Scala CodeCount™ Counting Standard

University of Southern California

Center for Systems and Software Engineering

December , 2015

Revision Sheet

Date	Version	Revision Description	Author
12/23/15	1.0	Original Release	Randy Maxwell

Table of Contents

1. Definitions

1.1. **Overview –** Unified Code Count (UCC) has a Procedural Programming perspective when looking at languages and source files, rather than an Object Oriented or Function Programming point of view that Scala supports. Although this difference may cause a "disconnect" from a Scala developer's point of view, it is hope that the metrics gathered by UCC will be of use to the Scala community.

1.2. **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.

1.3. **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.

1.4. **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. A Scala interpretor/compiler infers line endings without requiring the source code to have very many semicolons. A semicolon is needed to separate 2 or more logical source statements on the same physical line, but a line with only 1 logical statement does not require a semicolon. Hence, UCC gives an approximation for Logical SLOC that might not exactly match what is expected.

1.5. **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 table lists the Scala keywords that denote data declaration lines:

abstract	Array	Boolean	Byte	Char
class	Double	extends	Float	HashMap
HashSet	implements	Int	LinkedHashMap	LinkedList
Long	object	override	private	protected
sealed	Short	static	String	TreeMap
val	var	Vector		

Table 1 Data Declaration Types

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

The following table lists the Scala keywords that denote compiler directives:

package	import

Table 2 Compiler Directives

1.7. Executable Keywords – Scala keywords are reserved with predefined characteristics as far as syntax and meanings (semantic or otherwise) to enable various Scala language specific features.

The following table lists the Scala executable keywords:

break	case	catch	def	do
else	finally	for	if	match
new	return	super	this	throw
try	while			

Table 3 Executable Keywords

1.8. 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).

1.9. Comment Line – A comment is defined as a string of zero or more characters that follow language-specific comment delimiter.

Scala comment delimiters are "//" for a single line (until the end of the line) and "/*" for a possible multiline comment. A whole comment line may span one line and does not contain any compliable source code. An embedded comment can co-exist with compliable source code on the same physical line. Banners and empty comments are treated as types of comments. Scala allows nesting of multiline comments to any arbitrary depth, for example:

```
/* This is an outer block comment

/* this is an inner block comment */

now ending the outer block */
```

1.10. 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, match)
  - Iteration statements (for, while, do-while)
  - Empty statements (one or more ";")
  - Jump statements (return, break, exit function)
  - Expression statements (function calls, assignment statements, operations, etc.)
  - Block statements

NOTE: See Section 3 of this document for examples of control 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

2. Checklist for source statement counts

PHYSICAL SLOC COUNTING RULES			
MEASUREMENT UNIT	ORDER OF PRECEDENCE	PHYSICAL SLOC	COMMENTS
Executable Lines	1	One Per line	Defined in 1.10 Keywords in 1.7
Non-executable Lines			
Declaration (Data) lines	2	One per line	Defined in 1.5
Compiler Directives	3	One per line	Defined in 1.6
Comments			Defined in 1.9
On their own lines	4	Not Included (NI)	
Embedded	5	NI	
Banners	6	NI	
Empty Comments	7	NI	
Blank Lines	8	NI	Defined in 1.8

LOGICAL SLOC COUNTING RULES				
NO.	STRUCTURE	ORDER OF PRECEDENCE	LOGICAL SLOC RULES	COMMENTS
R01	"for", "while", "match" or "if" statement	1	Count Once	"while" is an independent statement.
R02	*do {…} while (…); statement*	2	Count Once	Braces {…} and semicolon ; used with this statement are not counted.
R03	Statements ending by a semicolon	3	Count once per statement, including empty statement	Semicolons within "for" statement are not counted. Semicolons used with R01 and R02 are not counted.

R04	Block delimiters, braces {…}	4	Count once per pair of braces {..}, except where a closing brace is followed by a semicolon, i.e. };or an opening brace comes after a keyword “else”.	Braces used with R01 and R02 are not counted. Function definition is counted once since it is followed by {…}.
R05	Compiler Directive	5	Count once per directive	

3. Examples

EXECUTABLE LINES

SELECTION Statement

ESS1 – if, else if, else and nested if statements

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
if (<boolean expression>) <statements>;	if (x != 0) System.out.print (“non-zero”);	1 1

if (<boolean expression>)
  <statements>;
else <statements>;
	if (x > 0)
	  System.out.print (“positive”);
	else
	  System.out.print (“negative”);
		1
		1
		0
		1

if (<boolean expression>)
  <statements>;
else if (<boolean expression>)
  <statements>;.
.
.
else <statements>;
	if (x == 0)
	  System.out.print (“zero”);
	else if (x > 0)
	  System.out.print (“positive”);
	else {
	  System.out.print (“negative”);
	}
		1
		1
		1
		1
		0
		1
		0

NOTE: complexity is not considered, i.e. multiple “&&” or “||” as part of the expression.	if ((x != 0) **&&** (x > 0)) System.out.print (x);	1 1

ESS2 – match and nested match statements

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

match (<expression>)
{
  case <constant 1> :
  <statements>;
  break;
  default
  <statements>;
}
	match (number)
	{
	  case 1:
	  foo1();
	  break;
	  default
	  System.out.print (“invalid case”);
	}
		1
		0
		0
		1
		1
		0
		1
		0

ESS3 – try-catch

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

try
{
  // code that could throw
  // an exception
}
catch (exception-declaration)
{
  // code that executes when
  // exception-declaration is thrown
  // in the try block
}
	try {
	  inputFileName=args[0];
	}
	catch (IOException e) {
	  System.err.println(e);
	  System.exit(1);
	}
		1
		1
		0
		1
		1
		1
		0

ITERATION Statement

EIS1 - for

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++)
	  printf ("%d", i);
		1
		1

EIS2 – empty statements (could be used for time delays – not recommended)

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
for (i = 0; i < SOME_VALUE; i++) ;	for (i = 0; i < 10; i++) ;	2

EIS3 - while

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

while (<boolean expression>)
<statement>;
	while (i < 10)
	{
	  System.out.print (i);
	  i++;
	}
		1
		0
		1
		1
		0

EIS4 – do-while

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

do
{
<statements>;
} while (<boolean expression>);
	do
	{
	ch = getCharacter();
	} while (ch != '\n');
		1
		0
		1
		1

EIS5 – for-each

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
for (<boolean expression>) <statements>;	for (String n: Names) System.out.println(ncharAt(0));	1 1

JUMP Statement

EJS1 - return

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
return expression	if (lineCount==0) return;	1 1

EJS2 - break

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
break;	if (complexity > 10) break;	1 1

EJS3 – exit function

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
void exit (int return_code);	if (x < 0) exit (1);	1 1

EJS4 – continue

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

continue;	while (!done)
	{
	ch = getCharacter();
	if (char == '\n')
	{
	done = true;
	continue;
	}
	}
		1
		0
		1
		1
		0
		1
		1
		0
		0

EXPRESSION Statement

EES1 – function call

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<function_name> (<parameters>);	read_file (name);	1

EES2 – assignment statement

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

<name> = <value>;	x = y;
	char name[6] = "file1";
	a = 1; b = 2; c = 3;
		1
		1
		3

ESS3 – empty statement (is counted as it is considered to be a placeholder for something to call attention)

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
one or more ";" in succession	;	1 per each

BLOCK Statement

EBS1 – block=related statements treated as a unit

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

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

DECLARATION OR DATA LINES

DDL1 – function prototype, variable declaration

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<type> <name> (< parameter_list>);	private static void foo (int param);	1
<type> <name>;	double amount;	1
Class<T>	Iterator<String>	1

COMPILER DIRECTIVES

CDL1 – directive types

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
package <package_name> import <package_name>	package test import scala.math._	1 1

4. Cyclomatic Complexity

Cyclomatic complexity measures the number of linearly independent paths through a program. It is measured for each function, procedure, or method according to each specific program language. This metric indicates the risk of program complexity and also determines the number of independent test required to verify program coverage.

The cyclomatic complexity is computed by counting the number of decisions plus one for the linear path. Decisions are determined by the number of conditional statements in a function. A function without any decisions would have a cyclomatic complexity of one. Each decision such as an if condition or a for loop adds one to the cyclomatic complexity.

The cyclomatic complexity metric v(G) was defined by Thomas McCabe. Several variations are commonly used but are not included in the UCC. The modified cyclomatic complexity counts select blocks as a single decision rather than counting each case. The strict or extended cyclomatic complexity includes boolean operators within conditional statements as additional decisions. Please see: **cyclomatic_complexity_standard.pdf** which has more details of different ways specific cyclomatic complexity metrics are found and presented.

Cyclomatic Complexity	Risk Evaluation
1-10	A simple program, without much risk
11-20	More complex, moderate risk
21-50	Complex, high risk program
> 50	Untestable program, very high risk

For Scala, the following table lists the conditional keywords used to compute cyclomatic complexity.

Scala Statement	CC Count	Rationale
if	+1	if adds a decision
else if	+1	else if adds a decision
else	0	Decision is at the if statement
match / case	+1 per case	Each case adds a decision – not the match
for	+1	for adds a decision at loop start
while	+1	while adds a decision at loop start or at end of do loop
do	0	Decision is at while statement – no decision at unconditional loop
try	0	Decision is at catch statement
catch	+1	catch adds a decision