GO--

Language Reference Manual (LRM)

Chen Chen (cc4351)
Lingyu (Arya) Zhao (lz2650)
Yang Li (yl4111)
Yuyan Ke (yk2822)

1 Comments and Whitespace

1.1 Comments

Comments are written between /* and */, and can be nested as long as the opening /∗ and closing ∗/ are all matched.

/* single-line comment */

/* multi-line comment
multi-line comment*/

/* nested /* multi-line */ comment */

1.2 Whitespace

Whitespace, including newline characters, tabs, and spaces, is used only to separate tokens and is otherwise ignored by our Go-- compiler.

2 Data Type

2.1 Data Types & Operations

This language has primitive data types including int, bool, float, char and string. Unlike C, this language doesn't have pointers. The declaration of array type is of the format dataType [arraySize] arrayName. For example, int[3] arr.

Data Type	Description	Operations	Examples
int	At least 2 bytes, usually 4 bytes	==, < , >, !=, +, -, *, /, %, <=, >=, +=, -=, ++, --	6 ++; /* 7 */ 6 / 3; /* 2 */ 7 % 3; /* 1 */ 6 < 7; /* false */
float	8 bytes	==, < , >, !=, +, -, *, /, <=, >=, +=, -=, ++, --	5.22 + 2.0; /* 7.22 */ 4.3 < 5.6; /* true */
bool	1 bit	==, !, !=, &&, ||	x = true; !x; /* false */ 1 == 3; /* false */
char	1 byte	=, ==, !=, +, ++, –, <, >, <=, >=	char test = ‘h’; ‘A’ < ‘B’; /* true */
string	Size varies. An immutable array of chars	=, ==, !=, <, >, <=, >= (lexicographical), + (concatenate)	x = “hi”; y = “boye”; x + y; /* returns ”hiboye” */

func	Standard function type: runs in the same thread	N/A	function int foo(int x, int y){
			  int ret = x + y ;
			  return ret;
			}
			foo(1, 2);
			 /* run in same thread */

gofunc	Concurrent function type: runs in a new thread	To run the gofunction, a go keyword is needed before the gofunc identifier	gofunction int foo(int x, int y){
			  int ret = x + y ;
			  return ret;
			}
			
			go foo(1, 2)
			/* run in a new thread */

channel	Shared memory for all threads	<-, ->	channel example= new_channel(int, 3); 5 -> example; /* push 5 into channel */ x <- example; /* x = 5 */
dataType[]	array	indexing []	int mark[] = [19, 10, 8, 17,

			9]; mark[0]; /* 19 */

2.2 The Void Keyword

The type void has no associated value and can only be used as the return type for functions that returns nothing. This is useful for functions which are intended to perform "side-effect" operations only. The return statement can be omitted in this case or can be written as:

```
return;
```

3 Variables

3.1 Variable Naming

All variable names must follow [a-z A-Z][a-z A-Z 0-9]* and cannot be any of the reserved words listed below.

Use	Reserved Words						
booleans	true	false					
Control flows	if	while	else	for			
data types	int	float	char	string	bool	func	gofunc
functions	function	gofunction	go	void	return		
data structures	array	struct	channel				

3.2 Scope of Variables

Go-- is a statically scoped language. Variables declared inside blocks, where blocks include functions, for loops, if statements, while loops, and struct definitions, exist only inside the block in which they are declared and override any variables of the same name declared before that

block within that block only. Variables outside of all blocks have global scope and thus can be accessed anywhere in the program following their declaration. Multiple variables and/or functions of the same name, even if their types differ, cannot be declared in the same scope. Variable, struct, and function declarations are not visible to statements that precede them so Go-- does not support recursive or mutually recursive functions or struct type definitions.

3.3 Variable Declaration and Assignment

Variables must be declared with a type and a name in the form of

```
type variable ;
```

type: Go-- data types and data structures
variable: string

Variables assignments must follow the convention shown above with a type followed by a proper variable name. The variable declaration statement must terminate with a semicolon. Variable types can be any of the types listed in Seciont 2.1.

4 Statement

Statements are executed in sequence.

4.1 Expression Statement

Expression statements include assignments and function calls take the form of

```
expression ;
```

An empty statement is also possible, often for loops, denoted by

```
;
```

4.2 Compound Statement

Compound statements are organized into blocks within braces `{}`, such that an open brace must be matched with a corresponding a closing brace in the form of

```
Statement_list {
     Statement;
     Statement;
     …
}
```

4.3 Conditional Statement

The two forms of conditional statements are

```
if (expression) {statement }

if (expression) { statement } else  statement
```

In all cases, expression is evaluated first and the corresponding statement block will execute if the expression results in a non-zero value. If the result of expression is zero and there exists sequential else if blocks, then the following expression for the else if block will be evaluated the same way as the original if block. This process continues for each else if block in a sequential order. If the result from the expression evaluation is zero and the else block follows next, then the statements within the else block will be executed.

4.4 While Statement

The while statement has the form of

```
while (expression) statement
```

The statement within the block is executed repeatedly while the expression is evaluated to be true or non-zero. The expression is re-evaluated after each iteration of the execution of the statement.

4.5 For Statement

The for statement has the form of

```
for (expression1; expression2; expression3)  statement
```

equivalent to:

```
expression1;
while (expression2) {
```

```
        statement
        expression3
    }
```

such that expression1 denotes the starting value for the loop, expression2 denotes the test made after each iteration, and expression3 denotes an incrementation performed after each iteration. The loop terminates when expression2 evaluated to be zero.

Note that all three expressions are optional. A missing expression2 would make the equivalent while (expression2) evaluates to “while(1)”.

4.6 Return Statement

Return statement taken one of the following forms

```
return expression ;
return ;
```

In the first case, the expression is returned to the caller of the function. In the second case, no value is returned. Since the return expression must be the same type as specified in the function declaration, the no value in case two will be returned as a null object for the specified type.

5 Channel

5.1 Channel Data Structure

Channel is a special data structure featured in our language meant for inter-threadcommunication. There are two main components in a channel, the first one is an array of primitive types or data structures to hold information to be communicated between functions, the second one is a lock to guarantee data integrity and consistency of the array during read and write.

5.2 Channel Creation

A new channel can be created by calling `new_channel(data_type, size_of_channel)`, where data type could be any of the primitive types or native data structures (struct, array), and size_of_buffer specifies the maximum number of items of matching data types that could be simultaneously stored in the channel. The function new_channel(data_type, size_of_channel) could be called from anywhere

(including in the main function), and the naming of which should be unique. The declared channel should be accessible by name anywhere within the process.

```
    channel my_chan=new_channel(string, 5);
/* creates a new channel called my_chan that could hold up to 5
strings simultaneously */
```

5.3 Enqueue into channel (->)

One may enqueue data into a channel with matching type using the right arrow (->) operator, with the name of the channel on the right-hand side of the -> operator and the variable to be enqueued on the left. If the channel is at its maximum storage capacity, the thread trying to enqueue data into the channel will be blocked on the enqueue statement, until process termination or there is vacancy in the channel. Note that the enqueue operation is protected by an inherent lock in channel such that there could be at most one thread accessing the channel data region at any given time.

```
    Variable ->name_of_channel
```

5.4 Dequeue from channel (<-)

One may dequeue data from a channel with the <- operator, with the name of the channel on the right-hand size of the <- operator. Only one item of matching data type could be retrieved and removed from the channel with one <- operation. If the channel is empty, the thread trying to dequeue data from the channel will be blocked on the <- statement, until process termination or there is new data enqueued into the channel. Note that the dequeue operation is protected by the inherent lock in channel such that there could be at most one thread accesssing the channel at any given time. Note that the variable name on the left-hand side could be omitted when the value of the dequeued item is not evaluated, or directly fed into another function.

```
    ( Variable| None ) <- Name_of_channel
```

6 Array

Arrays are containers, denoted with `[]`, with a fixed size to group a number of items of the same type, primitive or a composite type defined by a struct.

6.1 Declaring Array

Declaration of arrays need to be in the following form

```
    type[expr] variable ;
```

such that type defines the type of each element, num defines the max number of elements allowed in an array, and variable defines the variable name for the array. When the array declaration ends with a semicolon with assignment, it has initialized the correct space in memory to hold elements when needed.

6.2 Defining and Indexing Array

Array definition would be achieved in the form of

```
int[5] variable1;

variable1 = [ element_0, element_1, element_2, element_3,
element_4 ];

type[n] variable2 = [ element_0, element_1, …,
element_(n-1) ];
```

The first case assumed that variable1 has been declared prior to be int[5], and the array named variable1 has the elements in the square brackets in the order they are given. The second case declared an array variable for n elements and assign each element in the array to be the corresponding element in the braces. The first element of all arrays occupy position 0, or index 0, thus the last element always occupy position (n-1). Individual element in an array could be accessed directly with its index in the form of

```
variable[index];
```

such that variable is the name of the array and index specifies the desired index of the element.

6.3 Arrays of Array

We define array as a type and allow nested arrays as well.

```
int[5][4] //array of 5 elements, each one is a 4-int array
```

7 Structs

7.1 Struct Declaration

A struct is a sequence of semi-colon-separated typed variables, which could be any data type supported by Go--, including func, gofunc and structs. Note that we do not allow anonymous

declaration of nested structs. In other words, all structs, no matter nested or not, need to be named. Structs are defined with “struct” keyword. The variables nested in the struct should be defined in the same way as the normal variables, and must start with a letter followed by a combination of letters and numbers. Field names need to be unique within the same struct. All field declarations are encapsulated in a pair of braces {}. Initialization of fields is not allowed during struct declaration.Here is a sample declaration:

/* struct_type definition */

Struct_type:

struct { type-decl-list } ;

struct identifier {type-decl-list} ;

struct identifier ;

Type-decl-list:

Type-declaration

Type-declaration Type-decl-list

Type-declaration:

Typ declarator;

```
struct sample {
    int prime;
    func bool checkPrime (int );
    struct Nested {
        float field1;
        int[5] field2;
    };
```

```
};
```

7.2 Struct Variable Instantiation

One has to declare a struct before creating an instance of it. Otherwise, the compiler would complain about missing type definition when processing the code. The name of the struct is used as the type name of the struct. One can simply instantiate a struct as on line 10, but also on line 9, when declaring a struct.-

```
struct sample err; // error: type def not found
struct ample {
   int prime;
   func bool checkPrime ( int );
   struct nested {
       float field1;
       int[5] field2;
   };
} anotherSample;
struct sample correct;
```

7.3 Reading and Updating Struct Fields

Struct fields could be accessed with the dot operator (.). One would need the name of the struct variable, followed by the dot and then by the name of the field to get the value. Incorrect variable name or field name would result in a compilation error. If a field is not initialized, the type default value would be returned. For example, 0 will be returned if the uninitialized field is of type int. When accessing nested fields, one needs to perform one dot operation per struct for access. Here is a short example with the same struct definition as specified in 7.1 and 7.2.

```
struct sample correct2;

correct2.Nested.field2[1] = 1;

correct2.prime /*evaluates to 0, the default */

correct2.Nested.field2 /*evaluates to 1, the value
initialized */
```

8 Arithmetic Operators

8.1 Order of Evaluation

Our arithmetic operators follow the PEMDAS convention. In other words, () takes priority over *, / and %, over + and -. All operators except the NOT operator (!) are left associative; NOT(!) is right associative.

8.2 Addition (+) and Subtraction (-) Operators

We allow add (+) between two variables of type int or float, or two expressions that would yield int or float. Note that the expressions on the two ends of the operator need to be of the same type, otherwise a compilation error would be thrown. There is no automatic type promotion from int to float. The minus operator also only accepts two variables of type int or float, or expressions that evaluates to int or float, and is used for subtraction in the traditional mathematical sense. We also allow ++ and –- as shorthand for +1 and -1.

```
1   int a = 1;
2   int b = 3;
3   float c = 2.2;
4   a+b;//int 4
5   a+c;//error
6   b-c;//error
7   a++; //int 2
8   c--; //error
9
10  string h = "hello";
11  string t = "ocaml";
12  h+t; //string "hello
13  ocaml"
    h+a; //error: type
    mismatch
```

8.3 Multiplication (*) and Division (/) Operators

The multiplication (*) and division (/) operators are used in the traditional mathematical sense for ints and floats only. When performing int-int division, the quotient is kept and the remainder is discarded, which is the equivalent of rounding down. If at least one of the numerator and the denominator is of type float, the result would be a float.

```
1   int a = 1;
2   int b = 3;
3   float c = 2.2;
4   a*b;//int 3
5   a*c;//float 2.2
6   b/a;//int 3
7   c/a//float 2.2
```

8.4 Modulo Operator (%)

The percent sign (%) is used for the modulo operation and can be used for ints only.

```
1   int a = 1;
2   int b = 3;
3   float c = 2.2;
4   b%a;//int 3
5   c%a//error
```

8.5 Boolean Operators (<, >, <=, >=, !=, ==)

Boolean operators in this language are: == , !=, <, >, >=, <=. They operate on ints and floats only. There is automatic type promotion when comparing ints and floats so they can be compared without any problems. == and != can also be used for booleans and boolean expressions that evaluate to true or false.

8.6 Logical Operators (!, &&, ||)

! is used for NOT in boolean expressions.
&& is used for AND in boolean expressions.
|| is used for OR in boolean expressions.

9 Functions

Go-- supports two types of functions, normal functions and functions that can run in concurrent manners. Besides that, Go-- also has first-class functions, which means functions can be declared and stored as variables.

9.1 Function declaration

To define a function, we will have the key word of ’function’ or ‘gofunction’ at the beginning of the definition, and it should be followed by the return type of the function, return type void is allowed in Go--, and after the return type programmers should specify the function name followed by parentheses in which function arguments are specified. Function arguments should follow the format “type name” and multiple function arguments should be separated by comma. A function takes no argument when empty parentheses are present. And inside the braces ‘{’ , ‘}’ programmer should put in the function body.

’The function below is an example function called “foo” that takes two integer as arguments and return their sum as results.

```
function int foo(int x, int y){

     int ret = x + y ;

     return ret;

}
```

To call the functions in the program, programmers only have to specify the function name and the function arguments, and the argument types should match the ones in function argument.

For example, to call foo on 1 and 2, one should use the syntax:

```
     foo(1,2);
```

9.2 Concurrent Functions

To define a concurrent function, users should use keyword ‘gofunction’ instead of ‘function’ in the function definition to specify that the function can run in concurrent manner. For example

```
gofunction int goo(int x, int y){

     int ret;

     ret = x + y ;

     return ret;

}
```

is a concurrent function called “goo” that takes two integer as arguments and return their sum as results. Unlike normal functions, when calling concurrent functions, the key word go should be specified. For example, to call goo in the program, one should use the syntax

```
go goo(1,2);
```

And because go function can run in concurrent manner. Code like

```
go goo(1,2);
go goo(3,4);
```

Runs in a concurrent manner. That is, function in line 2 can finish execution before line 1 finishes execution, depending on the underlying scheduling architecture of the operating system.

9.3 First Class Functions

All functions can be treated as variables of built-in types of “gofunc” and “func” and be assigned to variables. To assign a function to a variable, we first need to declare a variable with matching parameter and output types. Then we assign the variable to an anonymous function/gofunction expression using the = operator. The general syntax and a small example are provided:

For example, we can have

```
func int prod (int, int); /* function declaration */

prod = function int (int x, int y) {return x*y;}; /* function
assignment */
```

which means we assign a function that takes two integers as argument and return their product to variable prod.

10 Sample program

Some sample code that demos the use of the any keyword, function definitions, and the utility of concurrent function feature and first-class function features in Go--.

```
// Sample program

// Creating a channel for a maximum of 5 strings
channel message = new_channel(string, 5);
```

```
gofunction void f(int num)
{
    for (int i = 0; i < 3; i++)
    {
        printf("%d", num);
        message <- “goodbye”; // Insert string into channel
    }
    “final goodbye” -> message;
}

function string ftwo(int num)
{
    “hello” -> message;
    return “hello”;
}

func int main()
{
    int a;
    string str;
    func string funcvar (int);
    func void func2var ();

    a = 3;

    // f() is running on a concurrent thread
    go f(a);
    a = 4;

    str = ftwo(a);      // Execute in main thread

    // Store an existing or a new function as a variable
    funcvar = ftwo;
    funcvar(1);
    func2var = function void ()
    {
        while (1){}
    }

    // Read strings from the channel
    for (int i = 0; i < 5; i++)
    {
        printf("%s", <- message);
    }

    return 0;
}
```

11 References

http://www.cs.columbia.edu/~sedwards/classes/2018/4115-fall/lrms/Shoo.pdf
http://www.cs.columbia.edu/~sedwards/classes/2017/4115-fall/lrms/GoBackwards.pdf
C LRM: https://www.bell-labs.com/usr/dmr/www/cman.pdf
Golang channel data structure breakdown:
https://codeburst.io/diving-deep-into-the-golang-channels-549fd4ed21a8