEE 754; việc có panic tại runtime hay không phụ thuộc vào triển khai.

Triển khai có thể kết hợp nhiều phép toán số thực thành một phép hợp nhất, có thể qua nhiều câu lệnh, và cho kết quả khác với việc thực hiện và làm tròn từng phép toán riêng lẻ. Chuyển đổi kiểu số thực tường minh sẽ làm tròn về độ chính xác của kiểu đích, ngăn hợp nhất bỏ qua việc làm tròn đó.

Ví dụ, một số kiến trúc có lệnh "fused multiply and add" (FMA) tính `x*y + z` mà không làm tròn kết quả trung gian `x*y`. Các ví dụ sau cho thấy khi nào Go có thể dùng lệnh này:

```
// FMA được phép cho r, vì x*y không làm tròn tường
minh:
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 không được phép cho r, vì sẽ bỏ qua làm tròn
x*y:
r  = float64(x*y) + z
r  = z; r += float64(x*y)
t  = float64(x*y); r = t + z
```

String concatenation

Chuỗi có thể nối bằng toán tử `+` hoặc toán tử gán `+=`:

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

Cộng chuỗi tạo chuỗi mới bằng cách nối các toán hạng.

Comparison operators

Toán tử so sánh so sánh hai toán hạng và trả về giá trị boolean không kiểu.

```
==    bằng
!=    không bằng
<     nhỏ hơn
```

```
<=    nhỏ hơn hoặc bằng
>     lớn hơn
>=    lớn hơn hoặc bằng
```

Trong mọi phép so sánh, toán hạng đầu phải có thể gán cho kiểu của toán hạng thứ hai, hoặc ngược lại.

Toán tử bằng `==` và không bằng `!=` áp dụng cho toán hạng kiểu *so sánh được*. Toán tử thứ tự `<`, `<=`, `>`, và `>=` áp dụng cho toán hạng kiểu *có thứ tự*. Các thuật ngữ này và kết quả so sánh được định nghĩa như sau:

- Kiểu boolean so sánh được. Hai giá trị boolean bằng nhau nếu đều `true` hoặc đều `false`.
- Kiểu số nguyên so sánh và có thứ tự. Hai giá trị số nguyên so sánh như thông thường.
- Kiểu số thực so sánh và có thứ tự. Hai giá trị số thực so sánh theo chuẩn IEEE 754.
- Kiểu phức so sánh được. Hai giá trị phức `u` và `v` bằng nhau nếu `real(u) == real(v)` và `imag(u) == imag(v)`.
- Kiểu chuỗi so sánh và có thứ tự. Hai giá trị chuỗi so sánh theo thứ tự byte.
- Kiểu con trỏ so sánh được. Hai giá trị con trỏ bằng nhau nếu trỏ đến cùng biến hoặc đều `nil`. Con trỏ đến biến kích thước 0 khác nhau có thể bằng hoặc không.
- Kiểu channel so sánh được. Hai giá trị channel bằng nhau nếu được tạo bởi cùng lời gọi `make` hoặc đều `nil`.
- Kiểu interface không phải tham số kiểu so sánh được. Hai giá trị interface bằng nhau nếu có kiểu động giống hệt và giá trị động bằng nhau hoặc đều `nil`.
- Giá trị `x` kiểu không phải interface `X` và giá trị `t` kiểu interface `T` có thể so sánh nếu kiểu `X` so sánh được và `X` triển khai `T`. Chúng bằng nhau nếu kiểu động của `t` giống hệt `X` và giá trị động của `t` bằng `x`.

- Kiểu struct so sánh được nếu tất cả trường của nó so sánh được. Hai giá trị struct bằng nhau nếu các trường không blank tương ứng bằng nhau. Trường được so sánh theo thứ tự khai báo, dừng khi hai trường khác nhau (hoặc đã so sánh hết).
- Kiểu array so sánh được nếu kiểu phần tử array so sánh được. Hai giá trị array bằng nhau nếu các phần tử tương ứng bằng nhau. Phần tử so sánh theo thứ tự chỉ số tăng dần, dừng khi hai phần tử khác nhau (hoặc đã so sánh hết).
- Tham số kiểu so sánh được nếu chúng so sánh nghiêm ngặt (xem dưới).

So sánh hai giá trị interface với kiểu động giống hệt gây panic tại runtime nếu kiểu đó không so sánh được. Hành vi này áp dụng cả khi so sánh trực tiếp giá trị interface và khi so sánh array interface hoặc struct có trường kiểu interface.

Kiểu slice, map, và function không so sánh được. Tuy nhiên, ngoại lệ, slice, map, hoặc function có thể so sánh với identifier định nghĩa sẵn `nil`. So sánh con trỏ, channel, và interface với `nil` cũng được phép và theo quy tắc chung trên.

```
const c = 3 < 4            // c là hằng boolean không kiểu true

type MyBool bool
var x, y int
var (
    // Kết quả so sánh là boolean không kiểu.
    // Quy tắc gán thông thường áp dụng.
    b3        = x == y // b3 kiểu bool
    b4 bool   = x == y // b4 kiểu bool
    b5 MyBool = x == y // b5 kiểu MyBool
)
```

Một kiểu *so sánh nghiêm ngặt* nếu nó so sánh được và không phải interface hoặc không chứa interface. Cụ thể:

- Kiểu boolean, số, chuỗi, con trỏ, và channel so sánh nghiêm ngặt.
- Kiểu struct so sánh nghiêm ngặt nếu tất cả trường của nó so sánh nghiêm ngặt.
- Kiểu array so sánh nghiêm ngặt nếu kiểu phần tử array so sánh nghiêm ngặt.
- Tham số kiểu so sánh nghiêm ngặt nếu mọi kiểu trong type set của nó so sánh nghiêm ngặt.

Logical operators

Toán tử logic áp dụng cho giá trị boolean và trả về kết quả cùng kiểu với toán hạng. Toán hạng trái được đánh giá trước, sau đó toán hạng phải nếu điều kiện cần.

```
&&    AND có điều kiện    p && q  là  "nếu p thì q, ngược lại false"
||    OR có điều kiện     p || q  là  "nếu p thì true, ngược lại q"
!     NOT                 !p      là  "không p"
```

Address operators

Với toán hạng `x` kiểu `T`, phép lấy địa chỉ `&x` tạo con trỏ kiểu `*T` trỏ đến `x`. Toán hạng phải *có thể lấy địa chỉ*, tức là biến, phép gián tiếp con trỏ, hoặc phép truy cập slice; hoặc selector trường của struct có thể lấy địa chỉ; hoặc phép truy cập array của array có thể lấy địa chỉ. Ngoại lệ, `x` cũng có thể là composite literal (có thể có dấu ngoặc). Nếu đánh giá `x` gây panic tại runtime, thì đánh giá `&x` cũng vậy.

Với toán hạng `x` kiểu con trỏ `*T`, phép gián tiếp con trỏ `*x` biểu diễn biến kiểu `T` mà `x` trỏ đến. Nếu `x` là `nil`, đánh giá `*x` gây panic tại runtime.

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

var x *int = nil
*x   // gây panic tại runtime
&*x  // gây panic tại runtime
```

Receive operator

Với toán hạng `ch` kiểu channel, giá trị phép nhận `<-ch` là giá trị nhận từ channel `ch`. Hướng channel phải cho phép nhận, và kiểu phép nhận là kiểu phần tử của channel. Biểu thức sẽ block cho đến khi có giá trị. Nhận từ channel `nil` sẽ block mãi mãi. Nhận từ channel đã đóng luôn thực hiện được ngay, trả về giá trị zero của kiểu phần tử sau khi đã nhận hết giá trị gửi trước đó.

```
v1 := <-ch
v2 = <-ch
f(<-ch)
<-strobe  // chờ đến xung clock và bỏ giá trị nhận
được
```

Nếu kiểu toán hạng là tham số kiểu, mọi kiểu trong type set phải là channel cho phép nhận, và tất cả phải có cùng kiểu phần tử, là kiểu của

phép nhận.

Phép nhận dùng trong câu lệnh gán hoặc khởi tạo dạng đặc biệt

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

trả về thêm giá trị boolean không kiểu báo cáo giao tiếp thành công hay không. Giá trị của `ok` là `true` nếu giá trị nhận được gửi thành công vào channel, hoặc `false` nếu là giá trị zero do channel đã đóng và rỗng.

Conversions

Chuyển đổi thay đổi kiểu của một biểu thức sang kiểu chỉ định bởi phép chuyển đổi. Chuyển đổi có thể xuất hiện tường minh trong mã nguồn, hoặc *ngầm định* bởi ngữ cảnh biểu thức xuất hiện.

Chuyển đổi *tường minh* là biểu thức dạng `T(x)` với `T` là kiểu và `x` là biểu thức có thể chuyển sang kiểu `T`.

Conversion = Type "(" Expression [","] ")" .

Nếu kiểu bắt đầu bằng toán tử `*` hoặc `<-`, hoặc bắt đầu bằng từ khóa `func` và không có danh sách kết quả, phải đặt trong ngoặc khi cần để tránh mơ hồ:

```
*Point(p)        // giống *(Point(p))
(*Point)(p)      // p chuyển sang *Point
<-chan int(c)    // giống <-(chan int(c))
(<-chan int)(c)  // c chuyển sang <-chan int
```

```
func()(x)         // chữ ký hàm func() x
(func())(x)       // x chuyển sang func()
(func() int)(x)   // x chuyển sang func() int
func() int(x)     // x chuyển sang func() int (không
mơ hồ)
```

Giá trị hằng x có thể chuyển sang kiểu T nếu x có thể biểu diễn bởi giá trị kiểu T. Ngoại lệ, hằng số nguyên x có thể chuyển tường minh sang kiểu chuỗi theo quy tắc như với x không phải hằng.

Chuyển hằng sang kiểu không phải tham số kiểu trả về hằng kiểu.

```
uint(iota)               // iota kiểu uint
float32(2.718281828)     // 2.718281828 kiểu
float32
complex128(1)            // 1.0 + 0.0i kiểu
complex128
float32(0.49999999)      // 0.5 kiểu float32
float64(-1e-1000)        // 0.0 kiểu float64
string('x')              // "x" kiểu string
string(0x266c)           // "♬" kiểu string
myString("foo" + "bar")  // "foobar" kiểu myString
string([]byte{'a'})      // không phải hằng:
[]byte{'a'} không phải hằng
(*int)(nil)              // không phải hằng: nil
không phải hằng, *int không phải boolean, số, hoặc
chuỗi
int(1.2)                 // không hợp lệ: 1.2 không
thể biểu diễn thành int
string(65.0)             // không hợp lệ: 65.0
không phải hằng số nguyên
```

Chuyển hằng sang tham số kiểu trả về giá trị *không hằng* kiểu đó, với giá trị biểu diễn là giá trị của type argument mà tham số kiểu được khởi

tạo. Ví dụ, với hàm:

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

chuyển đổi `P(1.1)` trả về giá trị không hằng kiểu `P` và giá trị `1.1` được biểu diễn là `float32` hoặc `float64` tùy type argument cho `f`. Do đó, nếu `f` được khởi tạo với kiểu `float32`, giá trị số của biểu thức `P(1.1) + 1.2` sẽ được tính với độ chính xác như phép cộng `float32` không hằng tương ứng.

Giá trị không hằng `x` có thể chuyển sang kiểu `T` trong các trường hợp sau:

- `x` có thể gán cho `T`.
- bỏ qua struct tag (xem dưới), kiểu của `x` và `T` không phải tham số kiểu nhưng có kiểu nền giống hệt.
- bỏ qua struct tag (xem dưới), kiểu của `x` và `T` là con trỏ không phải kiểu có tên, và kiểu cơ sở con trỏ không phải tham số kiểu nhưng có kiểu nền giống hệt.
- kiểu của `x` và `T` đều là kiểu số nguyên hoặc số thực.
- kiểu của `x` và `T` đều là kiểu phức.
- `x` là số nguyên hoặc slice byte hoặc rune và `T` là kiểu chuỗi.
- `x` là chuỗi và `T` là slice byte hoặc rune.
- `x` là slice, `T` là array [Go 1.20] hoặc con trỏ đến array [Go 1.17], và kiểu slice và array giống hệt kiểu phần tử.

Ngoài ra, nếu `T` hoặc kiểu `V` của `x` là tham số kiểu, `x` cũng có thể chuyển sang kiểu `T` nếu một trong các điều kiện sau đúng:

- Cả `V` và `T` là tham số kiểu và giá trị của mỗi kiểu trong type set của `V` có thể chuyển sang mỗi kiểu trong type set của `T`.

- Chỉ `V` là tham số kiểu và giá trị của mỗi kiểu trong type set của `V` có thể chuyển sang `T`.
- Chỉ `T` là tham số kiểu và `x` có thể chuyển sang mỗi kiểu trong type set của `T`.

Struct tag bị bỏ qua khi so sánh kiểu struct để chuyển đổi:

```
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)  // bỏ qua tag, kiểu
nền giống hệt
```

Quy tắc riêng áp dụng cho chuyển đổi (không hằng) giữa kiểu số hoặc sang/và từ kiểu chuỗi. Các chuyển đổi này có thể thay đổi biểu diễn của `x` và tốn chi phí runtime. Các chuyển đổi khác chỉ thay đổi kiểu, không thay đổi biểu diễn của `x`.

Không có cơ chế ngôn ngữ để chuyển đổi giữa con trỏ và số nguyên. Gói `unsafe` thực hiện chức năng này trong điều kiện hạn chế.

Conversions between numeric types

Với chuyển đổi giá trị số không hằng, áp dụng các quy tắc sau:

1. Khi chuyển giữa kiểu số nguyên, nếu giá trị là số nguyên có dấu, nó được mở rộng dấu đến độ chính xác vô hạn ngầm định; nếu không thì mở rộng bằng 0. Sau đó cắt bớt để vừa với kích thước kiểu kết quả. Ví dụ, nếu `v := uint16(0x10F0)`, thì `uint32(int8(v)) == 0xFFFFFFF0`. Chuyển đổi luôn trả về giá trị hợp lệ; không báo tràn.
2. Khi chuyển số thực sang số nguyên, phần thập phân bị loại bỏ (làm tròn về 0).
3. Khi chuyển số nguyên hoặc số thực sang kiểu số thực, hoặc số phức sang kiểu phức khác, giá trị kết quả được làm tròn về độ chính xác kiểu đích. Ví dụ, giá trị biến `x` kiểu `float32` có thể lưu với độ chính xác cao hơn IEEE 754 32 bit, nhưng float32(x) là kết quả làm tròn về 32 bit. Tương tự, `x + 0.1` có thể dùng hơn 32 bit, nhưng `float32(x + 0.1)` thì không.

Với mọi chuyển đổi không hằng liên quan đến số thực hoặc phức, nếu kiểu kết quả không biểu diễn được giá trị, chuyển đổi vẫn thành công nhưng giá trị kết quả phụ thuộc vào triển khai.

Conversions to and from a string type

1. Chuyển slice byte sang kiểu chuỗi trả về chuỗi có các byte liên tiếp là phần tử của 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. Chuyển slice rune sang kiểu chuỗi trả về chuỗi là nối các giá trị rune chuyển sang chuỗi.

```
    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. Chuyển giá trị kiểu chuỗi sang slice byte trả về slice không nil, các phần tử liên tiếp là byte của chuỗi. Dung lượng của slice kết quả phụ thuộc vào triển khai và có thể lớn hơn độ dài slice.

```
[]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. Chuyển giá trị kiểu chuỗi sang slice rune trả về slice chứa các mã Unicode của chuỗi. Dung lượng của slice kết quả phụ thuộc vào triển khai và có thể lớn hơn độ dài slice.

```
[]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. Cuối cùng, vì lý do lịch sử, giá trị số nguyên có thể chuyển sang kiểu chuỗi. Dạng chuyển đổi này trả về chuỗi chứa biểu diễn UTF-8 (có thể nhiều byte) của mã Unicode với giá trị số nguyên

đó. Giá trị ngoài phạm vi Unicode hợp lệ chuyển thành `"\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"
```

Lưu ý: Dạng chuyển đổi này có thể bị loại bỏ khỏi ngôn ngữ trong tương lai. Công cụ `go vet` cảnh báo một số chuyển đổi số nguyên sang chuỗi là lỗi tiềm ẩn. Nên dùng hàm thư viện như `utf8.AppendRune` hoặc `utf8.EncodeRune`.

Conversions from slice to array or array pointer

Chuyển slice sang array trả về array chứa các phần tử của mảng nền của slice. Tương tự, chuyển slice sang con trỏ array trả về con trỏ đến mảng nền của slice. Trong cả hai trường hợp, nếu độ dài của slice nhỏ hơn độ dài array, panic tại runtime.

```
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)         // panic: 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)      // panic: len([4]byte) >
len(s)

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

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

Constant expressions

Biểu thức hằng chỉ chứa hằng làm toán hạng và được đánh giá tại thời gian biên dịch.

Hằng boolean, số, và chuỗi không kiểu có thể dùng làm toán hạng ở bất kỳ đâu hợp lệ với toán hạng kiểu boolean, số, hoặc chuỗi tương ứng.

So sánh hằng luôn trả về hằng boolean không kiểu. Nếu toán hạng trái của biểu thức shift hằng là hằng không kiểu, kết quả là hằng số nguyên; nếu không, là hằng kiểu của toán hạng trái, phải là kiểu số nguyên.

Mọi phép toán khác trên hằng không kiểu trả về hằng không kiểu cùng loại; tức là, hằng boolean, số nguyên, số thực, số phức, hoặc chuỗi. Nếu toán hạng không kiểu của phép toán hai ngôi (trừ shift) khác loại, kết quả là loại của toán hạng xuất hiện sau trong danh sách: số nguyên, rune, số thực, phức. Ví dụ, hằng số nguyên không kiểu chia cho hằng số phức không kiểu trả về hằng số phức không kiểu.

```
const a = 2 + 3.0          // a == 5.0   (hằng số
thực không kiểu)
const b = 15 / 4           // b == 3     (hằng số
nguyên không kiểu)
const c = 15 / 4.0         // c == 3.75  (hằng số
thực không kiểu)
const Θ float64 = 3/2      // Θ == 1.0   (kiểu
float64, 3/2 là chia số nguyên)
const Π float64 = 3/2.     // Π == 1.5   (kiểu
float64, 3/2. là chia số thực)
const d = 1 << 3.0         // d == 8     (hằng số
nguyên không kiểu)
const e = 1.0 << 3         // e == 8     (hằng số
nguyên không kiểu)
const f = int32(1) << 33   // không hợp lệ    (hằng
8589934592 tràn int32)
const g = float64(2) >> 1  // không hợp lệ
(float64(2) là hằng số thực kiểu)
const h = "foo" > "bar"    // h == true  (hằng
boolean không kiểu)
const j = true             // j == true  (hằng
boolean không kiểu)
const k = 'w' + 1          // k == 'x'   (hằng rune
không kiểu)
const l = "hi"             // l == "hi"  (hằng
chuỗi không kiểu)
const m = string(k)        // m == "x"   (kiểu
string)
const Σ = 1 - 0.707i       //            (hằng số
phức không kiểu)
const Δ = Σ + 2.0e-4       //            (hằng số
phức không kiểu)
const Φ = iota*1i - 1/1i   //            (hằng số
phức không kiểu)
```

Áp dụng hàm dựng sẵn `complex` cho hằng số nguyên, rune, hoặc số thực không kiểu trả về hằng số phức không kiểu.

```
const ic = complex(0, c)   // ic == 3.75i  (hằng số
phức không kiểu)
const i0 = complex(0, 0)   // i0 == 1i     (kiểu
complex128)
```

Biểu thức hằng luôn được đánh giá chính xác; giá trị trung gian và hằng có thể cần độ chính xác lớn hơn bất kỳ kiểu định nghĩa sẵn nào trong ngôn ngữ. Các khai báo sau hợp lệ:

```
const Huge = 1 << 100         // Huge ==
1267650600228229401496703205376  (hằng số nguyên
không kiểu)
const Four int8 = Huge >> 98  // Four == 4
(kiểu int8)
```

Số chia của phép chia hoặc lấy dư hằng không được là 0:

```
3.14 / 0.0   // không hợp lệ: chia cho 0
```

Giá trị của hằng *có kiểu* phải luôn có thể biểu diễn chính xác bởi giá trị kiểu hằng. Các biểu thức hằng sau không hợp lệ:

```
uint(-1)     // -1 không thể biểu diễn thành uint
int(3.14)    // 3.14 không thể biểu diễn thành int
int64(Huge)  // 1267650600228229401496703205376
không thể biểu diễn thành int64
```

```
Four * 300   // toán hạng 300 không thể biểu diễn
thành int8 (kiểu của Four)
Four * 100   // tích 400 không thể biểu diễn thành
int8 (kiểu của Four)
```

Mask dùng cho toán tử đảo bit một ngôi ^ giống quy tắc với không hằng: mask là tất cả 1 với hằng không dấu và -1 với hằng có dấu và không kiểu.

```
^1         // hằng số nguyên không kiểu, bằng -2
uint8(^1)  // không hợp lệ: giống uint8(-2), -2
không thể biểu diễn thành uint8
^uint8(1)  // hằng uint8, giống 0xFF ^ uint8(1) =
uint8(0xFE)
int8(^1)   // giống int8(-2)
^int8(1)   // giống -1 ^ int8(1) = -2
```

Giới hạn triển khai: Trình biên dịch có thể làm tròn khi tính biểu thức hằng số thực hoặc phức không kiểu; xem giới hạn trong phần hằng. Việc làm tròn này có thể khiến biểu thức hằng số thực không hợp lệ trong ngữ cảnh số nguyên, dù sẽ là số nguyên nếu tính với độ chính xác vô hạn, và ngược lại.

Order of evaluation

Ở cấp package, phụ thuộc khởi tạo xác định thứ tự đánh giá các biểu thức khởi tạo riêng lẻ trong khai báo biến. Ngoài ra, khi đánh giá toán hạng của biểu thức, phép gán, hoặc câu lệnh return, mọi lời gọi hàm, gọi method, phép nhận, và phép logic hai ngôi được đánh giá theo thứ tự từ trái sang phải trong mã nguồn.

Ví dụ, trong phép gán (cục bộ hàm)

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

các lời gọi hàm và giao tiếp diễn ra theo thứ tự `f()`, `h()` (nếu `z` là false), `i()`, `j()`, `<-c`, `g()`, và `k()`. Tuy nhiên, thứ tự các sự kiện này so với đánh giá và truy cập chỉ số của `x` và đánh giá `y` và `z` không được chỉ định, trừ khi cần theo thứ tự mã nguồn. Ví dụ, `g` không được gọi trước khi các đối số của nó được đánh giá.

```
a := 1
f := func() int { a++; return a }
x := []int{a, f()}            // x có thể là [1, 2]
hoặc [2, 2]: thứ tự đánh giá giữa a và f() không
xác định
m := map[int]int{a: 1, a: 2}  // m có thể là {2: 1}
hoặc {2: 2}: thứ tự đánh giá hai phép gán map không
xác định
n := map[int]int{a: f()}      // n có thể là {2: 3}
hoặc {3: 3}: thứ tự đánh giá key và value không xác
định
```

Ở cấp package, phụ thuộc khởi tạo ghi đè quy tắc trái sang phải cho các biểu thức khởi tạo riêng lẻ, nhưng không cho toán hạng trong mỗi biểu thức:

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

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

// các hàm u và v không phụ thuộc vào biến và hàm khác

Các lời gọi hàm diễn ra theo thứ tự `u()`, `sqr()`, `v()`, `f()`, `v()`, và `g()`.

Các phép toán số thực trong một biểu thức được đánh giá theo tính kết hợp của toán tử. Ngoặc đơn tường minh ảnh hưởng đến đánh giá bằng cách ghi đè tính kết hợp mặc định. Trong biểu thức `x + (y + z)` phép cộng `y + z` được thực hiện trước khi cộng với `x`.

Statements

Các câu lệnh điều khiển luồng thực thi.

```
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

Một *terminating statement* (câu lệnh kết thúc) sẽ ngắt luồng điều khiển thông thường trong một block. Các câu lệnh sau đây được xem là terminating:

1. Một câu lệnh "return" hoặc "goto".
2. Một lời gọi hàm dựng sẵn `panic`.
3. Một block mà danh sách câu lệnh kết thúc bằng một terminating statement.

4. Một câu lệnh "if" mà:
    - nhánh "else" tồn tại, và
    - cả hai nhánh đều là terminating statement.
5. Một câu lệnh "for" mà:
    - không có câu lệnh "break" nào tham chiếu đến vòng lặp "for" đó,
    - điều kiện vòng lặp không có,
    - và vòng lặp "for" không sử dụng range clause.
6. Một câu lệnh "switch" mà:
    - không có câu lệnh "break" nào tham chiếu đến "switch" đó,
    - có case mặc định (default case),
    - danh sách câu lệnh trong mỗi case, bao gồm cả default, đều kết thúc bằng một terminating statement, hoặc một "fallthrough" statement (có thể có nhãn).
7. Một câu lệnh "select" mà:
    - không có câu lệnh "break" nào tham chiếu đến "select" đó,
    - danh sách câu lệnh trong mỗi case, bao gồm cả default nếu có, đều kết thúc bằng một terminating statement.
8. Một labeled statement gán nhãn cho một terminating statement.

Tất cả các câu lệnh khác không phải là terminating statement.

Một statement list kết thúc bằng một terminating statement nếu danh sách không rỗng và câu lệnh không rỗng cuối cùng là terminating.

Empty statements

Empty statement (câu lệnh rỗng) không thực hiện gì cả.

```
EmptyStmt = .
```

Labeled statements

Labeled statement (câu lệnh có nhãn) có thể là đích đến của câu lệnh `goto`, `break` hoặc `continue`.

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

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

Expression statements

Ngoại trừ một số hàm dựng sẵn nhất định, các lời gọi hàm và phương thức calls và receive operations có thể xuất hiện trong ngữ cảnh câu lệnh. Các câu lệnh này có thể được đặt trong dấu ngoặc đơn.

```
ExpressionStmt = Expression .
```

Các hàm dựng sẵn sau không được phép xuất hiện trong ngữ cảnh câu lệnh:

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

Send statement (câu lệnh gửi) gửi một giá trị vào channel. Biểu thức channel phải là channel type, hướng channel phải cho phép gửi, và kiểu của giá trị gửi phải assignable cho kiểu phần tử của channel.

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

Cả channel và biểu thức giá trị đều được đánh giá trước khi bắt đầu giao tiếp. Giao tiếp sẽ block cho đến khi gửi thành công. Gửi vào channel không buffer sẽ thành công nếu có receiver sẵn sàng. Gửi vào channel có buffer sẽ thành công nếu còn chỗ trống trong buffer. Gửi vào channel đã đóng sẽ gây ra run-time panic. Gửi vào channel `nil` sẽ block mãi mãi.

```
ch <- 3 // gửi giá trị 3 vào channel ch
```

Nếu kiểu của biểu thức channel là type parameter, tất cả các kiểu trong type set phải là channel cho phép gửi, có cùng kiểu phần tử, và kiểu giá trị gửi phải assignable cho kiểu phần tử đó.

IncDec statements

Câu lệnh "++" và "--" tăng hoặc giảm toán hạng của nó lên 1 (là constant không kiểu). Tương tự như phép gán, toán hạng phải addressable hoặc là biểu thức truy cập phần tử map.

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

Các assignment statements sau là tương đương về mặt ngữ nghĩa:

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

Assignment statements

An assignment (phép gán) thay thế giá trị hiện tại lưu trong một variable bằng giá trị mới xác định bởi một expression. Một câu lệnh gán có thể gán một giá trị cho một biến, hoặc nhiều giá trị cho số biến tương ứng.

Assignment = ExpressionList assign_op ExpressionList .

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

Mỗi toán hạng bên trái phải là addressable, biểu thức truy cập phần tử map, hoặc (chỉ với phép gán =) là blank identifier. Toán hạng có thể được đặt trong dấu ngoặc đơn.

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

Một *assignment operation* x *op*= y với *op* là một arithmetic operator nhị phân tương đương với x = x *op* (y) nhưng chỉ đánh giá x một lần. Cấu trúc *op*= là một token duy nhất. Trong assignment operation, cả danh sách biểu thức bên trái và bên phải phải chứa đúng một biểu thức đơn giá trị, và biểu thức bên trái không được là blank identifier.

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

Tuple assignment gán từng phần tử của một phép toán trả về nhiều giá trị cho danh sách biến. Có hai dạng. Dạng đầu, toán hạng bên phải là một biểu thức trả về nhiều giá trị như gọi hàm, thao tác channel hoặc map, hoặc type assertion. Số toán hạng bên trái phải khớp với số giá trị trả về. Ví dụ, nếu f là hàm trả về hai giá trị,

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

gán giá trị đầu cho x và giá trị thứ hai cho y. Dạng thứ hai, số toán hạng bên trái phải bằng số biểu thức bên phải, mỗi biểu thức phải là đơn giá trị, và biểu thức thứ *n* bên phải gán cho toán hạng thứ *n* bên trái:

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

Blank identifier cho phép bỏ qua giá trị bên phải trong phép gán:

```
_ = x // đánh giá x nhưng bỏ qua
x, _ = f() // đánh giá f() nhưng bỏ qua giá trị trả
về thứ hai
```

Phép gán diễn ra theo hai pha. Đầu tiên, các toán hạng của index expressions và pointer indirections (bao gồm cả pointer ngầm định trong selectors) bên trái và các biểu thức bên phải đều được đánh giá theo thứ tự thông thường. Thứ hai, các phép gán được thực hiện từ trái sang phải.

```
a, b = b, a // hoán đổi a và b

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

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

x[0], x[0] = 1, 2 // gán x[0] = 1, sau đó x[0] = 2
(kết quả x[0] == 2)

x[1], x[3] = 4, 5 // gán x[1] = 4, sau đó panic khi
gán x[3] = 5.

type Point struct { x, y int }
var p *Point
x[2], p.x = 6, 7 // gán x[2] = 6, sau đó panic khi
gán p.x = 7

i = 2
x = []int{3, 5, 7}
for i, x[i] = range x { // gán i, x[2] = 0, x[0]
break
}
// sau vòng lặp này, i == 0 và x là []int{3, 5, 3}
```

Trong phép gán, mỗi giá trị phải assignable cho kiểu của toán hạng nhận giá trị, với các trường hợp đặc biệt sau:

1. Bất kỳ giá trị có kiểu nào cũng có thể gán cho blank identifier.
2. Nếu một hằng số không kiểu được gán cho biến kiểu interface hoặc blank identifier, hằng số đó sẽ được chuyển đổi ngầm định sang kiểu mặc định.
3. Nếu một giá trị boolean không kiểu được gán cho biến kiểu interface hoặc blank identifier, nó sẽ được chuyển đổi ngầm định sang kiểu `bool`.

Khi một giá trị được gán cho biến, chỉ dữ liệu lưu trong biến đó được thay thế. Nếu giá trị chứa reference, phép gán chỉ sao chép reference chứ không sao chép dữ liệu tham chiếu (như mảng nền của slice).

```
var s1 = []int{1, 2, 3}
var s2 = s1 // s2 lưu descriptor của s1
s1 = s1[:1] // độ dài s1 là 1 nhưng vẫn dùng chung
mảng nền với s2
s2[0] = 42 // thay đổi s2[0] cũng thay đổi s1[0]
fmt.Println(s1, s2) // in ra [42] [42 2 3]

var m1 = make(map[string]int)
var m2 = m1 // m2 lưu descriptor của m1
m1["foo"] = 42 // thay đổi m1["foo"] cũng thay đổi
m2["foo"]
fmt.Println(m2["foo"]) // in ra 42
```

If statements

Câu lệnh "if" xác định việc thực thi có điều kiện hai nhánh dựa trên giá trị của một biểu thức boolean. Nếu biểu thức là true, nhánh "if" được thực thi, ngược lại, nếu có, nhánh "else" được thực thi.

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

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

Biểu thức có thể được đặt trước bởi một simple statement, thực thi trước khi đánh giá biểu thức.

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

Switch statements

Câu lệnh "switch" cung cấp thực thi đa nhánh. Một biểu thức hoặc kiểu được so sánh với các "case" trong "switch" để xác định nhánh nào sẽ thực thi.

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

Có hai dạng: expression switch và type switch. Trong expression switch, các case chứa biểu thức được so sánh với giá trị của biểu thức switch. Trong type switch, các case chứa kiểu được so sánh với kiểu động của

biểu thức switch đặc biệt. Biểu thức switch chỉ được đánh giá một lần trong câu lệnh switch.

Expression switches

Trong expression switch, biểu thức switch được đánh giá và các biểu thức case (không nhất thiết phải là hằng số) được đánh giá từ trái sang phải, từ trên xuống dưới; case đầu tiên bằng với biểu thức switch sẽ thực thi các câu lệnh liên quan; các case khác bị bỏ qua. Nếu không case nào khớp và có case "default", các câu lệnh của nó sẽ được thực thi. Chỉ được có tối đa một default case và nó có thể xuất hiện ở bất kỳ đâu trong "switch". Nếu thiếu biểu thức switch, nó tương đương với giá trị boolean `true`.

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

Nếu biểu thức switch là hằng số không kiểu, nó sẽ được chuyển đổi ngầm định sang kiểu mặc định. Giá trị không kiểu `nil` không thể dùng làm biểu thức switch. Kiểu của biểu thức switch phải comparable.

Nếu biểu thức case là không kiểu, nó sẽ được chuyển đổi ngầm định sang kiểu của biểu thức switch. Với mỗi biểu thức case `x` (có thể đã chuyển đổi) và giá trị `t` của biểu thức switch, `x == t` phải là một phép so sánh hợp lệ.

Nói cách khác, biểu thức switch được xem như khai báo và khởi tạo một biến tạm `t` không khai báo kiểu; giá trị của `t` sẽ được so sánh với từng biểu thức case `x`.

Trong một case hoặc default clause, câu lệnh không rỗng cuối cùng có thể là (có thể labeled) "fallthrough" statement để chuyển điều khiển sang câu lệnh đầu tiên của clause tiếp theo. Nếu không, điều khiển sẽ chuyển đến cuối câu lệnh "switch". "fallthrough" chỉ được phép là câu lệnh cuối cùng của tất cả các clause trừ clause cuối cùng.

Biểu thức switch có thể được đặt trước bởi một simple statement, thực thi trước khi đánh giá biểu thức.

```
switch tag {
default: s3()
case 0, 1, 2, 3: s1()
case 4, 5, 6, 7: s2()
}

switch x := f(); { // thiếu biểu thức switch, tương đương "true"
case x < 0: return -x
default: return x
}

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

Giới hạn hiện thực: Compiler có thể không cho phép nhiều biểu thức case có giá trị hằng số giống nhau. Ví dụ, các compiler hiện tại không cho phép trùng lặp hằng số số nguyên, số thực, hoặc chuỗi trong case.

Type switches

Type switch so sánh kiểu thay vì giá trị. Cách hoạt động tương tự expression switch. Nó được đánh dấu bởi biểu thức switch đặc biệt có dạng type assertion sử dụng từ khóa `type` thay vì kiểu thực tế:

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

Các case sẽ so khớp kiểu thực tế `T` với kiểu động của biểu thức `x`. Tương tự type assertion, `x` phải là interface type, nhưng không phải type parameter, và mỗi kiểu không phải interface `T` trong case phải implement kiểu của `x`. Các kiểu trong case của type switch phải khác nhau.

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

TypeSwitchGuard có thể bao gồm short variable declaration. Khi dùng dạng này, biến sẽ được khai báo ở cuối TypeSwitchCase trong implicit block của mỗi clause. Với clause chỉ liệt kê một kiểu, biến sẽ có kiểu đó; nếu không, biến sẽ có kiểu của biểu thức trong TypeSwitchGuard.

Thay vì kiểu, một case có thể dùng identifier `nil`; case này được chọn khi biểu thức trong TypeSwitchGuard là giá trị interface `nil`. Chỉ được có tối đa một case `nil`.

Với biểu thức `x` kiểu `interface{}`, type switch sau:

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

có thể viết lại thành:

```
v := x // x chỉ được đánh giá một lần
if v == nil {
i := v // kiểu của i là kiểu của x (interface{})
printString("x is nil")
} else if i, isInt := v.(int); isInt {
printInt(i) // kiểu của i là int
} else if i, isFloat64 := v.(float64); isFloat64 {
printFloat64(i) // kiểu của i là float64
} else if i, isFunc := v.(func(int) float64);
isFunc {
printFunction(i) // kiểu của i là func(int) float64
} else {
_, isBool := v.(bool)
_, isString := v.(string)
```

```
if isBool || isString {
i := v // kiểu của i là kiểu của x (interface{})
printString("type is bool or string")
} else {
i := v // kiểu của i là kiểu của x (interface{})
printString("don't know the type")
}
}
```

Type parameter hoặc generic type có thể dùng làm kiểu trong case. Nếu khi instantiation kiểu đó trùng với một entry khác trong switch, case khớp đầu tiên sẽ được chọn.

```
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
```

Type switch guard có thể được đặt trước bởi một simple statement, thực thi trước khi đánh giá guard.

Câu lệnh "fallthrough" không được phép trong type switch.

For statements

Câu lệnh "for" xác định việc lặp lại thực thi một block. Có ba dạng: lặp theo điều kiện, theo "for" clause, hoặc theo "range" clause.

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

For statements with single condition

Dạng đơn giản nhất, câu lệnh "for" lặp lại thực thi block miễn là điều kiện boolean là true. Điều kiện được đánh giá trước mỗi vòng lặp. Nếu không có điều kiện, tương đương với giá trị boolean `true`.

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

For statements with `for` clause

Câu lệnh "for" với ForClause cũng được điều khiển bởi điều kiện, nhưng có thể chỉ định thêm câu lệnh *init* và *post*, như phép gán, tăng/giảm. Init có thể là short variable declaration, nhưng post thì không.

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

```
PostStmt = SimpleStmt .

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

Nếu không rỗng, init được thực thi một lần trước khi đánh giá điều kiện lần đầu; post được thực thi sau mỗi lần thực thi block (và chỉ khi block đã thực thi). Bất kỳ thành phần nào của ForClause có thể rỗng nhưng dấu chấm phẩy semicolons là bắt buộc trừ khi chỉ có điều kiện. Nếu không có điều kiện, tương đương với boolean `true`.

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

Mỗi vòng lặp có biến khai báo riêng [Go 1.22]. Biến dùng cho vòng lặp đầu tiên được khai báo bởi init. Biến dùng cho các vòng lặp tiếp theo được khai báo ngầm định trước khi thực thi post và khởi tạo bằng giá trị của biến vòng trước tại thời điểm đó.

```
var prints []func()
for i := 0; i < 5; i++ {
prints = append(prints, func() { println(i) })
i++
}
for _, p := range prints {
p()
}
```

in ra

```
1
3
5
```

Trước [Go 1.22], các vòng lặp dùng chung một biến thay vì biến riêng. Khi đó, ví dụ trên in ra

```
6
6
6
```

For statements with `range` clause

Câu lệnh "for" với "range" clause lặp qua tất cả phần tử của array, slice, string, map, giá trị nhận từ channel, giá trị nguyên từ 0 đến giới hạn trên [Go 1.22], hoặc giá trị trả về từ hàm iterator [Go 1.23]. Mỗi phần tử, các *iteration values* được gán cho *iteration variables* tương ứng (nếu có) rồi thực thi block.

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

Biểu thức bên phải trong "range" gọi là *range expression*, có thể là array, pointer to array, slice, string, map, channel cho phép receive operations, số nguyên, hoặc hàm với signature cụ thể (xem dưới). Tương tự phép gán, nếu có, toán hạng bên trái phải addressable hoặc là truy cập phần tử map; chúng là iteration variables. Nếu range expression là hàm, số lượng iteration variable tối đa phụ thuộc signature. Nếu là channel hoặc

số nguyên, chỉ được tối đa một iteration variable; nếu không, tối đa hai. Nếu iteration variable cuối là blank identifier, range clause tương đương với clause không có identifier đó.

Range expression `x` được đánh giá trước khi bắt đầu vòng lặp, ngoại trừ: nếu chỉ có tối đa một iteration variable và `x` hoặc `len(x)` là constant, range expression không được đánh giá.

Các lời gọi hàm bên trái được đánh giá mỗi vòng lặp. Mỗi vòng lặp, iteration values được tạo ra như sau nếu có iteration variable tương ứng:

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
integer value n integer type, or untyped int value
i see below
function, 0 values f func(func() bool)
function, 1 value f func(func(V) bool) value v V
function, 2 values f func(func(K, V) bool) key k K
v V
```

1. Với array, pointer to array, hoặc slice `a`, giá trị index được tạo theo thứ tự tăng dần, bắt đầu từ 0. Nếu chỉ có tối đa một iteration variable, vòng lặp tạo giá trị từ 0 đến `len(a)-1` và không truy cập vào array hoặc slice. Nếu slice là `nil`, số vòng lặp là 0.
2. Với string, "range" lặp qua các code point Unicode trong chuỗi bắt đầu từ byte index 0. Mỗi vòng lặp, index là vị trí byte đầu của

code point UTF-8 tiếp theo, giá trị thứ hai (kiểu `rune`) là giá trị code point. Nếu gặp chuỗi UTF-8 không hợp lệ, giá trị thứ hai là `0xFFFD` (Unicode replacement character), vòng lặp tiếp theo sẽ tăng một byte.

3. Thứ tự lặp qua map không xác định và không đảm bảo giống nhau giữa các lần lặp. Nếu một entry chưa lặp bị xóa trong quá trình lặp, giá trị đó sẽ không được tạo. Nếu một entry được tạo trong quá trình lặp, entry đó có thể được lặp hoặc bị bỏ qua. Nếu map là `nil`, số vòng lặp là 0.
4. Với channel, giá trị lặp là các giá trị gửi vào channel cho đến khi channel đóng. Nếu channel là `nil`, range expression sẽ block mãi mãi.
5. Với số nguyên `n`, là integer type hoặc integer constant không kiểu, giá trị lặp từ 0 đến `n-1` theo thứ tự tăng dần. Nếu `n` là integer type, giá trị lặp có cùng kiểu. Nếu không, kiểu của `n` được xác định như khi gán cho iteration variable. Cụ thể: nếu iteration variable đã tồn tại, kiểu giá trị lặp là kiểu của iteration variable (phải là integer type). Nếu không, nếu iteration variable được khai báo bởi "range" clause hoặc không có, kiểu giá trị lặp là default type của `n`. Nếu `n` <= 0, vòng lặp không chạy.
6. Với hàm `f`, lặp bằng cách gọi `f` với một hàm `yield` mới tạo làm đối số. Nếu `yield` được gọi trước khi `f` trả về, các đối số truyền vào `yield` trở thành iteration values cho một lần thực thi thân vòng lặp. Sau mỗi vòng lặp, `yield` trả về true và có thể được gọi tiếp để tiếp tục lặp. Nếu thân vòng lặp không kết thúc, "range" clause tiếp tục tạo giá trị lặp cho mỗi lần gọi `yield` cho đến khi `f` trả về. Nếu thân vòng lặp kết thúc (ví dụ bởi `break`), `yield` trả về false và không được gọi lại.

Iteration variable có thể được khai báo bởi "range" clause dùng short variable declaration (`:=`). Khi đó, scope là block của câu lệnh "for" và mỗi vòng lặp có biến mới riêng [Go 1.22] (xem "for" statements with a ForClause). Biến có kiểu của iteration value tương ứng.

Nếu iteration variable không được khai báo bởi "range" clause, chúng phải tồn tại trước. Khi đó, giá trị lặp được gán cho biến tương ứng như trong assignment statement.

```
var testdata *struct {
a *[7]int
}
for i, _ := range testdata.a {
// testdata.a không được đánh giá; len(testdata.a)
là hằng số
// i chạy từ 0 đến 6
f(i)
}

var a [10]string
for i, s := range a {
// kiểu của i là int
// kiểu của s là string
// s == a[i]
g(i, s)
}

var key string
var val interface{} // kiểu phần tử của m
assignable cho 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 == key cuối cùng lặp qua trong map
// val == map[key]

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

```
}

// làm rỗng channel
for range ch {}

// gọi f(0), f(1), ... f(9)
for i := range 10 {
// kiểu của i là int (kiểu mặc định cho hằng số
không kiểu 10)
f(i)
}

// không hợp lệ: 256 không thể gán cho uint8
var u uint8
for u = range 256 {
}

// không hợp lệ: 1e3 là hằng số số thực
for range 1e3 {
}

// fibo sinh dãy Fibonacci
fibo := func(yield func(x int) bool) {
f0, f1 := 0, 1
for yield(f0) {
f0, f1 = f1, f0+f1
}
}

// in các số Fibonacci nhỏ hơn 1000:
for x := range fibo {
if x >= 1000 {
break
}
fmt.Printf("%d ", x)
}
// output: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
610 987

// hỗ trợ lặp cho cấu trúc cây đệ quy
type Tree[K cmp.Ordered, V any] struct {
left, right *Tree[K, V]
key K
value V
}

func (t *Tree[K, V]) walk(yield func(key K, val V)
bool) bool {
return t == nil || t.left.walk(yield) &&
yield(t.key, t.value) && t.right.walk(yield)
}

func (t *Tree[K, V]) Walk(yield func(key K, val V)
bool) {
t.walk(yield)
}

// lặp cây t theo thứ tự in-order
var t Tree[string, int]
for k, v := range t.Walk {
// xử lý k, v
}
```

Nếu kiểu của range expression là type parameter, tất cả các kiểu trong type set phải có cùng underlying type và range expression phải hợp lệ cho kiểu đó, hoặc nếu type set chứa channel, chỉ được chứa channel với cùng kiểu phần tử, và tất cả channel phải cho phép nhận.

Go statements

Câu lệnh "go" khởi chạy thực thi một lời gọi hàm như một luồng điều khiển đồng thời độc lập (*goroutine*) trong cùng không gian địa chỉ.

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

Biểu thức phải là lời gọi hàm hoặc phương thức; không được đặt trong dấu ngoặc đơn. Lời gọi hàm dựng sẵn bị giới hạn như expression statements.

Giá trị hàm và tham số được đánh giá như thường lệ trong goroutine gọi, nhưng khác với lời gọi thông thường, chương trình không chờ hàm hoàn thành. Thay vào đó, hàm bắt đầu thực thi độc lập trong goroutine mới. Khi hàm kết thúc, goroutine cũng kết thúc. Nếu hàm có giá trị trả về, chúng sẽ bị bỏ qua khi hàm hoàn thành.

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

Select statements

Câu lệnh "select" chọn một trong các send hoặc receive có thể thực hiện. Nó giống với "switch" nhưng các case đều liên quan đến giao tiếp.

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

Case với RecvStmt có thể gán kết quả của RecvExpr cho một hoặc hai biến, có thể khai báo bằng short variable declaration. RecvExpr phải là (có thể đặt trong ngoặc) phép nhận giá trị từ channel. Chỉ được có tối đa một default case và nó có thể xuất hiện ở bất kỳ đâu trong danh sách case.

Thực thi "select" diễn ra theo các bước:

1. Với tất cả case, toán hạng channel của phép nhận và channel cùng biểu thức bên phải của phép gửi được đánh giá đúng một lần, theo thứ tự nguồn, khi vào "select". Kết quả là tập các channel để nhận/gửi và giá trị gửi tương ứng. Mọi side effect trong đánh giá này sẽ xảy ra bất kể giao tiếp nào được chọn. Biểu thức bên trái của RecvStmt với khai báo/gán biến chưa được đánh giá.
2. Nếu có một hoặc nhiều giao tiếp có thể thực hiện, một giao tiếp được chọn ngẫu nhiên đều. Nếu không, nếu có default case, case đó được chọn. Nếu không có default, "select" sẽ block cho đến khi có giao tiếp thực hiện được.
3. Trừ khi case được chọn là default, giao tiếp tương ứng sẽ được thực hiện.
4. Nếu case được chọn là RecvStmt với khai báo/gán biến, biểu thức bên trái được đánh giá và giá trị nhận được gán cho biến.
5. Danh sách câu lệnh của case được chọn sẽ được thực thi.

Vì giao tiếp trên channel `nil` không bao giờ thực hiện được, select chỉ có channel `nil` và không có default sẽ block mãi mãi.

```
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): // tương đương: i3, ok := <-
c3
if ok {
print("received ", i3, " from c3\n")
} else {
print("c3 is closed\n")
}
case a[f()] = <-c4:
// tương đương:
// case t := <-c4
// a[f()] = t
default:
print("no communication\n")
}

for { // gửi chuỗi bit ngẫu nhiên vào c
select {
case c <- 0: // lưu ý: không có statement, không
fallthrough, không gộp case
case c <- 1:
}
}

select {} // block mãi mãi
```

Return statements

Câu lệnh "return" trong hàm `F` kết thúc thực thi của `F`, và tùy chọn trả về một hoặc nhiều giá trị. Mọi hàm deferred bởi `F` sẽ được thực thi trước khi `F` trả về cho caller.

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

Trong hàm không có kiểu trả về, "return" không được chỉ định giá trị trả về.

```
func noResult() {
return
}
```

Có ba cách trả về giá trị từ hàm có kiểu trả về:

1. Giá trị trả về có thể được liệt kê rõ ràng trong "return". Mỗi biểu thức phải là đơn giá trị và assignable cho phần tử tương ứng của kiểu trả về.

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

    func complexF1() (re float64, im float64) {
    return -7.0, -4.0
    }
```

2. Danh sách biểu thức trong "return" có thể là một lời gọi hàm trả về nhiều giá trị. Hiệu ứng như thể mỗi giá trị trả về được gán cho biến tạm có kiểu tương ứng, sau đó "return" liệt kê các biến này, áp dụng quy tắc như trường hợp trước.

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

3. Danh sách biểu thức có thể rỗng nếu kiểu trả về của hàm chỉ định tên cho result parameters. Các result parameter hoạt động như biến cục bộ thông thường và hàm có thể gán giá trị cho chúng. "return" trả về giá trị của các biến này.

```
    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
    }
```

Bất kể khai báo thế nào, tất cả giá trị trả về được khởi tạo bằng zero values cho kiểu của chúng khi vào hàm. "return" chỉ định giá trị sẽ gán cho result parameter trước khi thực thi hàm deferred.

Giới hạn hiện thực: Compiler có thể không cho phép danh sách biểu thức rỗng trong "return" nếu có thực thể khác (hằng số, kiểu, biến) cùng tên với result parameter trong scope tại vị trí return.

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

Break statements

Câu lệnh "break" kết thúc thực thi của "for", "switch", hoặc "select" gần nhất trong cùng hàm.

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

Nếu có label, nó phải là label của "for", "switch", hoặc "select" bao ngoài, và đó là câu lệnh sẽ bị kết thúc.

```
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

Câu lệnh "continue" bắt đầu vòng lặp tiếp theo của "for" loop bao ngoài gần nhất bằng cách chuyển điều khiển đến cuối block vòng lặp. "for" loop phải nằm trong cùng hàm.

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

Nếu có label, nó phải là label của "for" bao ngoài, và đó là vòng lặp sẽ được tiếp tục.

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

Goto statements

Câu lệnh "goto" chuyển điều khiển đến câu lệnh có label tương ứng trong cùng hàm.

```
GotoStmt = "goto" Label .

goto Error
```

Thực thi "goto" không được làm cho biến nào vào scope mà chưa có trong scope tại vị trí goto. Ví dụ:

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

L: là sai vì nhảy đến label L bỏ qua việc tạo biến v.

Câu lệnh "goto" ngoài block không được nhảy vào label trong block đó. Ví dụ:

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

là sai vì label `L1` nằm trong block của "for" nhưng goto thì không.

Fallthrough statements

Câu lệnh "fallthrough" chuyển điều khiển đến câu lệnh đầu tiên của case tiếp theo trong expression "switch" statement. Chỉ được dùng làm câu lệnh không rỗng cuối cùng trong clause như vậy.

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

Defer statements

Câu lệnh "defer" gọi một hàm mà việc thực thi sẽ được hoãn lại cho đến khi hàm bao ngoài trả về, có thể do thực thi return statement, kết thúc function body, hoặc goroutine panicking.

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

Biểu thức phải là lời gọi hàm hoặc phương thức; không được đặt trong ngoặc đơn. Lời gọi hàm dựng sẵn bị giới hạn như expression statements.

Mỗi lần thực thi "defer", giá trị hàm và tham số được đánh giá như thường lệ và lưu lại, nhưng hàm thực tế chưa được gọi. Thay vào đó, các hàm deferred sẽ được gọi ngay trước khi hàm bao ngoài trả về, theo thứ tự ngược lại với thứ tự defer. Nghĩa là, nếu hàm bao ngoài trả về qua return statement, các hàm deferred được thực thi *sau* khi result parameter được gán bởi return nhưng *trước* khi hàm trả về cho caller. Nếu giá trị hàm deferred là `nil`, sẽ panic khi gọi hàm, không phải khi thực thi "defer".

Ví dụ, nếu hàm deferred là function literal và hàm bao ngoài có named result parameters trong scope của literal, hàm deferred có thể truy cập và thay đổi result parameter trước khi trả về. Nếu hàm deferred có giá trị trả về, chúng sẽ bị bỏ qua khi hàm hoàn thành. (Xem thêm phần handling panics.)

```
lock(l)
defer unlock(l) // unlock sẽ thực hiện trước khi
hàm bao ngoài trả về

// in ra 3 2 1 0 trước khi hàm bao ngoài trả về
for i := 0; i <= 3; i++ {
defer fmt.Print(i)
}

// f trả về 42
func f() (result int) {
defer func() {
// result được truy cập sau khi được gán 6 bởi
return
```

```
result *= 7
}()
return 6
}
```

Built-in functions

Các hàm dựng sẵn (built-in) được khai báo trước. Chúng được gọi như bất kỳ hàm nào khác nhưng một số hàm có thể nhận một kiểu (type) thay vì một biểu thức làm đối số đầu tiên.

Các hàm dựng sẵn không có kiểu Go chuẩn, vì vậy chúng chỉ có thể xuất hiện trong biểu thức gọi hàm; chúng không thể được sử dụng như giá trị hàm.

Appending to and copying slices

Các hàm dựng sẵn `append` và `copy` hỗ trợ các thao tác phổ biến với slice. Đối với cả hai hàm, kết quả không phụ thuộc vào việc bộ nhớ được tham chiếu bởi các đối số có bị chồng lấp hay không.

Hàm variadic `append` thêm không hoặc nhiều giá trị `x` vào một slice `s` kiểu `S` và trả về slice kết quả, cũng có kiểu `S`. Các giá trị `x` được truyền vào một tham số kiểu `...E` trong đó `E` là kiểu phần tử của `S` và các quy tắc truyền tham số tương ứng được áp dụng. Trường hợp đặc biệt, `append` cũng chấp nhận đối số đầu tiên có thể gán cho kiểu `[]byte` với đối số thứ hai là kiểu chuỗi (string) theo sau là `...`. Dạng này sẽ thêm các byte của chuỗi vào slice.

```
append(s S, x ...E) S // E is the element type of S
```

Nếu S là một type parameter, tất cả các kiểu trong tập kiểu của nó phải có cùng kiểu slice cơ bản []E.

Nếu dung lượng (capacity) của s không đủ lớn để chứa các giá trị bổ sung, append sẽ cấp phát một mảng cơ sở mới đủ lớn để chứa cả các phần tử hiện có và các giá trị bổ sung. Nếu không, append sẽ tái sử dụng mảng cơ sở.

```
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' }
```

Hàm copy sao chép các phần tử slice từ nguồn src sang đích dst và trả về số phần tử đã sao chép. Cả hai đối số phải có kiểu phần tử giống hệt E và phải có thể gán cho một slice kiểu []E. Số phần tử được sao chép là giá trị nhỏ nhất của len(src) và len(dst). Trường hợp đặc biệt, copy cũng chấp nhận đối số đích có thể gán cho kiểu []byte với đối số nguồn là kiểu string. Dạng này sẽ sao chép các byte từ chuỗi vào slice byte.

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

Nếu kiểu của một hoặc cả hai đối số là type parameter, tất cả các kiểu trong tập kiểu tương ứng phải có cùng kiểu slice cơ bản `[]E`.

Ví dụ:

```
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

Hàm dựng sẵn `clear` nhận một đối số kiểu map, slice, hoặc type parameter, và xóa hoặc đặt về giá trị zero tất cả các phần tử [Go 1.21].

Kiểu đối số | Kết quả

```
clear(m) map[K]T xóa tất cả các phần tử, kết quả là
map rỗng (len(m) == 0)

clear(s) []T đặt tất cả phần tử đến độ dài của `s`
về giá trị zero của T

clear(t) type parameter xem bên dưới
```

Nếu kiểu đối số của `clear` là type parameter, tất cả các kiểu trong tập kiểu của nó phải là map hoặc slice, và `clear` thực hiện thao tác tương ứng với kiểu thực tế.

Nếu map hoặc slice là `nil`, `clear` không làm gì cả.

Close

Với một channel `ch`, hàm dựng sẵn `close(ch)` ghi nhận rằng sẽ không còn giá trị nào được gửi lên channel nữa. Nếu `ch` là channel chỉ nhận, sẽ xảy ra lỗi. Gửi hoặc đóng một channel đã đóng sẽ gây ra panic lúc chạy. Đóng channel nil cũng gây ra panic lúc chạy. Sau khi gọi `close`, và sau khi tất cả các giá trị đã gửi trước đó được nhận, các thao tác nhận sẽ trả về giá trị zero cho kiểu của channel mà không bị chặn. Nhận nhiều giá trị sẽ trả về giá trị nhận được cùng với chỉ báo channel đã đóng hay chưa.

Nếu kiểu đối số của `close` là type parameter, tất cả các kiểu trong tập kiểu của nó phải là channel với cùng kiểu phần tử. Nếu bất kỳ channel nào là channel chỉ nhận, sẽ xảy ra lỗi.

Manipulating complex numbers

Ba hàm dùng để tạo và tách số phức. Hàm dựng sẵn `complex` tạo một giá trị phức từ phần thực và phần ảo kiểu số thực, trong khi `real` và `imag` tách phần thực và phần ảo của một giá trị phức.

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

Kiểu của các đối số và giá trị trả về phải tương ứng. Với `complex`, hai đối số phải cùng kiểu số thực và kiểu trả về là kiểu phức với thành phần số thực tương ứng: `complex64` cho đối số `float32`, và `complex128` cho đối số `float64`. Nếu một trong hai đối số là hằng không kiểu, nó sẽ được chuyển đổi ngầm định sang kiểu của đối số còn lại. Nếu cả hai là hằng không kiểu, chúng phải là số không phức hoặc phần ảo phải bằng 0, và giá trị trả về là hằng phức không kiểu.

Với `real` và `imag`, đối số phải là kiểu phức, và kiểu trả về là kiểu số thực tương ứng: `float32` cho đối số `complex64`, và `float64` cho đối số `complex128`. Nếu đối số là hằng không kiểu, nó phải là số, và giá trị trả về là hằng số thực không kiểu.

Hai hàm `real` và `imag` kết hợp lại tạo thành phép nghịch đảo của `complex`, nên với giá trị `z` kiểu phức `Z`, `z == Z(complex(real(z), imag(z)))`.

Nếu các toán hạng của các hàm này đều là hằng, giá trị trả về cũng là hằng.

```
var a = complex(2, -2) // complex128
const b = complex(1.0, -1.4) // hằng phức không
kiểu 1 - 1.4i
x := float32(math.Cos(math.Pi/2)) // float32
var c64 = complex(5, -x) // complex64
var s int = complex(1, 0) // hằng phức không kiểu 1
+ 0i có thể chuyển sang int
_ = complex(1, 2<<s) // không hợp lệ: 2 giả định là
kiểu số thực, không thể shift
var rl = real(c64) // float32
var im = imag(a) // float64
const c = imag(b) // hằng không kiểu -1.4
_ = imag(3 << s) // không hợp lệ: 3 giả định là
kiểu phức, không thể shift
```

Không cho phép đối số là kiểu type parameter.

Deletion of map elements

Hàm dựng sẵn `delete` xóa phần tử có khóa `k` khỏi map `m`. Giá trị `k` phải có thể gán cho kiểu khóa của `m`.

delete(m, k) // xóa phần tử m[k] khỏi map m

Nếu kiểu của `m` là type parameter, tất cả các kiểu trong tập kiểu đó phải là map, và tất cả phải có kiểu khóa giống hệt nhau.

Nếu map `m` là `nil` hoặc phần tử `m[k]` không tồn tại, `delete` không làm gì cả.

Length and capacity

Các hàm dựng sẵn `len` và `cap` nhận các đối số nhiều kiểu khác nhau và trả về kết quả kiểu `int`. Việc triển khai đảm bảo kết quả luôn vừa với kiểu `int`.

Kiểu đối số | Kết quả

```
len(s) string độ dài chuỗi tính theo byte
[n]T, *[n]T độ dài mảng (== n)
[]T độ dài slice
map[K]T số lượng khóa đã định nghĩa trong map
chan T số phần tử đang chờ trong buffer của channel
type parameter xem bên dưới

cap(s) [n]T, *[n]T độ dài mảng (== n)
[]T dung lượng slice
chan T dung lượng buffer của channel
type parameter xem bên dưới
```

Nếu kiểu đối số là type parameter `P`, lời gọi `len(e)` (hoặc `cap(e)`) phải hợp lệ với mỗi kiểu trong tập kiểu của `P`. Kết quả là độ dài (hoặc dung lượng) của đối số với kiểu tương ứng với type argument mà `P` được khởi tạo.

Dung lượng của một slice là số phần tử mà mảng cơ sở đã cấp phát đủ chỗ. Luôn đảm bảo:

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

Độ dài của slice, map hoặc channel `nil` là 0. Dung lượng của slice hoặc channel `nil` là 0.

Biểu thức `len(s)` là hằng số nếu `s` là hằng chuỗi. Biểu thức `len(s)` và `cap(s)` là hằng nếu kiểu của `s` là mảng hoặc con trỏ đến mảng và biểu thức `s` không chứa nhận giá trị từ channel hoặc gọi hàm không hằng; trong trường hợp này `s` không được đánh giá. Nếu không, các lời gọi `len` và `cap` không phải là hằng và `s` sẽ được đánh giá.

```
const (
c1 = imag(2i) // imag(2i) = 2.0 là hằng
c2 = len([10]float64{2}) // [10]float64{2} không
chứa gọi hàm
c3 = len([10]float64{c1}) // [10]float64{c1} không
chứa gọi hàm
c4 = len([10]float64{imag(2i)}) // imag(2i) là hằng
và không gọi hàm
c5 = len([10]float64{imag(z)}) // không hợp lệ:
imag(z) là gọi hàm không hằng
)
var z complex128
```

Making slices, maps and channels

Hàm dựng sẵn `make` nhận một kiểu `T`, phải là kiểu slice, map hoặc channel, hoặc type parameter, tùy chọn theo sau là danh sách biểu thức tùy theo kiểu. Nó trả về giá trị kiểu `T` (không phải `*T`). Bộ nhớ được khởi tạo như mô tả trong phần giá trị zero.

Kiểu T | Kết quả

```
make(T, n) slice slice kiểu T với độ dài n và dung
lượng n
make(T, n, m) slice slice kiểu T với độ dài n và
dung lượng m

make(T) map map kiểu T
make(T, n) map map kiểu T với không gian ban đầu
cho khoảng n phần tử

make(T) channel channel không buffer kiểu T
make(T, n) channel channel có buffer kiểu T, kích
thước buffer n

make(T, n) type parameter xem bên dưới
make(T, n, m) type parameter xem bên dưới
```

Nếu đối số đầu tiên là type parameter, tất cả các kiểu trong tập kiểu của nó phải có cùng kiểu cơ sở, phải là slice hoặc map, hoặc nếu là channel thì chỉ được là channel, tất cả phải có cùng kiểu phần tử, và hướng channel không được mâu thuẫn.

Mỗi đối số kích thước `n` và `m` phải là kiểu số nguyên, có tập kiểu chỉ chứa kiểu số nguyên, hoặc là hằng số không kiểu. Đối số kích thước là hằng phải không âm và có thể biểu diễn bởi giá trị kiểu `int`; nếu là hằng không kiểu thì được gán kiểu `int`. Nếu cả `n` và `m` đều là hằng, thì `n`

không được lớn hơn `m`. Với slice và channel, nếu `n` âm hoặc lớn hơn `m` lúc chạy, sẽ gây ra panic lúc chạy.

```
s := make([]int, 10, 100) // slice với len(s) ==
10, cap(s) == 100
s := make([]int, 1e3) // slice với len(s) == cap(s)
== 1000
s := make([]int, 1<<63) // không hợp lệ: len(s)
không thể biểu diễn bằng int
s := make([]int, 10, 0) // không hợp lệ: len(s) >
cap(s)
c := make(chan int, 10) // channel với buffer size
10
m := make(map[string]int, 100) // map với không
gian ban đầu cho khoảng 100 phần tử
```

Gọi `make` với kiểu map và size hint `n` sẽ tạo map với không gian ban đầu đủ chứa `n` phần tử. Hành vi cụ thể phụ thuộc vào triển khai.

Min and max

Các hàm dựng sẵn `min` và `max` tính giá trị nhỏ nhất—hoặc lớn nhất—trong số các đối số kiểu ordered types. Phải có ít nhất một đối số [Go 1.21].

Các quy tắc kiểu giống như toán tử: với các đối số ordered `x` và `y`, `min(x, y)` hợp lệ nếu `x + y` hợp lệ, và kiểu của `min(x, y)` là kiểu của `x + y` (tương tự với `max`). Nếu tất cả đối số là hằng, kết quả cũng là hằng.

```
var x, y int
m := min(x) // m == x
m := min(x, y) // m là giá trị nhỏ hơn giữa x và y
```

```
m := max(x, y, 10) // m là giá trị lớn hơn giữa x
và y nhưng ít nhất là 10
c := max(1, 2.0, 10) // c == 10.0 (kiểu số thực)
f := max(0, float32(x)) // kiểu của f là float32
var s []string
_ = min(s...) // không hợp lệ: không cho phép
truyền slice
t := max("", "foo", "bar") // t == "foo" (kiểu
chuỗi)
```

Với đối số số học, giả sử tất cả NaN là bằng nhau, `min` và `max` là giao hoán và kết hợp:

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

Với đối số số thực âm zero, NaN, và vô cực, áp dụng các quy tắc sau:

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

-0.0 0.0 -0.0 0.0 // âm zero nhỏ hơn zero không âm
-Inf y -Inf y // âm vô cực nhỏ hơn mọi số khác
+Inf y y +Inf // dương vô cực lớn hơn mọi số khác
NaN y NaN NaN // nếu có đối số là NaN, kết quả là
NaN
```

Với đối số chuỗi, kết quả của `min` là đối số đầu tiên có giá trị nhỏ nhất (hoặc với `max`, lớn nhất), so sánh theo thứ tự byte:

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

Allocation

Hàm dựng sẵn `new` nhận một kiểu `T`, cấp phát bộ nhớ cho một biến kiểu đó lúc chạy, và trả về giá trị kiểu `*T` trỏ đến nó. Biến được khởi tạo như mô tả trong phần giá trị zero.

```
new(T)
```

Ví dụ

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

cấp phát bộ nhớ cho biến kiểu `S`, khởi tạo nó (`a=0`, `b=0.0`), và trả về giá trị kiểu `*S` chứa địa chỉ vùng nhớ đó.

Handling panics

Hai hàm dựng sẵn, `panic` và `recover`, hỗ trợ báo cáo và xử lý panic lúc chạy và các điều kiện lỗi do chương trình định nghĩa.

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

Khi thực thi một hàm `F`, gọi tường minh `panic` hoặc panic lúc chạy sẽ kết thúc thực thi của `F`. Mọi hàm được defer bởi `F` sẽ được thực thi như bình thường. Tiếp theo, mọi hàm defer của caller của `F` sẽ chạy, và cứ thế cho đến hàm defer của hàm cấp cao nhất trong goroutine đang thực thi. Lúc đó, chương trình sẽ kết thúc và báo cáo lỗi, bao gồm giá trị đối số truyền vào `panic`. Trình tự kết thúc này gọi là *panicking*.

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

Hàm `recover` cho phép chương trình kiểm soát hành vi của goroutine đang panicking. Giả sử một hàm `G` defer một hàm `D` gọi `recover` và một panic xảy ra trong một hàm cùng goroutine với `G`. Khi thực thi các hàm defer đến `D`, giá trị trả về của `recover` sẽ là giá trị truyền vào `panic`. Nếu `D` trả về bình thường, không gây panic mới, chuỗi panicking sẽ dừng lại. Khi đó, trạng thái của các hàm gọi giữa `G` và `panic` sẽ bị loại bỏ, và chương trình tiếp tục thực thi bình thường. Mọi hàm defer của `G` trước `D` sẽ được chạy và `G` kết thúc bằng cách trả về cho caller.

Giá trị trả về của `recover` là `nil` khi goroutine không panicking hoặc `recover` không được gọi trực tiếp bởi một hàm defer. Ngược lại, nếu goroutine đang panicking và `recover` được gọi trực tiếp bởi một hàm defer, giá trị trả về của `recover` đảm bảo không phải là `nil`. Để đảm bảo điều này, gọi `panic` với giá trị interface `nil` (hoặc `nil` không kiểu) sẽ gây ra panic lúc chạy.

Hàm `protect` trong ví dụ dưới đây gọi hàm đối số `g` và bảo vệ caller khỏi panic lúc chạy do `g` gây ra.

```
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

Các triển khai hiện tại cung cấp một số hàm dựng sẵn hữu ích trong quá trình bootstrapping. Các hàm này được tài liệu hóa để đầy đủ nhưng không đảm bảo sẽ tồn tại lâu dài trong ngôn ngữ. Chúng không trả về kết quả.

Hàm | Hành vi

print | in ra tất cả đối số; định dạng phụ thuộc vào triển khai println | như print nhưng in thêm dấu cách giữa các đối số và xuống dòng ở cuối

Giới hạn triển khai: `print` và `println` không cần chấp nhận mọi kiểu đối số, nhưng phải hỗ trợ in các kiểu boolean, số, và chuỗi.

Packages

Chương trình Go được xây dựng bằng cách liên kết các *package*. Một package được tạo thành từ một hoặc nhiều file nguồn cùng khai báo hằng số, kiểu, biến và hàm thuộc về package và có thể truy cập trong

tất cả các file cùng package. Các phần tử này có thể được export và sử dụng ở package khác.

Source file organization

Mỗi file nguồn gồm một câu lệnh package xác định package mà nó thuộc về, theo sau là tập khai báo import (có thể rỗng) khai báo các package muốn sử dụng, tiếp theo là tập khai báo các hàm, kiểu, biến và hằng số (có thể rỗng).

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

Package clause

Một package clause bắt đầu mỗi file nguồn và xác định package mà file thuộc về.

PackageClause = "package" PackageName . PackageName = identifier .

PackageName không được là blank identifier.

```
package math
```

Tập các file cùng PackageName tạo thành phần hiện thực của một package. Một số triển khai có thể yêu cầu tất cả file nguồn của một package phải nằm cùng thư mục.

Import declarations

Khai báo import cho biết file nguồn chứa khai báo đó phụ thuộc vào chức năng của package *imported* (§Program initialization and execution)

và cho phép truy cập các định danh export của package đó. Import đặt tên định danh (PackageName) để truy cập và ImportPath xác định package cần import.

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

PackageName được dùng trong qualified identifiers để truy cập các định danh export của package trong file import. Nó được khai báo trong file block. Nếu PackageName bị bỏ qua, nó mặc định là định danh trong package clause của package import. Nếu có dấu chấm (`.`) thay cho tên, tất cả các định danh export của package sẽ được khai báo trong file block của file import và phải truy cập không cần qualifier.

Việc diễn giải ImportPath phụ thuộc vào triển khai nhưng thường là một chuỗi con của tên file đầy đủ của package đã biên dịch và có thể là đường dẫn tương đối đến kho package đã cài đặt.

Giới hạn triển khai: Compiler có thể giới hạn ImportPath là chuỗi không rỗng chỉ dùng các ký tự thuộc các nhóm L, M, N, P, S của Unicode (các ký tự Graphic không có khoảng trắng) và cũng có thể loại trừ các ký tự `!"#$%&'()*,:;<=>?[\]^`{|}` và ký tự thay thế Unicode U+FFFD.

Giả sử đã biên dịch một package chứa câu lệnh `package math`, export hàm `Sin`, và cài đặt package đã biên dịch vào file `"lib/math"`. Bảng sau minh họa cách truy cập `Sin` trong các file import package với các kiểu khai báo import khác nhau.

Khai báo import | Tên cục bộ của Sin

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

Khai báo import tạo quan hệ phụ thuộc giữa package import và package được import. Không hợp lệ nếu một package tự import chính nó, trực tiếp hoặc gián tiếp, hoặc import trực tiếp một package mà không sử dụng bất kỳ định danh export nào của nó. Để import package chỉ vì hiệu ứng phụ (khởi tạo), dùng blank làm tên package:

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

An example package

Dưới đây là một package Go hoàn chỉnh cài đặt thuật toán sàng số nguyên tố đồng thời.

```
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

Khi bộ nhớ được cấp phát cho một biến, thông qua khai báo hoặc gọi new, hoặc khi một giá trị mới được tạo ra, thông qua literal hợp thành hoặc gọi make, và không có khởi tạo tường minh, biến hoặc giá trị đó

sẽ nhận giá trị mặc định. Mỗi phần tử của biến hoặc giá trị như vậy được đặt về *giá trị zero* cho kiểu của nó: `false` cho boolean, `0` cho kiểu số, `""` cho chuỗi, và `nil` cho con trỏ, hàm, interface, slice, channel, và map. Việc khởi tạo này được thực hiện đệ quy, ví dụ mỗi phần tử của mảng struct sẽ có các trường được đặt về zero nếu không chỉ định giá trị.

Hai khai báo đơn giản sau là tương đương:

```
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

Trong một package, việc khởi tạo biến cấp package diễn ra từng bước, mỗi bước chọn biến sớm nhất theo *thứ tự khai báo* mà không phụ thuộc vào biến chưa khởi tạo.

Cụ thể hơn, một biến cấp package được coi là *sẵn sàng khởi tạo* nếu nó chưa được khởi tạo và hoặc không có biểu thức khởi tạo hoặc biểu thức khởi tạo của nó không phụ thuộc vào biến chưa khởi tạo. Việc khởi tạo tiến hành bằng cách lặp lại việc khởi tạo biến cấp package tiếp theo sớm nhất theo thứ tự khai báo và sẵn sàng khởi tạo, cho đến khi không còn biến nào sẵn sàng khởi tạo.

Nếu còn biến chưa khởi tạo khi quá trình này kết thúc, các biến đó thuộc một hoặc nhiều chu trình khởi tạo, và chương trình không hợp lệ.

Nhiều biến ở bên trái của khai báo biến được khởi tạo bởi một biểu thức (có thể trả về nhiều giá trị) ở bên phải sẽ được khởi tạo cùng lúc: Nếu bất kỳ biến nào ở bên trái được khởi tạo, tất cả các biến đó sẽ được khởi tạo trong cùng một bước.

```
var x = a
var a, b = f() // a và b được khởi tạo cùng lúc,
trước khi x được khởi tạo
```

Với mục đích khởi tạo package, biến blank được xử lý như các biến khác trong khai báo.

Thứ tự khai báo của biến khai báo ở nhiều file được xác định bởi thứ tự file được trình biên dịch nhận: Biến khai báo ở file đầu tiên được khai báo trước bất kỳ biến nào ở file thứ hai, v.v. Để đảm bảo hành vi khởi tạo có thể tái lập, hệ thống build nên truyền các file thuộc cùng package cho compiler theo thứ tự tên file từ điển.

Phân tích phụ thuộc không dựa vào giá trị thực tế của biến, chỉ dựa vào *tham chiếu* trong mã nguồn, phân tích chuyển tiếp. Ví dụ, nếu biểu thức khởi tạo của biến `x` tham chiếu đến một hàm mà thân hàm đó tham chiếu đến biến `y` thì `x` phụ thuộc vào `y`. Cụ thể:

- Tham chiếu đến biến hoặc hàm là một định danh chỉ biến hoặc hàm đó.
- Tham chiếu đến phương thức `m` là method value hoặc method expression dạng `t.m`, trong đó kiểu (tĩnh) của `t` không phải là interface, và phương thức `m` nằm trong method set của `t`. Việc giá trị hàm `t.m` có được gọi hay không không quan trọng.
- Một biến, hàm, hoặc phương thức `x` phụ thuộc vào biến `y` nếu biểu thức khởi tạo hoặc thân hàm (với hàm và phương thức) chứa tham chiếu đến `y` hoặc đến hàm/phương thức phụ thuộc vào `y`.

Ví dụ, với các khai báo

```
var (
a = c + b // == 9
b = f() // == 4
c = f() // == 5
d = 3 // == 5 sau khi khởi tạo xong
)

func f() int {
d++
return d
}
```

thứ tự khởi tạo là `d`, `b`, `c`, `a`. Lưu ý thứ tự các biểu thức con trong biểu thức khởi tạo không quan trọng: `a = c + b` và `a = b + c` đều cho cùng thứ tự khởi tạo trong ví dụ này.

Phân tích phụ thuộc được thực hiện theo từng package; chỉ các tham chiếu đến biến, hàm, và phương thức (không phải interface) khai báo trong package hiện tại mới được xét. Nếu có các phụ thuộc dữ liệu ẩn khác giữa các biến, thứ tự khởi tạo giữa các biến đó là không xác định.

Ví dụ, với các khai báo

```
var x = I(T{}).ab() // x có phụ thuộc ẩn, không
phát hiện được vào a và b
var _ = sideEffect() // không liên quan đến x, a,
hoặc b
var a = b
var b = 42

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

biến `a` sẽ được khởi tạo sau `b` nhưng việc `x` được khởi tạo trước `b`, giữa `b` và `a`, hay sau `a`, và do đó thời điểm gọi `sideEffect()` (trước hay sau khi `x` được khởi tạo) là không xác định.

Biến cũng có thể được khởi tạo bằng các hàm tên `init` khai báo trong package block, không có đối số và không có tham số trả về.

```
func init() { … }
```

Có thể định nghĩa nhiều hàm như vậy cho mỗi package, thậm chí trong cùng một file nguồn. Trong package block, định danh `init` chỉ dùng để khai báo hàm `init`, nhưng bản thân định danh này không được khai báo. Do đó không thể tham chiếu đến hàm `init` từ bất kỳ đâu trong chương trình.

Toàn bộ package được khởi tạo bằng cách gán giá trị khởi tạo cho tất cả biến cấp package, sau đó gọi tất cả hàm `init` theo thứ tự xuất hiện trong mã nguồn, có thể ở nhiều file, như được trình biên dịch nhận.

Program initialization

Các package của một chương trình hoàn chỉnh được khởi tạo từng bước, mỗi lần một package. Nếu một package có import, các package được import sẽ được khởi tạo trước khi khởi tạo package đó. Nếu nhiều package import một package, package được import chỉ được khởi tạo một lần. Việc import package, theo thiết kế, đảm bảo không có chu trình phụ thuộc khởi tạo. Cụ thể:

Với danh sách tất cả package, sắp xếp theo import path, mỗi bước chọn package chưa khởi tạo đầu tiên trong danh sách mà tất cả các package import (nếu có) đã được khởi tạo, và khởi tạo package đó. Lặp lại bước này cho đến khi tất cả package được khởi tạo.

Khởi tạo package—khởi tạo biến và gọi hàm `init`—diễn ra trong một goroutine, tuần tự, mỗi lần một package. Một hàm `init` có thể khởi tạo các goroutine khác, có thể chạy song song với mã khởi tạo. Tuy nhiên, việc khởi tạo luôn tuần tự các hàm `init`: không gọi hàm tiếp theo cho đến khi hàm trước trả về.

Program execution

Một chương trình hoàn chỉnh được tạo bằng cách liên kết một package duy nhất không import nào gọi là *main package* với tất cả các package nó import, kể cả transitively. Main package phải có tên package là `main` và khai báo hàm `main` không nhận đối số và không trả về giá trị.

func main() { … }

Thực thi chương trình bắt đầu bằng khởi tạo chương trình và sau đó gọi hàm `main` trong package `main`. Khi lời gọi hàm đó trả về, chương trình kết thúc. Nó không chờ các goroutine khác (không phải `main`) hoàn thành.

Errors

Kiểu khai báo trước `error` được định nghĩa như sau

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

Đây là interface quy ước để biểu diễn điều kiện lỗi, với giá trị nil biểu thị không có lỗi. Ví dụ, một hàm đọc dữ liệu từ file có thể được định nghĩa:

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

Run-time panics

Lỗi thực thi như truy cập ngoài chỉ số mảng sẽ gây ra *panic lúc chạy* tương đương với việc gọi hàm dựng sẵn `panic` với giá trị kiểu interface do triển khai định nghĩa `runtime.Error`. Kiểu này thỏa mãn interface khai báo trước `error`. Giá trị lỗi cụ thể đại diện cho các điều kiện lỗi lúc chạy là không xác định.

```
package runtime

type Error interface {
error
// và có thể có các phương thức khác
}
```

System considerations

Package `unsafe`

Package dựng sẵn `unsafe`, được compiler biết đến và truy cập qua import path `"unsafe"`, cung cấp các chức năng lập trình cấp thấp bao gồm các thao tác vi phạm hệ thống kiểu. Package sử dụng `unsafe` phải được kiểm tra thủ công về an toàn kiểu và có thể không di động. Package cung cấp giao diện sau:

```
package unsafe

type ArbitraryType int // viết tắt cho một kiểu Go
bất kỳ; không phải kiểu thực
type Pointer *ArbitraryType

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

type IntegerType int // viết tắt cho kiểu số
nguyên; không phải kiểu thực
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
```

`Pointer` là một kiểu con trỏ nhưng giá trị `Pointer` có thể không được giải tham chiếu. Bất kỳ con trỏ hoặc giá trị kiểu underlying type `uintptr` đều có thể chuyển đổi sang kiểu cơ sở là `Pointer` và ngược lại. Nếu các kiểu tương ứng là type parameter, tất cả các kiểu trong tập kiểu tương ứng phải có cùng kiểu cơ sở, phải là `uintptr` và `Pointer`. Hiệu ứng của việc chuyển đổi giữa `Pointer` và `uintptr` phụ thuộc vào triển khai.

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

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

func f[P ~*B, B any](p P) uintptr {
return uintptr(unsafe.Pointer(p))
}

var p ptr = nil
```

Các hàm `Alignof` và `Sizeof` nhận một biểu thức `x` bất kỳ kiểu nào và trả về độ căn chỉnh hoặc kích thước, tương ứng, của một biến giả định `v` như thể `v` được khai báo qua `var v = x`.

Hàm `Offsetof` nhận một selector (có thể có ngoặc) `s.f`, chỉ trường `f` của struct do `s` hoặc `*s` chỉ đến, và trả về offset của trường tính bằng byte so với địa chỉ struct. Nếu `f` là trường nhúng, nó phải truy cập được mà không qua giải tham chiếu con trỏ. Với struct `s` có trường `f`:

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

Kiến trúc máy tính có thể yêu cầu địa chỉ bộ nhớ phải *aligned*; tức là, địa chỉ của biến phải là bội số của một số, gọi là *alignment* của kiểu biến đó. Hàm `Alignof` nhận một biểu thức chỉ biến bất kỳ kiểu nào và trả về độ căn chỉnh (kiểu của biến) tính bằng byte. Với biến `x`:

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

Một (biến kiểu) `T` có *kích thước biến đổi* nếu `T` là type parameter, hoặc là kiểu mảng hoặc struct chứa phần tử hoặc trường có kích thước biến đổi. Nếu không thì kích thước là *cố định*. Gọi `Alignof`, `Offsetof`, và `Sizeof` là biểu thức hằng số kiểu `uintptr` nếu đối số (hoặc struct `s` trong selector `s.f` với `Offsetof`) là kiểu có kích thước cố định.

Hàm `Add` cộng `len` vào `ptr` và trả về con trỏ cập nhật `unsafe.Pointer(uintptr(ptr) + uintptr(len))` [Go 1.17]. Đối số `len` phải là kiểu số nguyên hoặc hằng số không kiểu. Đối số hằng phải có thể biểu diễn bằng giá trị kiểu `int`; nếu là hằng không kiểu thì được gán kiểu `int`. Các quy tắc về sử dụng hợp lệ của `Pointer` vẫn áp dụng.

Hàm `Slice` trả về một slice có mảng cơ sở bắt đầu tại `ptr` và độ dài, dung lượng là `len`. `Slice(ptr, len)` tương đương với

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

trừ trường hợp đặc biệt, nếu `ptr` là `nil` và `len` là 0, `Slice` trả về `nil` [Go 1.17].

Đối số `len` phải là kiểu số nguyên hoặc hằng số không kiểu. Đối số hằng phải không âm và có thể biểu diễn bằng giá trị kiểu `int`; nếu là hằng không kiểu thì được gán kiểu `int`. Lúc chạy, nếu `len` âm, hoặc nếu `ptr` là `nil` và `len` khác 0, sẽ gây ra panic lúc chạy [Go 1.17].

Hàm `SliceData` trả về con trỏ đến mảng cơ sở của đối số `slice`. Nếu dung lượng `cap(slice)` khác 0, con trỏ đó là `&slice[:1][0]`. Nếu `slice` là `nil`, kết quả là `nil`. Nếu không thì là con trỏ không `nil` đến địa chỉ bộ nhớ không xác định [Go 1.20].

Hàm `String` trả về giá trị `string` có các byte cơ sở bắt đầu tại `ptr` và độ dài là `len`. Các yêu cầu với đối số `ptr` và `len` giống như hàm `Slice`. Nếu `len` là 0, kết quả là chuỗi rỗng `""`. Vì chuỗi Go là bất biến, các byte truyền vào `String` không được thay đổi sau đó. [Go 1.20]

Hàm `StringData` trả về con trỏ đến các byte cơ sở của đối số `str`. Với chuỗi rỗng, giá trị trả về là không xác định, có thể là `nil`. Vì chuỗi Go là bất biến, các byte trả về bởi `StringData` không được thay đổi [Go 1.20].

Size and alignment guarantees

Với kiểu số, đảm bảo các kích thước sau:

```
type size in bytes

byte, uint8, int8 1
uint16, int16 2
uint32, int32, float32 4
uint64, int64, float64, complex64 8
complex128 16
```

Các thuộc tính căn chỉnh tối thiểu sau được đảm bảo:

1. Với biến `x` bất kỳ kiểu nào: `unsafe.Alignof(x)` ít nhất là 1.
2. Với biến `x` kiểu struct: `unsafe.Alignof(x)` là lớn nhất trong các giá trị `unsafe.Alignof(x.f)` với mỗi trường `f` của `x`, nhưng ít nhất là 1.
3. Với biến `x` kiểu mảng: `unsafe.Alignof(x)` giống như căn chỉnh của biến kiểu phần tử mảng.

Kiểu struct hoặc mảng có kích thước zero nếu không chứa trường (hoặc phần tử) nào có kích thước lớn hơn zero. Hai biến khác nhau có kích

thước zero có thể có cùng địa chỉ trong bộ nhớ.

Appendix

Language versions

Cam kết tương thích Go 1 đảm bảo các chương trình viết theo đặc tả Go 1 sẽ tiếp tục biên dịch và chạy đúng, không thay đổi, trong suốt vòng đời của đặc tả đó. Nói chung, khi có điều chỉnh và bổ sung tính năng cho ngôn ngữ, cam kết này đảm bảo chương trình Go chạy được với một phiên bản ngôn ngữ cụ thể sẽ tiếp tục chạy được với mọi phiên bản sau đó.

Ví dụ, khả năng dùng tiền tố `0b` cho literal số nhị phân được giới thiệu từ Go 1.13, được chỉ ra bằng [Go 1.13] trong phần literal số nguyên. Mã nguồn chứa literal như `0b1011` sẽ bị từ chối nếu phiên bản ngôn ngữ mà compiler sử dụng cũ hơn Go 1.13.

Bảng sau mô tả phiên bản ngôn ngữ tối thiểu cần thiết cho các tính năng được giới thiệu sau Go 1.

Go 1.9

- Khai báo alias có thể dùng để khai báo tên alias cho kiểu.

Go 1.13

- Literal số nguyên có thể dùng tiền tố `0b`, `0B`, `0o`, và `0O` cho nhị phân và bát phân.
- Literal số thực hệ 16 có thể viết với tiền tố `0x` và `0X`.
- Hậu tố số ảo `i` có thể dùng với mọi literal số nguyên hoặc số thực (nhị phân, thập phân, hệ 16), không chỉ thập phân.
- Các chữ số của literal số có thể ngăn cách bằng dấu gạch dưới `_`.
- Số shift trong phép shift có thể là kiểu số nguyên có dấu.

Go 1.14

- Việc nhúng một phương thức nhiều lần qua các interface nhúng khác nhau không còn là lỗi.

Go 1.17

- Một slice có thể chuyển đổi sang con trỏ mảng nếu kiểu phần tử trùng và độ dài mảng không lớn hơn slice.
- Package `unsafe` có thêm hàm `Add` và `Slice`.

Go 1.18

Phiên bản 1.18 bổ sung hàm và kiểu đa hình ("generics") cho ngôn ngữ. Cụ thể:

- Tập toán tử và dấu câu có thêm token `~`.
- Khai báo hàm và kiểu có thể khai báo type parameter.
- Interface có thể nhúng kiểu bất kỳ (không chỉ tên interface) cũng như union và phần tử kiểu `~T`.
- Tập kiểu khai báo trước có thêm kiểu `any` và `comparable`.

Go 1.20

- Một slice có thể chuyển đổi sang mảng nếu kiểu phần tử trùng và độ dài mảng không lớn hơn slice.
- Package `unsafe` có thêm hàm `SliceData`, `String`, và `StringData`.
- Kiểu so sánh được (như interface thông thường) có thể thỏa mãn ràng buộc `comparable`, kể cả khi type argument không thực sự so sánh được.

Go 1.21

- Tập hàm khai báo trước có thêm hàm `min`, `max`, và `clear`.
- Suy luận kiểu sử dụng kiểu của phương thức interface để suy luận. Nó cũng suy luận type argument cho hàm generic gán cho biến hoặc truyền làm đối số cho hàm khác (có thể là generic).

Go 1.22

- Trong "for" statement, mỗi vòng lặp có tập biến lặp riêng thay vì dùng chung biến cho mọi vòng.
- "for" với "range" clause có thể lặp qua giá trị số nguyên từ 0 đến giới hạn trên.

Go 1.23

- "for" với "range" clause chấp nhận hàm iterator làm biểu thức range.

Go 1.24

- Khai báo alias có thể khai báo type parameter.

Type unification rules

Quy tắc hợp nhất kiểu mô tả nếu và cách hai kiểu hợp nhất. Chi tiết cụ thể quan trọng với các trình biên dịch Go, ảnh hưởng đến thông báo lỗi (ví dụ compiler báo lỗi suy luận kiểu hay lỗi khác), và có thể giải thích vì sao suy luận kiểu thất bại trong các trường hợp bất thường. Tuy nhiên, khi viết mã Go, phần lớn có thể bỏ qua các quy tắc này: suy luận kiểu được thiết kế để "hoạt động như mong đợi", và các quy tắc hợp nhất được tinh chỉnh phù hợp.

Việc hợp nhất kiểu được kiểm soát bởi *chế độ so khớp* (matching mode), có thể là *exact* hoặc *loose*. Khi hợp nhất đệ quy cấu trúc kiểu hợp thành, chế độ so khớp dùng cho phần tử (element matching mode) giữ

nguyên như chế độ so khớp trừ khi hai kiểu được hợp nhất cho khả năng gán (≡A): khi đó, chế độ so khớp là *loose* ở cấp cao nhất nhưng chuyển sang *exact* cho kiểu phần tử, phản ánh rằng các kiểu không cần giống hệt để có thể gán.

Hai kiểu không phải type parameter bị ràng buộc hợp nhất chính xác nếu một trong các điều kiện sau đúng:

- Cả hai kiểu giống hệt.
- Cả hai kiểu có cấu trúc giống hệt và kiểu phần tử hợp nhất chính xác.
- Chỉ một kiểu là type parameter chưa ràng buộc, và tất cả kiểu trong tập kiểu của nó hợp nhất với kiểu còn lại theo quy tắc hợp nhất cho ≡A (loose ở cấp cao nhất và exact cho kiểu phần tử).

Nếu cả hai là type parameter bị ràng buộc, chúng hợp nhất theo chế độ so khớp nếu:

- Cả hai type parameter giống hệt.
- Tối đa một type parameter có type argument đã biết. Khi đó, hai type parameter được *gộp*: cả hai đại diện cho cùng một type argument. Nếu chưa cái nào có type argument, type argument suy luận cho một cái sẽ đồng thời áp dụng cho cả hai.
- Cả hai có type argument đã biết và các type argument hợp nhất theo chế độ so khớp.

Một type parameter bị ràng buộc P và kiểu khác T hợp nhất theo chế độ so khớp nếu:

- P chưa có type argument. Khi đó, T được suy luận làm type argument cho P.
- P đã có type argument A, A và T hợp nhất theo chế độ so khớp, và một trong các điều kiện sau đúng:

    - Cả A và T là interface: Nếu cả hai là kiểu định nghĩa, chúng phải giống hệt. Nếu không, nếu cả hai không phải kiểu định nghĩa, chúng phải có cùng số phương thức (việc hợp nhất A và T đã đảm bảo các phương thức khớp).
    - Cả A và T không phải interface: Nếu T là kiểu định nghĩa, T thay thế A làm type argument cho P.

Cuối cùng, hai kiểu không phải type parameter bị ràng buộc hợp nhất loose (và theo element matching mode) nếu:

- Cả hai kiểu hợp nhất chính xác.
- Một kiểu là kiểu định nghĩa, kiểu còn lại là type literal, không phải interface, và kiểu cơ sở hợp nhất theo element matching mode.
- Cả hai là interface (không phải type parameter) với type terms giống hệt, cả hai hoặc không cái nào nhúng kiểu comparable, kiểu phương thức tương ứng hợp nhất chính xác, và tập phương thức của một interface là tập con của interface còn lại.
- Chỉ một kiểu là interface (không phải type parameter), các phương thức tương ứng của hai kiểu hợp nhất theo element matching mode, và tập phương thức của interface là tập con của kiểu còn lại.
- Cả hai kiểu có cùng cấu trúc và kiểu phần tử hợp nhất theo element matching mode.