Additional Golang Programming Notes

1. Package: the main package and the fmt package

Students often start by writing the following Go program hello.go to print out "hello!" on the terminal.

```
package main
import "fmt"
func main() {
      fmt.Println("hello!")
}
```

Any Go program belongs to a package, which is declared in the first line of the program. There are two types of programs: (1) standalone executable programs (executables) and (2) library programs (libraries). The above hello.go is a standalone executable program, meaning it can be compiled into a binary code to execute. It must be included in the main package and contain a main function. An executable can import a library, such as "fmt" above. A library program is not standalone. It must be used by other programs. Package reuse is a form of abstraction which increases productivity.

For instance, the library program fmt.go provides input/output formatting functionality and is reused by many programs. It can be understood to have a structure similar to the following source code:

```
package fmt                 // It belongs to the fmt package
……
func Println(…) …{          // It contains a function Println which other
      …                     // programs can call by using fmt.Println
}
……
func Printf(…) …{           // It contains a function Printf which other
      …                     // programs can call by using fmt.Printf
}
……
func Scanf(…) …{            // It contains a function Scanf which other
      …                     // programs can call by using fmt.Scanf
}
```

Note that in Go, a capitalized function is visible by other packages. Thus, we cannot define the Println function by

```
func println(…) …{
      …
}
```

Incidentally, Println stands for “Print Line”. It prints the items one by one according to their default formats. Then, it prints a new line symbol, i.e., "\n".

In contrast, Printf formats and prints each item according to the corresponding formatting verb. Printf has more control of the output format, but using Println results in shorter code.

For instance, the following two statements do the same thing:

```
fmt.Printf("hello! %d\n",63)
fmt.Println("hello!",63)
```

That is, they print the following output:

```
> hello! 63
```

The function Scanf is the opposite of Printf. It inputs data from the terminal by formatting and inputting each item according to the corresponding formatting verb. For instance, the following code receives a user entered integer into variable A.

```
var A int
fmt.Scanf(“%d”,&A)          // &A indicates the address of A
```

Five formatting verbs are listed in Table 1. The three verbs %b, %d, and %x are for binary, decimal, and hexadecimal representations of a number, respectively. The verb %c is for character representation. The %s verb is for string. The verb %c is the most basic formatting verb. Any other verb can be constructed from %c.

Table 1 Formatting verb examples for binary/decimal/hex numbers, character and string

Verb	**Meaning**	**Example**
%b	Binary	fmt.Printf("%b",63) outputs 111111
%c	Character	fmt.Printf("%c",63) outputs ?
%d	Decimal	fmt.Printf("%d",63) outputs 63
%s	String	fmt.Printf("%s",string(63)) outputs ?
%x or %X	Hexadecimal	fmt.Printf("%X",63) outputs 3F

One cannot put special characters such as ‘\’, ‘"’ in a string directly. Instead, escape values starting with a backslash are used, as demonstrated in Table 2.

Table 2 Escape values in a string

Value	**Meaning**	**Example**
\\	Backslash	fmt.Printf("\t Use \\\" to output %c%c\n",92,34)
\t	Tab	Outputs
\n	Newline	Use \" to output \"
\"	Double quote	>

2. How to pronounce the ASCII characters?

ASCII, or US-ASCII by the Internet Assigned Numbers Authority, stands for American Standard Code for Information Interchange. It is widely used in computers and the Internet. We should be able to pronounce the ASCII characters, at least the subset of frequently used characters.

Figure 1 shows binary encodings of ASCII symbols, which consist of control characters (boldfaced red characters) and printable characters.

How to pronounce the ASCII characters? Tables 3 and 4 list the decimal and hexadecimal values, as well as the pronunciations of ASCII control characters and printable characters.

As shown in Fig. 1, the ASCII of character encoding has historical roots in typewriting, telegraph or teleprinting applications. Some notations may be missing in the keyboards of modern digital devices. For instance, the ASCII character for "Carriage Return" reappears as the "Enter" key in many modern keyboards.

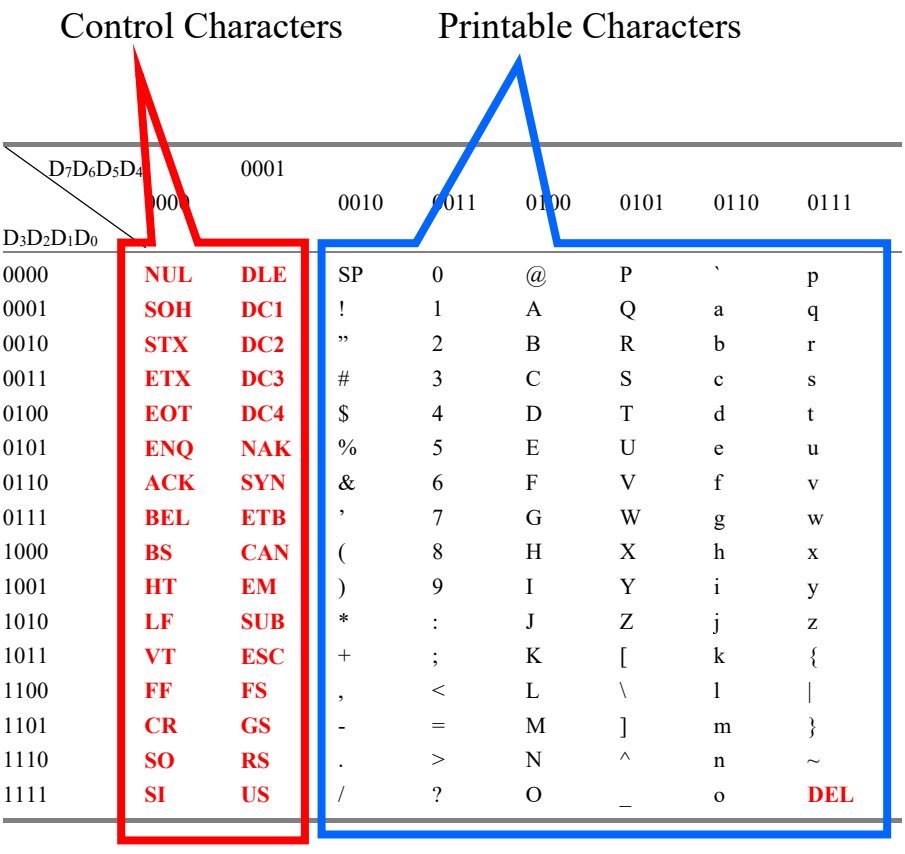

Fig. 1 ASCII symbols consist of control characters and printable control characters

Table 3 Values and pronunciations of the ASCII symbols: **control characters**

Decimal	Hex	Symbol	Description	Chinese
0	0x00	NUL	Null Character	空字符
1	0x01	SOH	Start of Heading	标题开始
2	0x02	STX	Start of Text	正文开始
3	0x03	ETX	End of Text	正文结束
4	0x04	EOT	End of Transmission	传输结束
5	0x05	ENQ	Enquiry	请求
6	0x06	ACK	Acknowledgment	收到通知
7	0x07	BEL	Bell	响铃
8	0x08	BS	Back Space	退格
9	0x09	HT	Horizontal Tab	水平制表符
10	0x0A	LF, NL	Line Feed, New Line	换行键
11	0x0B	VT	Vertical Tab	垂直制表符
12	0x0C	FF, NP	Form Feed, New Page	换页键
13	0x0D	CR	Carriage Return	回车键
14	0x0E	SO	Shift Out	不用切换
15	0x0F	SI	Shift In	启用切换
16	0x10	DLE	Data Line Escape	数据链路转义
17	0x11	DC1	Device Control 1	设备控制1
18	0x12	DC2	Device Control 2	设备控制2
19	0x13	DC3	Device Control 3	设备控制3
20	0x14	DC4	Device Control 4	设备控制4
21	0x15	NAK	Negative Acknowledgement	拒绝接收
22	0x16	SYN	Synchronous Idle	同步空闲
23	0x17	ETB	End of Transmit Block	结束传输块
24	0x18	CAN	Cancel	取消
25	0x19	EM	End of Medium	媒介结束
26	0x1A	SUB	Substitute	代替
27	0x1B	ESC	Escape	换码
28	0x1C	FS	File Separator	文件分隔符
29	0x1D	GS	Group Separator	分组符
30	0x1E	RS	Record Separator	记录分隔符
31	0x1F	US	Unit Separator	单元分隔符
127	0x7F	DEL	Delete	删除

Table 4 Values and pronunciations of the ASCII symbols: **printable characters**

Decimal	Hex	Symbol	Description	Chinese
32	0x20	SP	Space	空格
33	0x21	!	Exclamation mark	叹号
34	0x22	"	Double quotes	双引号
35	0x23	#	Number, sharp	井号
36	0x24	$	Dollar sign	美元符
37	0x25	%	Percent sign	百分号
38	0x26	&	Ampersand	与号
39	0x27	'	Single quote	闭单引号
40	0x28	(	Open parenthesis, (or open bracket)	开括号
41	0x29	)	Close parenthesis, (or close bracket)	闭括号
42	0x2A	*	Asterisk, multiply	星号
43	0x2B	+	Plus	加号
44	0x2C	,	Comma	逗号
45	0x2D	-	Hyphen, minus	减号/破折号
46	0x2E	.	Period, dot	句号
47	0x2F	/	Slash, divide	斜杠
48	0x30	0	Zero	字符0
49	0x31	1	One	字符1
57	0x39	9	Nine	字符9
58	0x3A	:	Colon	冒号
59	0x3B	;	Semicolon	分号
60	0x3C	<	Less than, (or open angled bracket)	小于
61	0x3D	=	Equals	等号
62	0x3E	>	Greater than, (or close angled bracket)	大于
63	0x3F	?	Question mark	问号
64	0x40	@	At symbol	电子邮件符号
65	0x41	A	Uppercase A	大写字母A
66	0x42	B	Uppercase B	大写字母B
90	0x5A	Z	Uppercase Z	大写字母Z
91	0x5B	[	Opening bracket	开方括号
92	0x5C	\	Backslash	反斜杠
93	0x5D	]	Closing bracket	闭方括号
94	0x5E	^	Caret	脱字符
95	0x5F	_	Underscore	下划线
96	0x60	`	Grave accent	开单引号
97	0x61	a	Lowercase a	小写字母a
98	0x62	b	Lowercase b	小写字母b
122	0x7A	z	Lowercase z	小写字母z
123	0x7B	{	Opening brace	开花括号
124	0x7C	|	Vertical bar	垂线
125	0x7D	}	Closing brace	闭花括号
126	0x7E	~	Tilde	波浪号

3. Data types

Data and variables holding data in computer programs are organized in specific ways called *data types*. The textbook uses the data types shown in Tables 5-6.

Usually, we use bool type to define Boolean values and variables, byte (uint8) type to specify characters, unsigned integer (uint64) and signed integers (int) to specify numbers, and float64 to specify real numbers (floating-point numbers).

A variable is defined by a declaration statement such as the following

```
var n int = 100
```

Or, equivalently, the shorter version

```
n := 100
```

Table 5 Basic data types of Golang used in the textbook

Type	Size	Literals	Values	Operations
bool	1 bit	true, false	true, false	&&, ||, !
byte, uint8	1 byte	63, '?'	[0, 255]	+, -, *, /, %;
int	8 bytes	-12345, 69	$[-2^{63}, 2^{63}-1]$	++, --;
uint64	8 bytes	12345	$[0, 2^{64}-1]$	>>, <<; &, |, ^
float64	8 bytes	3.14159	IEEE 754	+, -, *, /

Table 5 gives examples of literals of different types. There are three types of basic notational symbols for values in a program:

- A *literal* is a basic notational symbol for a fixed value.
- A *variable* is a basic notational symbol that can take on one of a set of values.
- A *constant* is a basic notational symbol that can take on one of a set of values, and the value cannot change once taken.

Note that byte is the same as uint8, i.e., 8-bit unsigned integer. It is often used to denote the encoding of an ASCII character. To represents the fixed value of 63 in byte type, which is the ASCII encoding for the question mark '?', we can use the literal 63 or the literal '?'. Also note that the double quote notation is used to denote string literals such as "Alan Turing".

Table 6 Composite data types of Golang used in the textbook

Type	Meaning	Example
array	*n* elements of the same type, e.g., the monthly temperature readings in 2019, which is an array of 12 elements of type byte	var temp [12]byte = [12]byte {3, 4, 15, 19, 27, 31, 32, 29, 28, 18, 9, 3} Or, temp := [12]byte {3, 4, 15, 19, 27, 31, 32, 29, 28, 18, 9, 3}
slice	A structure specifying part of an array, e.g.,	var Q4 []byte = temp[8:12] Or,

	the monthly temperature readings in the 4th Quarter of 2019	Q4 := temp[8:12]
string	An array of characters, e.g., student is an array of 11 characters	var student string = "Alan Turing" Or, student := "Alan Turing"

struct	A structure with named fields, which may be of different types,
	e.g.,
	ST is a structure with two fields: the pioneer's name and whether still alive
		type ST struct {
		name string
		alive bool
		}
		var pioneer = ST{"Alan Turing", false}

For more advanced topics on pointers, maps, and linked lists, please see Sections 4.3.4 and 5.2.2.

Note that in Table 6, the following code

```
type ST struct {
        name    string
        alive   bool
}
var pioneer = ST{"Alan Turing", false}
```

can be shortened to

```
var pioneer = struct {
        name    string
        alive   bool
}{"Alan Turing", false}
```

Similarly, when we discuss binary search in Section 4.3.4, the studentsArray definition is actually

```
var studentsArray = [n] struct {
        key     string
        value   string
}{
  {"Amdahl, Gene", "USA"}, {"Berners-Lee, Tim", "UK"},
  …
  {"Yao, Andrew", "China"},
 }
```

This code is a shortened version of the following code:

```
type Student struct {
{
        key     string
        value   string
}
var studentsArray = [n]Student {
  {"Amdahl, Gene", "USA"}, {"Berners-Lee, Tim", "UK"},
 …
  {"Yao, Andrew", "China"},
}
```

4. Expressions and statements

Programs use expressions and statements to realize digital symbol manipulation.

An *expression* is a combination of one or more notational symbols by zero or more operations. Examples follow.

```
var x, y, z int = 16, 3, 4
var i byte = 4
z = x / y          // z takes on the value 16/3 = 5
z = x % y          // z takes on the value 16%3 = 1
z = x % int(i)     // z takes on the value 16%4 = 0
```

An *assignment* statement, such as any one of the above three "z = …" statements, assigns the value of the right-hand expression to the left-side variable. Thus, we should read "z = x / y" as "z is assigned the value of x dividing by y", not "z is equal to the value of x dividing by y".

Any expression has a type, which means that all its component symbols should have the same type. If not, a type casting operation needs to be used. For instance, the expression "x % i" is illegal, since x and i have different types. We use a type casting operation int(i) to fix this bug.

An expression can use a function calls in the place of a notational symbol. An assignment statement can assign values to multiple variables.

```
_, err := fmt.Scanf("%d", &n)
```

If a left-side variable is of no concern to the program execution, it can be ignored by writing an underscore symbol (also called the **blank identifier**).

A Boolean expression is often used as the *conditional* in an if-then-else statement or a for loop statement. The following code, in program fib.dp.bu-checkInput.go, uses a Boolean expression to check three conditions in a simplified if-then-else statement, and aborts if any of the three abnormal conditions holds.

```
 if (err != nil) || (n < 0) || (n >92) {
            fmt.Println("Wrong input. Program aborts.")
            return
 }
```

Program fib.dp.bu-checkInput.go computes the value of a Fibonacci number by asking the user to enter a natural number between 0 and 92. It then checks the entered number to see if it is indeed a valid integer. If the user enters “six” (err != nil), -90 (n < 0), or 94 (n >92), the program aborts. Otherwise, the program knows that the user has entered an integer n such that $0 \leq n \leq 92$, and proceeds to compute the value of F(n). The complete source code of fib.dp.bu-checkInput.go follows.

```
package main                    // program fib.dp.bu-checkInput.go
import "fmt"
func main() {
 n := 0
 fmt.Printf("Please enter a natural number between 0 and 92: ")
   _,err := fmt.Scanf("%d", &n)                  // n holds user-entered integer
 if (err != nil) || (n < 0) || (n >92) {
            fmt.Println("Wrong input. Program aborts.")
            return
            }
 fmt.Printf("F(%d) = %d\n", n, fibonacci(n))
}
func fibonacci(n int) int {
   a :=0
   b :=1
   for i :=1; i < n+1; i++ {
      a = a + b
      a, b = b, a
   }
   return a
}
```