Go Counting Standard

University of Southern California

Center for Systems and Software Engineering

May , 2018

Revision Sheet

Date	Version	Revision Description	Author

Table of Contents

1. Definitions

1.1. SLOC

Source Lines of Code is a unit used to measure the size of software program. SLOC counts the program source code based on a certain set of rules. SLOC is a key input for estimating project effort and is also used to calculate productivity and other measurements.

1.2. Physical SLOC

One physical SLOC is corresponding to one line starting with the first character and ending by a carriage return or an end-of-file marker of the same line, and which excludes the blank and comment line.

1.3. Logical SLOC

Lines of code intended to measure “statements”, which normally terminate by a semicolon (C/C++, Java, C#) or a carriage return (VB, Assembly), etc. Logical SLOC are not sensitive to format and style conventions, but they are language-dependent.

1.4. Data declaration line or data line

A line that contains declaration of data and used by an assembler or compiler to interpret other elements of the program.

The following lists the Go keywords that denote data declaration lines:

Table 1 - Go Data Keywords

uint8	uint16	uint32	uint64	int8	int16	int32
int64	float32	float64	complex64	complex128	byte	rune
uint	int	uintptr	chan	var	const	interface
string	map	struct	type			

1.5. Compiler directive

A statement that tells the compiler how to compile a program, but not what to compile.

The following table lists the Go keywords that denote compiler directive lines:

Table 2 - Go Compiler Directives

package	import

1.6. Blank line

A physical line of code, which contains any number of white space characters (spaces, tabs, form feed, carriage return, line feed, or their derivatives).

1.7. Comment line

A comment is defined as a string of zero or more characters that follow language-specific comment delimiter. Go comment delimiters are "//" and "/*". A whole comment line may span one line and does not contain any compilable source code. An embedded comment can co-exist with compilable source code on the same physical line. Banners and empty comments are treated as types of comments.

1.8. Executable line of code

A line that contains software instruction executed during runtime and on which a breakpoint can be set in a debugging tool. An instruction can be stated in a simple or compound form.

An executable line of code may contain the following program control statements:

- Selection statements (if, select, switch)
- Iteration statements (for)
- Empty statements (one or more ";")
- Jump statements (return, go, goto, break, continue, exit function)
- Expression statements (function calls, assignment statements, operations, etc.)
- Block statements

An executable line of code may not contain the following statements:

- Compiler directives
- Data declaration (data) lines
- Whole line comments, including empty comments and banners
- Blank lines

1.9. Handling line termination for Golang

The Go language does not need explicit ";" for marking the end of line. A simple "\n" also marks the end of line. The Go compiler adds the semi colon when it is not explicitly added by the user. Following a similar approach, the GoCounter also appends a semi colon after reading every line EXCEPT in the following cases :

- When the line ends with a "{" in case of For/ If loop
- When the line ends with a closing bracket "{"
- When the line ends with comments
- Empty line

2. Checklist for Source Statement Counts

Table 3 - Physical SLOC Counting Rules

Measurement Unit	Order of Precedence	Physical SLOC	Comments
Executable lines	1	One per line	Defined in 1.8
Non-executable lines			
Declaration (Data) Lines	2	One per line	Defined in 1.4
Comments			Defined in 1.7
On their own line	4	Not included	
Embedded	5	Not included	
Banners	6	Not included	
Empty comments	7	Not included	
Blank lines	8	Not included	Defined in 1.6

Table 4 - Logical SLOC Counting Rules

No.	Structure	Order of Precedence	Logical SLOC Rules	Comments
R01	“for” or “if” statement	1	Count once	
R02	Statements ending by a new line character	2	Count once per statement, including empty statement	Semicolons within “for” statement are not counted. Semicolons used with R01 are not counted.
R03	Block delimeters, braces {…}	3	Count once per pair of braces {…} or an opening brace comes after a keyword “else”	Braces used with R01 are not counted. Function definition is counted once since it is followed by {…}.
R04	Compiler Directive	4	Count once per	

			directive	

3. Examples

3.1. Executable lines

3.1.1. Selection Statement

Table 5 - ESS1 (if, else if, else, and nested if statements)

General Example	Specific Example	SLOC Count

if <expression> {
<statements>
}
	if password == "pass" {
	fmt.println("Access Granted")
	}
		1
		1
		0

if <expression> {
<statement>
}
else {
<statement>
}
	if password == "name" {
	fmt.println("Access Granted")
	}
	else {
	fmt.println("Access Denied")
	}
		1
		1
		0
		0
		1
		0

if <expression> {
<statements>
}
else if <expression> {
<statements>
}
else {
<statements>
}
	if num > 0 {
	fmt.println('positive')
	}
	else if num < 0 {
	fmt.println('negative')
	}
	else {
	fmt.println('zero')
	}
		1
		1
		0
		1
		1
		0
		0
		1
		0

if <expression> {
<statements>
<statements>
}
else {
<statements>
}
 	
NOTE: complexity is not considered
	if x < 0 {
	x = 0
	fmt.println('Negative')
	}
	else {
	fmt.println('Positive')
	}
		1
		1
		1
		0
		0
		1
		0

Table 6 – ESS2 (fallthrough, switch and nested switch statements)

General Example	Specific Example	SLOC Count

switch <expression>
{
case <expression 1> :
<statements>
fallthrough
case <expression 2> :
<statements>
case <expression 3> :
<statements>
default :
<statements>
}
	switch (number)
	{
	case 1:
	case 11:
	foo1()
	fallthrough
	case 2:
	foo2()
	case 3:
	foo3()
	default:
	fmt.println (“invalid case”)
	}
		1
		0
		0
		0
		1
		1
		0
		1
		0
		1
		0
		1
		0

Table 7 – ESS3 (select)

General Example	Specific Example	SLOC Count

for <expression> {
select {
case <expression 1> :
<statements>
case <expression 2> :
<statements>
}
}
	for i:=0; i<2; i++ {
	select {
	case msg1 := <-c1:
	fmt.println(“received, msg1)
	case msg2 := <-c2:
	fmt.println(“received, msg2)
	}
	}
		1
		0
		1
		1
		1
		1
		0
		0

3.1.2. Iteration Statement

Table 8 - EIS1 (for loops)

General Example	Specific Example	SLOC Count

for initialization; condition;
increment {
statement
}
 	
NOTE: “for” statement counts as one, no matter how many optional expressions it contains, i.e. for i: = 0, j := 0; i < 5, j < 10; i++, ,j++
	for i := 0; i < 10; i++ {
	fmt.println(i)
	}
		1
		1
		0

Table 9 - EIS2 (empty statements)

General Example	Specific Example	SLOC Count

for i := 0; i < SOME_VALUE; i++ {
 	
}
	for i := 0; i < 10; i++ {
	
	}
		1
		1
		0

3.1.3. Jump Statement

Table 10 - EJS1 (return)

General Example	Specific Example	SLOC Count
return *expression*	return i	1

Table 11 - EJS2 (goto and label statements)

General Example	Specific Example	SLOC Count

goto label	loop1:
	x++
	if x < y {
	goto loop1
	}
		0
		1
		1
		1
		0

Table 12 - EJS3 (break statements)

General Example	Specific Example	SLOC Count

break	if I > 10 {
	break
	}
		1
		1
		0

Table 13 - EJS4 (exit function)

General Example	Specific Example	SLOC Count
func <function_name> {	func main() {	1

os.Exit(<status_code>) }	os.Exit(3) }	1 0

Table 14 - EJS5 (continue statements)

General Example	Specific Example	SLOC Count

continue	for a<20 {
	if a==15 {
	a = a+1
	continue
	}
	a++
	}
		1
		1
		1
		1
		0
		1
		0

3.1.4. Expression Statement

Table 15 - EES1 (function call)

General Example	Specific Example	SLOC Count
<function_name> (<parameters>)	read_file (name)	1

Table 16 - EES2 (assignment statements)

General Example	Specific Example	SLOC Count
<name> = <value>	x = y var a int = 10	1 1

Table 17 - EES3 (empty statement)

General Example	Specific Example	SLOC Count
one or more ";" in succession	; ;;;	1 3

3.1.5. Block Statement

Table 18 - EBS1 (blocks)

General Example	Specific Example	SLOC Count

/* start of block */
{
<definitions>
<statement>
}
/* end of block */
	/* start of block */
	{
	i = 0
	fmt.Printf (“%d”, i)
	}
	/* end of block */
		0
		0
		1
		1
		0
		0

3.1.6. Declaration or Data Lines

Table 19 - DDL1 (function prototype, variable declaration, struct declaration)

General Example	Specific Example	SLOC Count

func <function_name> (
<parameters>) [return types] {
}
	func max(num1, num2 int) int {
	}
		1
		0

var <name> <type>	var age int var price double	1 1

type <struct_name> struct {
<statements>
}
	type myTime struct {
	time.Time
	}
		1
		1
		0

3.1.7. Compiler Directives

Table 20 - CDL1 (directive type)

General Example	Specific Example	SLOC Count
package <package_name> import <package_name>	package main import “fmt”	1 1

4. Complexity

Complexity measures the occurrences of different keywords in code baseline. Below table identifies the categories and their respective keywords that are counted as part of the complexity metrics.

Table 21 - Complexity Keywords List

Math Functions	Trig	Log	Calculations	Conditionals	Logic	Pre-processor	Assignment
math.Abs	math.Cos	math.Log	+	if	==	package	=
math.Cbrt	math.Sin	math.Log10	-	else if	!=	import	
math.Ceil	math.Tan	math.Log1p	*	else	<		
math.Copysign	math.Acos	math.Log2	/	for	>		
math.Dim	math.Asin	math.Logb	%	switch	<=		
math.Erf	math.Atan		|	case	>=		
math.Erfc	math.Atan2		^	select	!		
math.Exp	math.Cosh		++		&&		
math.Exp2	math.Sinh		--		||		
math.Expm1	math.Tanh		>>				
math.Float32bits	math.Acosh		<<				
math.Float32frombits	math.Asinh		&				
math.Float64bits	math.Atanh		//				
math.Float64frombits							
math.Floor							
math.Gamma							

math.Hypot							
math.Ilogb							
math.Inf							
math.J0							
math.J1							
math.Jn							
math.Max							
math.Min							
math.Mod							
math.NaN							
math.Pow							
math.Pow10							
math.Remainder							
math.Sqrt							
math.Trunc							
math.Y0							
math.Y1							

5. Appendix - Differences between UCC and UCC-G

Below are the observations from our testing of UCC-G and comparing the results to UCC v2014.11.

- UCC counts != operation as an assignment operation. UCC-G counts it as a logical operation.
- UCC counts == operation as two assignment operations. UCC-G counts it as a logical operation.
- UCC counts << operation as two logical operations. UCC-G counts it as a left shift calculation operation.
- It appears that UCC is reporting CC2 cyclomatic complexity metrics. UCC-G counts and reports CC1 cyclomatic complexity metrics. UCC-G follows cyclomatic complexity report format of UCC v2015.12 (released by USC).