GoBackwards Language Reference Manual

Shaquan Nelson (sdn2115), Tahmid Munat (tfm2109), Julian Silerio (jjs2245)

Peter Richards (pfr2109), Catherine Zhao (caz2114)

Introduction

This is a reference manual for the programming language GoBackwards. GoBackwards offers an innovative approach to Go and its problems by making a simpler, easier to understand language while also implementing the web-server platform that Go is known for.

GoBackwards is made up of identifiers, keywords, operators and punctuation, and literals. White space is ignored unless used to separate tokens. GoBackwards uses single semicolons ; as terminators. This manual will cover the GoBackwards language in depth.

Lexical Conventions

Keywords

GoBackwards contains the following keywords, a reduced selection compared to full Go:

case	default	func	interface	return	type
chan	else	go	map	struct	var
const	for	if	range	switch	println

Operators and Punctuation

Operators combine operands into expressions. GoBackwards will provide the following types of operators: unary, arithmetic, logical, and comparison.

The following character sequences represent the operators and punctuation in GoBackwards:

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

Integer Literals

In GoB, an integer literal is a sequence of digits representing an integer constant.
int_lit = decimal_lit | octal_lit | hex_lit .
decimal_lit = ("1" … "9") { decimal_digit } .

Operator Precedence

Unary operators have the highest precedence. There are five levels of precedence for binary operators. Multiplicative arithmetic binds strongest, then additive arithmetic operators, followed by comparison operators, logical AND, and finally logical OR.

Unary Operators

- ***-expression*** : Denotes *negation*. The expression returned is of the same type. The expression must be an `int` or `char`.
- ****expression*** : Denotes *indirection*. The expression must be a `pointer` and an lvalue is returned.
- ***!expression*** : Denotes *logical negation*. If the value of the expression is 0 then ! returns 1. If the value of the expression is non-zero then 0 is returned.

Arithmetic Operators

- ***expression + expression*** : Returns the *sum* of expressions. If both operands are `int` or `char` then an `int` is returned. A `pointer` operand and the other is an `int` or `char`, then a pointer to another object of the same type is returned. For example, “P+1” is a pointer to another object of the same type as the first and immediately following it in storage.
- ***expression - expression*** : Returns the *difference* of expressions. The same considerations for + apply.
- ***expression * expression*** : Returns the *product* of expression one and expression two.
- ***expression / expression*** : Returns the *quotient* of expression one divided by expression two.
- ***expression % expression*** : Returns the remainder of the second expression from the second expression.

Logical Operators

- ***expression || expression*** : If either the first expression *or* the second expression are true, then 1 is returned. Otherwise, 0 is returned.
- ***expression && expression*** : If both the first and second expression are true or false, then 1 is returned. Otherwise, 0 is returned.

Comparison Operators

- ***expression < expressio****n* : If the first expression is *less than* the second expression, then 1 is returned.

- ***expression* > *expression*** : If the first expression is *greater than* the second expression, then 1 is returned.
- ***expression* <= *expression*** : If the first expression is *less than or equal to* the second expression, then 1 is returned.
-
- ***expression* >= *expression*** : If the first expression is *greater than or equal to* the second expression, then 1 is returned.
-
- ***expression* == *expression*** : If the first expression is *equal to* the second expression, then 1 is returned.
- ***expression* =! *expression*** : If the first expression is *not equal to* the second expression, then 1 is returned.

Expressions

Expression: specifies the computation of a value by applying operators and functions to operands.

Operands

operand: denotes an elementary value in an expression, may be a literal, a non-blank identifier denoting a constant, variable, or function, or a parenthesized expressions

- ***Operand*** = ***Literal*** | ***OperandName*** | ***MethodExpr*** | "(" ***Expression*** ")" .
- ***Literal*** = ***BasicLit*** | ***CompositeLit*** | ***FunctionLit*** .
- ***BasicLit*** = ***int_lit*** | ***string_lit*** .
- ***OperandName*** = ***identifier*** | ***QualifiedIdent***.

Primary expressions

primary expression: an operand for unary and binary expressions

- ***PrimaryExpr*** =
  Operand |
  Conversion |
  PrimaryExpr Index |
  PrimaryExpr Arguments .
- ***Arguments*** = "(" [(***ExpressionList*** | ***Type*** ["," ***ExpressionList***]) ["..."] [","]] ")" .

Index expressions

index expression: a primary expression denoting the element of an array given in the form ***a***[***x***]
If ***a*** is an array, the index given must be in range of ***a***. ***a***[x] represents the element in array ***a*** at position ***x***. If ***a*** is a string**,** the index given must be in range of ***a***. ***a***[x] represents the value in the string ***a*** at position ***x***. Otherwise ***a***[x] is illegal.

Calls

call: the evaluation and subsequent execution of a function expression ***f*** with any corresponding arguments. The arguments given must be single-valued expressions corresponding to the parameter types of the function.

The function is evaluated in the order of evaluation. Once function value and arguments have been evaluated, the function begins execution, and return parameters are passed by value to the function when the function returns.

Declarations

Declarations and Scope

declaration: binds non-blank identifier to a **constant**, **type**, **variable**, or **function**

- ***declaration*** = c***onstDecl*** | ***typeDecl*** | ***VarDecl***
- ***topLevelDecl*** = ***declaration*** | ***functionDecl*** | ***methodDecl***

The *scope* of the declaration identifier is the text that describes the specified constant, type, variable, or function. GoBackwards uses *blocks* to lexically scope the language.

- ***predeclared identifier***: scoped by the universe bloc
- ***constant, type, variable, or function identifier***: scoped by the package block
- method receiver, function parameter, or result variable: scoped by the function body
- constant or variable identifier declared inside a function:
    - begins at the end of the ConstSpec or VarSpec
    - ends at the end of the innermost containing block.
- type identifier declared inside a function:
    - begins at the identifier in the TypeSpec
    - ends at the end of the innermost containing block

Identifiers may be redeclared within inner blocks, and any identifiers redeclared within inner blocks denote the entity declared by the inner declaration.

Predeclared identifiers

These identifiers are declared in the universe block.

Types:
bool int string

Constants:
true false

Zero value:
nil

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

Uniqueness of identifiers

An identifier is considered *unique* if it is *different* from every other identifier. Different identifiers are spelled differently, otherwise they are the same.

Constant declarations

constant declaration: binds a list of identifiers (the names of the constants) to the values of a list of constant expressions

The number of identifiers must equal the number of expressions. The *n*th left-side identifier is bound to the value of the *n*th right-side expression

- ***ConstDecl*** = "***const***" (***ConstSpec*** | "(" { ***ConstSpec*** ";" } ")") **.**
- ***ConstSpec*** = ***IdentifierList*** [[***Type***] "=" ***ExpressionList***] .
- ***IdentifierList*** = ***identifier*** { "," ***identifier*** } .
- ***ExpressionList*** = ***Expression*** { "," ***Expression*** } .

If the type is present, all constants take the type specified, and the expressions must be assignable to that type. If the type is omitted, the constants take the types of the corresponding expressions. If the expression values are untyped constants, the declared constant remains untyped.

Within a parenthesized *const* declaration list, the expression list may be omitted from any but the first declaration. Such an empty list is aequivalent to the textual substitution of the first preceding non-empty expression list and its type if any.

Omitting the list of expressions is therefore equivalent to repeating the previous list. The number of identifiers must be equal to the number of expressions in the previous list.

Type declarations

type declaration: binds an identifier, the *type name*, to a type. Type declarations come in two forms: alias declarations and type definitions.

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

alias declaration: binds an identifier to the given type

- ***AliasDecl = identifier*** "=" ***Type*** .

type definition: creates a new, distinct type with the same underlying type and binds an identifier to it. Newly created type is called a *defined type* and is different from any other type.

- ***TypeDef = identifier Type***

Variable declarations

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

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

Variables are initialized with a corresponding list of expressions according to rules for assignments and are initialized to its zero value otherwise.

Variables are either given the type of a type that is present or given the type of the corresponding value. The predeclared value nil cannot be used to initialize a variable with no explicit type.

Function declarations

function declaration: binds an identifier to a function

- ***FunctionDecl*** = "***func***" ***FunctionName*** (***Function*** | ***Signature***) .
- **FunctionName** = **identifier** .
- ***Function*** = **Signature FunctionBody** .
- **FunctionBody** = **Block** .

If the function declares result parameters, the function body must end in a terminating statement.

Method declarations

method: a function with a receiver. The function applies its computations to the receiver instead of returning a value.

method declaration: binds an identifier to a method and associates the method with the receiver's base type.

- ***MethodDecl*** = "***func***" ***Receiver MethodName*** (***Function | Signature***) .
- ***Receiver = Parameters*** .

Statements

A statement in GoBackwards starts the execution of a function call as an independent concurrent thread of control within the same address space.

```
GobStmt = "gob" Expression .
```

The expression is not parenthesized and must be a function or method call. As for expression statements, calls of built-in functions are restricted.

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

Return statements

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

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

If there's a function without a result type, a "return" statement must not specify any result values.

```
func noResult() {
      return
}
```

Similar to how the Go language handles return values, we specify three ways in GoBackwards to return values from a function with a result type:

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

```
1.  func exampleF() int {
2.    return 2
3.  }
4.
5.  func exampleF1() (re int64, im int64) {
6.    return -7, -4
7.  }
```

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

```
1.    func exampleF2() (re int64, im int64) {
2.      return exampleF1()
3.    }
```

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

```
1.    func exampleF3() (re int64, im int64) {
2.          re = 7
3.          im = 4
4.          return
5.    }
6.
7.    func (devnull) Write(p []byte) (n int, _ error) {
8.          n = len(p)
9.          return
10.   }
```

Because we keep the three ways of Go, all the result values are initialized to the zero values for their type upon entry to the function. We consider this regardless of how they are declared. A "return" statement that specifies results sets the result parameters before any deferred functions are executed.

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

```
1.    func f(n int) (res int, err error) {
2.      if _, err = f(n-1); err != nil {
3.         return  // invalid return statement: err is shadowed
4.      }
5.      Return
6.    }
```

Built in Functions

Built-in functions are predeclared, called like any other function, and some of them accept a type instead of an expression as the first argument. They do not have standard Go types, so they can only appear in call expressions and cannot be used as function values.
We preserve the following builtins from golang:

Close

For a channel c, the built-in function close(c) records that no more values will be sent on the channel. It is an error if c is a receive-only channel. Sending to or closing a closed channel causes a runtime panic. Closing the nil channel also causes a runtime panic. After calling close, and after any previously sent values have been received, receive operations will return the zero value for the channel's type without blocking. The multi-valued receive operation returns a received value along with an indication of whether the channel is closed.

Length and capacity

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

```
Call       Argument type    Result

len(s)     string type      string length in bytes
           [n]T, `[n]T      array length (== n)
           []T              slice length
           map[K]T          map length (number of defined keys)
           chan T           number of elements queued in channelbuffer

cap(s)     [n]T, `[n]T      array length (== n)
           []T              slice capacity
```

```
chan T          channel buffer capacity
```

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

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

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

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

```
const (
      c1 = imag(2i)                    // imag(2i) = 2 is a
constant
      c2 = len([10]int64{2})           // [10]int64{2} contains
no function calls
      c3 = len([10]int64{c1})          // [10]int64{c1} contains
no function calls
      c4 = len([10]int64{imag(2i)})    // imag(2i) is a constant
and no function call is issued
      c5 = len([10]int64{imag(z)})     // invalid: imag(z) is a
(non-constant) function call
)
var z complex128
```

Making slices, maps

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

Call	Type T	Result
make(T, n)	slice	slice of type T with length n and capacity n

make(T, n, m) slice	slice	slice of type T with length n and capacity m
make(T)	map	map of type T
make(T, n)	map	map of type T with initial space for approximately n elements

The size arguments n and m must be of integer type or untyped. A constant size argument must be non-negative and representable by a value of type int. If both n and m are provided and are constant, then n must be no larger than m. If n is negative or larger than m at runtime, a runtime panic occurs.

```
s = make([]int, 10, 100)        // slice with len(s)==10,
cap(s)==100
s = make([]int, 10, 0)          // illegal: len(s) > cap(s)
M = make(map[string]int, 100)   // map with initial space for
approximately 100 elements
```

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

Deletion of map elements

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

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

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

Program initialization and execution

We exercise the ‘zero value’ initialization idea from Go when no explicit initialization is provided.

The zero value

When storage is allocated for a variable or when a new value is created, either through a composite literal or a call of make, and no explicit initialization is provided, the variable or

value is given a default value. Each element of such a variable or value is set to the *zero value* for its type: 0 for integers, "" (empty string) for strings, and nil for pointers, functions, interfaces, slices, and maps. This initialization is done recursively, so for instance each element of an array of structs will have its fields zeroed if no value is specified.

These two simple declarations are equivalent:

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

After

```
type T struct { i int; next `T }
t = new(T)
```

the following holds:

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

The same would also be true after

```
var t T
```

Program execution

A complete GoBackwards program is run by the *main* function. The program must declare a function main that takes no arguments and returns no value.

```
func main() { … }
```

Program execution begins by invoking the function *main*. When that function invocation returns, the program exits. It does not wait for other (non-main) GoBackwards routines to complete.

Errors

The predeclared type error is defined as

```
type error interface {
```

```
        Error() string
}
```

It is the conventional interface for representing an error condition, with the nil value representing no error.