Digo

Distributed Golang

Language Reference Manual

Manager: Wenqian Yan (wy2249)
Language Guru: Yufan Chen (yc3858)
System Architect: Sida Huang (sh4081)
Test Designer: Hanxiao Lu (hl3424)

Language Reference Manual

Notation

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

```
Plain Text
```

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

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

```
Plain Text
```

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

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

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

Lexical Elements

Comment

Comments serve as program documentation. Digo supports two forms of comment:

a. ***Inline comment***: starts with the character sequence `//` and stops at the end of the line.
b. ***Comment block****:* starts with the character sequence `/*` and stops with the first subsequent character sequence `*/`.

Comment cannot start inside a literal or inside a comment.

Tokens

Tokens form the vocabulary of Digo. There are five classes: *identifiers*, *keywords*, *operators and punctuation*, and *literals*.

Only white space that separates tokens would combine into a single token, and others (formed from spaces, horizontal tabs, carriage returns, and newlines) is ignored. While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token. Digo hides semicolon delimiters. When the input is broken into tokens, a semicolon is automatically inserted into the token stream immediately after a line's final token.

Identifiers

Identifier in Digo is a case-sensitive ASCII sequence of one or more letters, digits and underscore '_'. The first character in an identifier must be a letter. Identifiers name program entities such as variables and types.

Keywords

Keywords are case-sensitive sequence of letters reserved for use in Digo.The following keywords are reserved and may not be used as identifiers.

Plain Text

```
for, if, else, func, return, await, async, remote,
var, string, int, bool, float, continue, break
true, false
```

Operators and punctuation

The following character sequences represent operators and punctuation:

```
Go
```

```
1  +    &&    =     !=    (    )
2  -    ||    <     <=    [    ]
3  *    ==    >     >=    {    }
4  /    ++    :=          ,    ;
5  %    --    !           .    :
```

Basic Literals

Literals in Digo are the representation of a value of some primitive types. Basic literals in Digo include integer literals, float literals, character literals, string literals, boolean literals. Digo also implements slice literal, which is introduced in the later part.

Integer literals:

Sequence of one or more digits in decimal. For representing negative numbers, a negation operator is prefixed.

Example: 12

Float literals:

Sequence that consists of an integer part, a decimal point and a fraction part. For representing negative numbers, a negation operator is prefixed.

Example: 3.14

String literals:

Sequence of ASCII characters quoted by double quotes. A string can also be empty.

Example: "Digo is Great!"

Boolean literals:

Logical true and false.

Variables

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

A variable declaration or the signature of a function declaration reserves storage for a named variable.

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

Types

A type determines a set of values together with operations and methods specific to those values.

```
Plain Text
1  Type =  "string" | "float" | "bool" | "int" | SliceType | FutureType .
```

Boolean types

The predeclared boolean type is `bool`. A *boolean type* represents the set of Boolean truth values denoted by keywords `true` and `false`.

Integer types

An integer type represents sets of integer values. The predeclared type is `int`.

Float types

A float type represents sets of floating numbers. The predeclared type is `float`.

Slice types

A slice is a sequence container representing arrays that can change in size.

```
Plain Text
1  SliceType = "[" "]" Type .
```

The length of a slice `s` can be discovered by the built-in function `len`. The elements can be addressed by integer indices 0 through `len(s)-1`.

Function types

A function type denotes the set of all functions with the same parameter and result types.

Plain Text

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

Here are some examples:

Plain Text

```
func()
func(x int) int
func(a int, z float32) (bool)
```

String types

A *string type* represents the set of string values. A string value is a (possibly empty) sequence of bytes. Under the hood, string is implemented by Slice. The predeclared string type is `string`.

The length of a string `s` can be discovered using the built-in function `len`. A string's bytes can be accessed by integer indices 0 through `len(s)-1`.

Future types

A future type represents the set of asynchronous tasks. Under the hood, a future object keep tracks of a task's information, including the task's id, func_id, worker_id, serialized arguments, serialized response, and the current state of the task. The predeclared future type is `future`.

Properties of types and values

Type Identity

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

For `string`, `float`, `int`, `bool`, `future` ,each of these types is always different from any other type.

For `function` types, two function types are identical if they have the same number of parameters and result values, corresponding parameter and result types are identical.

For `slice` types, two slice types are identical if they have identical element types.

Assignability

A value `x` is *assignable* to a variable of type `T` ("`x` is assignable to `T`") iff `x`'s type is identical to `T`.

Scope

The scope of a type identifier declared inside a block (function/if statments/for statement) begins from the identifier declaration to the end of the innermost containing block.

We do not support global variables.

Declarations

A *declaration* binds a non-blank identifier to a variable or a function. Every identifier in a program must be declared. No identifier may be declared twice in the same block, and no identifier may be declared in both the file and package block.

the identifier `master` may only be used for master function declaration, it does not introduce a new binding.

An identifier declared in a block may be redeclared in an inner block

Variable declarations

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

Plain Text

```
VarDecl     = "var" VarSpec .
VarSpec     = IdentifierList Type [ "=" ExpressionList ] .
```

Plain Text

```
var i int = 0
var b string = a + b
var x, y float32 = -1, -2
```

Short variable declarations

A *short variable declaration* uses the syntax:

```
Plain Text
1  ShortVarDecl = IdentifierList ":=" ExpressionList .
```

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

```
Plain Text
1  "var" IdentifierList = ExpressionList .
```

Examples:

```
Plain Text
1  i, j := 0, 10
```

Function declarations

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

```
Plain Text
1  FunctionDecl = "func" FunctionName Signature  FunctionBody  .
2  FunctionName = identifier .
3  FunctionBody = "{" StatementList "}" .
```

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

A function declaration can not omit the body.

Expressions

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

Operands

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

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

Plain Text

```
Operand     = Literal | OperandName | "(" Expression ")" .
Literal     = BasicLit | SliceLit .
BasicLit    = int_lit | float_lit | string_lit | bool_lit .
OperandName = identifier .
```

Slice literals

Slice literals construct values for slices.

Plain Text

```
SliceLit = SliceType LiteralValue .
LiteralValue  = "{" [ ElementList [ "," ] ] "}" .
ElementList   = Element { "," Element } .
Element       = Expression | LiteralValue .
```

- Each element has an associated integer index marking its position in the slice.
- An element uses the previous element's index plus one. The first element's index is zero.

Primary expressions

Primary expressions are the operands for unary and binary expressions.

Plain Text

```
PrimaryExpr =
        Operand |
        PrimaryExpr Index |
        PrimaryExpr Slice |
        PrimaryExpr Arguments .
Index          = "[" Expression "]" .
Slice          = "[" [ Expression ] ":" [ Expression ] "]"
Arguments      = "(" [ ( ExpressionList | Type [ "," ExpressionList ] ) [ "..."
 ] [ "," ] ] ")" .
```

Plain Text

```
x
2
(s + ".txt")
f(3.1415, true)
s[i : j + 1]
p[10]
```

Index expressions

A primary expression of the form `a[x]` denotes the element of the slice. The following rules apply:

- if `x` is out of range at run time, the program sends an segment fault to itself and therefore exits.
- `a[x]` is the slice element at index `x` and the type of `a[x]` is the element type of `S`

Slice expressions

Slice expressions construct a substring or slice from a string or slice. It specifies a low and high bound.

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

Plain Text

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

constructs a substring or slice. The *indices* `low` and `high` select which elements of operand `a` appear in the result. The result has indices starting at 0 and length equal to `high` - `low`.

For convenience, any of the indices may be omitted. A missing `low` index defaults to zero; a missing `high` index defaults to the length of the sliced operand:

Bash

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

For slices or strings, the indices are *in range* if `0` <= `low` <= `high` < `len(a)`, otherwise they are *out of range*. If the indices are out of range at run time, a segment fault occurs.

Calls

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

```Apache
f(a1, a2, … an)
```

calls `f` with arguments `a1, a2, … an`. Except for one special case, arguments must be single-valued expressions assignable to the parameter types of `F` and are evaluated before the function is called. The type of the expression is the result type of `F`.

In a function call, the function value and arguments are evaluated in the usual order. After they are evaluated, if the types of the parameters are then they are passed by value to the function and the called function begins execution.

Operators

Operators combine operands into expressions.

```Plain Text
Expression = UnaryExpr | Expression binary_op Expression | Expression assign_op
 Expression.
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .
binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" .
assign_op  = "=" .
unary_op   = "-" | "!"  .
```

For binary operators, the operand types must be identical.

Operator precedence and associavity

The assocaitivity of operators are as follows:

Operator	Name	Associativity
=	Assign	Right
==	Equal to	Left

!=	Unequal to	Left
>	Greater than	Left
>=	Greater than or equal	Left
<	Less than	Left
<=	Less than or equal to	Left
+	Addition	Left
-	Subtraction	Left
*	Multiplication	Left
/	Division	Left
%	Modulo	Left
&&	Logical and	Left
||	Logical or	Left
!	Logical not	Right

The precedence of operators is as follows:

a. * / %
b. + -
c. > >= < <=
d. == !=
e. !
f. &&
g. ||
h. =

Arithmetic operators

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The four standard arithmetic operators (`+`, `-`, `*`, `/`) apply to integer, floating-point, and complex types; `+` also applies to strings.

SQL

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

Integer operators

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

Plain Text

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

Floating-point operators

For a given integer i, its floating point representation could be achieved by using float(). float() could also be used to convert the intermediate result of operation.

Plain Text

```
float(i)
float(a*b+c)
```

String concatenation

Strings can be concatenated using the `+` operator:

Plain Text

```
s := "hi" + string(c)
```

Comparison operators

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

Shell

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

Logical operators

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

Plain Text

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

Order of evaluation

when evaluating the operands of an expression, assignment, or return statement, all function calls, are evaluated in lexical left-to-right order.

Statements

Statements control execution.

Plain Text

```
Statement =
        Declaration | SimpleStmt |
        AwaitStmt | ReturnStmt | BreakStmt | ContinueStmt | IfStmt | ForStmt  .

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

Terminating statements

A *terminating statement* prevents execution of all statements that lexically appear after it in the same block. The following statements are terminating:

1. A "return" statement
2. An "if" statement in which:
   - the "else" branch is present, and both branches are terminating statements.
3. A "for" statement in which:
   - there are no "break" statements referring to the "for" statement, and the loop condition is absent.

Empty statements

The empty statement does nothing.

Plain Text

```
EmptyStmt = .
```

Expression statements

With the exception of specific built-in functions, functions can appear in statement context. Such statements may be parenthesized.

If statements

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

Plain Text

```
IfStmt = "if" Expression "{" StatementList "}" [ "else" ( IfStmt | "{" Statement
List "}" ) ] .
```

Plain Text

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

For statements

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

Plain Text

```
ForStmt = "for" [ Condition | ForClause ] "{" StatementList "}" .
Condition = Expression .
```

For statements with single condition

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

Plain Text

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

For statements with for clause

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

Plain Text

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

Plain Text

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

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

Plain Text

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

Await Statements

An "await" statement waits in a blocking way for the response of a task denoted by a future object.

Plain Text

```
AwaitStmt = "await" Expression .
```

The expression must be a future object.

Return statements

A "return" statement in a function `F` terminates the execution of `F`, and optionally provides one or more result values.

Plain Text

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

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

Plain Text

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

Break statements

A "break" statement terminates execution of the innermost "for" statement within the same function.

Plain Text

```
BreakStmt = "break" .
```

Continue statements

A "continue" statement begins the next iteration of the innermost "for" loop at its post statement. The "for" loop must be within the same function.

Plain Text

```
ContinueStmt = "continue"  .
```

Built-in functions

Call	Argument type	Result
len(s)	String type Slice type	string length in bytes slice length
append(s, x)	(Slice, element type of the Slice)	Append x to s.
gather(s)	Slice with element type `future`	Await all the future objects and return the result of them

Program Execution

Program execution begins by parsing three arguments:

Bash

```
-m [master|worker] specifies whether this is a master or a worker
-h <string> hostname of the master
-p <int> port of the master
```

If the command line does not contain these arguments, then the program will promt a error and exit.

Then according to whether this is a mster or a worker, the program will have different iniialzation procedures.

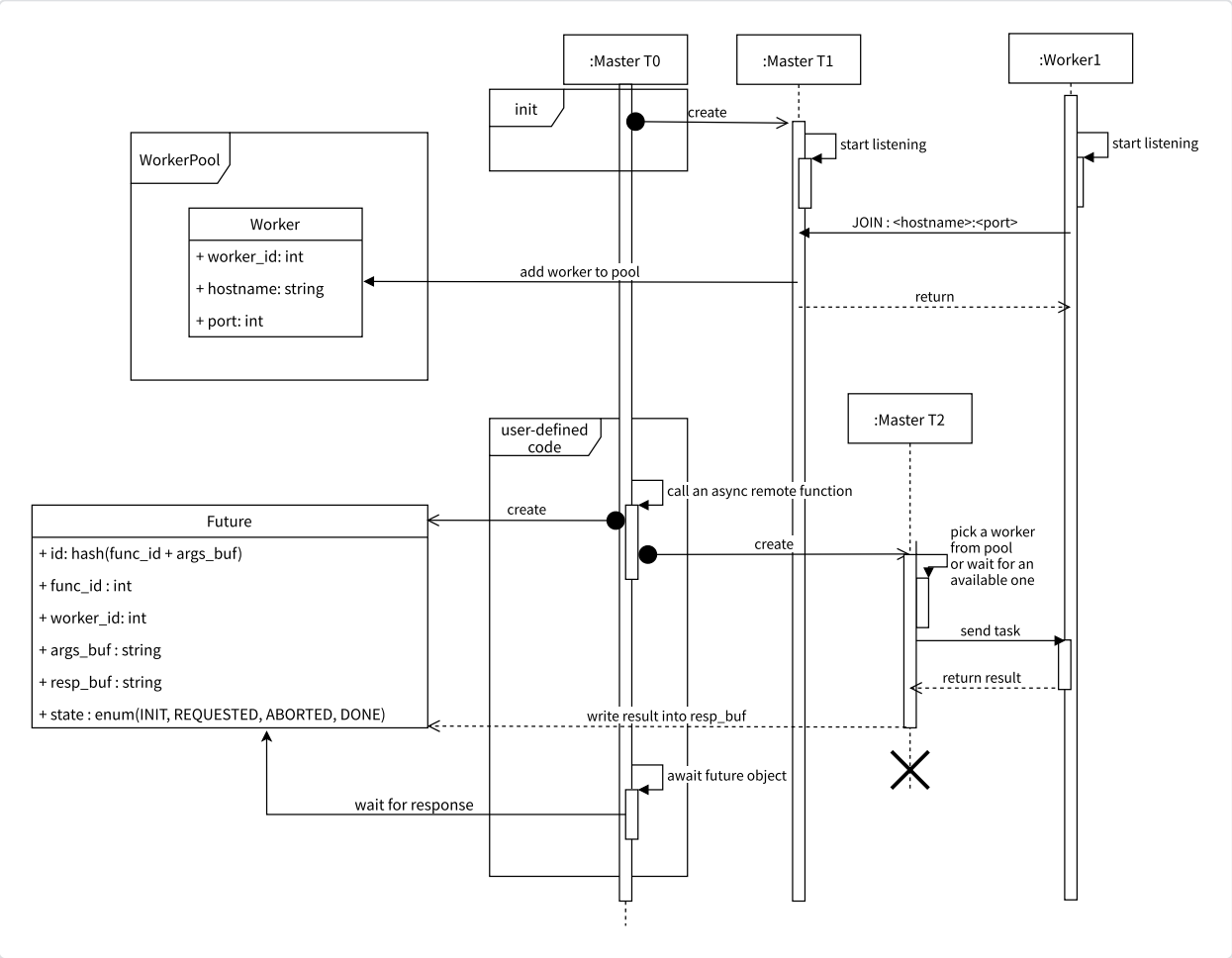
Master

If the program is a master, the program starts listening to <hostname:port> and waits for other workers to send a `join` request and then invoking the function `master`. When that function invocation returns, the program exits.

Worker

If the program is a worker, the program starts listening to a 0.0.0.0:<random_port>. Then the worker sends a `join` request to the master(the hostname and port of which are gotten from the cmdline arguments). Then the worker keeps alive and waiting for remote tasks from the master program, executes it and returns the result back to the master.

Overall Procedure



Reference

https://golang.org/ref/spec