Compiler Automation Tool

Model Performance Report

Report generated by :

CAT Statistic Studio

NAME	
DESIGNATION	
DATE	

Development Team : Gagandeep Singh, Hargeet Kaur, Amarpreet Singh, Harpreet Kaur

1. INTRODUCTION

Compiler Automation Tool (CAT) is a java based application, a tool for automating the task of compilation process. With CAT, a designer can design compiler for any language using user-friendly graphical interface, by specifying various language specification and then using it to compile source codes showing the intermediate results and performance information. CAT is compiler field specific tool and is useful for anyone related to compiler design, development as well as learning.

This report is the result of user's usage of the Compiler Automation Tool that describes his/her modelling specification regarding the programming language under consideration along with the working and the information generated by him/her. The complete report describes the summary of various actions and results that the user has obtained while using this software. Moreover, this report is software generated report whose structure and format are predefined by the CAT development team.

1.1 HOW WAS THIS REPORT GENERATED ?

As mentioned above this report has been automatically generated by CAT with respect to user specification and usage however, it is not just an outcome of a single click of a button. CAT uses three-tier architecture in which each phase forms a consumer-producer relationship with each other. This strategy not only allow user to model the language but also put it into test and generate statistical information. The below figure describes the basic architecture of CAT :

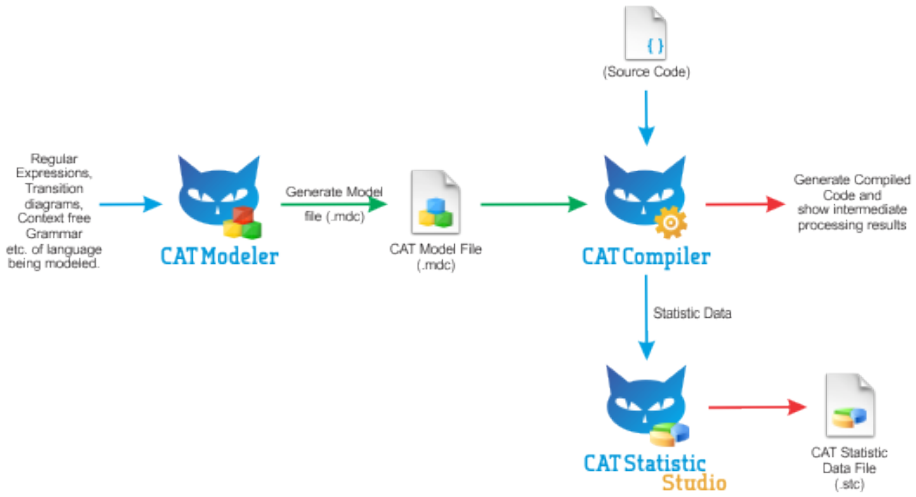


Fig 1.1 - CAT Architecture

Step 1: CAT Modeler - This is the first and most basic step in which the user decides upon a programming language and tries to model it by specifying various language related information such as regular expressions, transition diagram, context free grammar etc. Once finalised, all these information is saved in a CAT Model file with extension .mdc that is fed into the next phase.

Step 2: CAT Compiler - With the model of the language ready, the user may now use it to compile various test source code to verify the correctness of his model. He may choose any arbitrary source code of the extension supported and view intermediate actions of the compiler such as lexical analysis, symbol table etc. Besides this, the user may instruct the CAT Compiler to keep a track of the source codes compiled along with their performance information.

Step 3: CAT Statistic Studio - The information regarding source codes as collected by CAT Compiler is passed to CAT Statistic Studio where the data is analysed and presented in graphical form generating various statistical information useful in judging the performance of the model. The user may further use this application to generate

a report of format .pdf that contains all discussed information. Thus, this report is the result of this application that allows user to summarize his/her work in a well-structured formatted report.

1.2 WHAT DOES THIS REPORT CONTAIN ?

This reports contains rich collection of user usage about the product that ranges from the model he had created to the statistical information. The following report consists of the model properties followed by the environment details in which the compilation occurred, the performance data and statistical analysis including comparisons and analysis of all the phases of compilation.

2. MODEL PROPERTIES

A model in terms of Compiler Automation Tool (CAT) is a set of programming language specifications such as regular expressions, transition diagram, context free grammar etc. corresponding to a particular programming language that user wishes to model and test. Better the specifications, better the compiler performs. Besides, specifications related to the language, there are certain properties that go hand in hand with such specifications, distinguishing it from other models. These may be described in terms of language name, version, source code extensions and many more. These properties are essential part of the model which not only distinguishes them but also provide light over the language that has been modelled into it. The following table describes the properties of the model:

Table 2.1 - Model Properties

Model Attribute	Value
Model file	CppModelv2.mdc
Programming Language	C++
Model Version	2.0
Author	gagandeep Singh
Source Code extension(s)	cpp, h
Description	This model describes the compiler specifications of the programming language C++.

3. COMPILATION ENVIRONMENT

This section describes various attributes of the system on which the compilation was performed using various test source codes. Various system attributes such as memory, speed etc. has significant impact on the performance of the compiler. In fact, better the system, better is the compilation speed and response time. Of course, a system with higher memory and speed configuration is likely to compile and respond much earlier than a system with comparably lower configuration. Thus, it is necessary to specify the system attributes as the part of this report. The following table describes various system attributes on which the compilation were performed by the user:

Table 3.1 - Compilation Environment

System Attribute	Value
User Name	Gagandeep

Computer Name	Desktop-i3
OS Name	Windows 7
OS Version	6.1
System Manufacturer	INTEL_
System Model	DH55TC__
System Type	x86
CPU Model	Core(TM) i3 CPU 540 @ 3.07GHz
CPU Speed	3059
Total CPU Cores	4
CPU Manufacturer	Intel
RAM	3.2G

4. PERFORMANCE DATA

The performance data describes a tabular representation of the data related to the performance of various source codes that were compiled using the model described in above section under specified system environment. This data representation can be regarded analogous to a standard relational database (to some extent) that uses tables with attributes and tuples. As the source code gets compiled, certain information about the performance is generated if carefully observed. This can extend from overall compile time to memory usage as well as to those in individual compiler phases such as lexical, syntax and so on. So, as the source code(s) gets compiled, the user may instruct CAT Compiler to keep track of the performance. The following table describes the performance data collected by the user

Table 4.1 - Performance Data (Total no. of records :42)

S.No	Lines of Code (LOC)	Source Code Path	Lexical Time (ms)	Total Time (ms)
1	465	E:\Test Source Codes (CPP)\ADMIN.CPP	42	42
2	465	E:\Test Source Codes (CPP)\ADMIN.CPP	29	29
3	465	E:\Test Source Codes (CPP)\ADMIN.CPP	27	27
4	465	E:\Test Source Codes (CPP)\ADMIN.CPP	27	27
5	465	E:\Test Source Codes (CPP)\ADMIN.CPP	19	19
6	465	E:\Test Source Codes (CPP)\ADMIN.CPP	17	17
7	1069	E:\Test Source Codes (CPP)\AIEEEC.CPP	45	45
8	1069	E:\Test Source Codes (CPP)\AIEEEC.CPP	35	35
9	1069	E:\Test Source Codes (CPP)\AIEEEC.CPP	41	41
10	1069	E:\Test Source Codes (CPP)\AIEEEC.CPP	39	39
11	1069	E:\Test Source Codes (CPP)\AIEEEC.CPP	38	38
12	1069	E:\Test Source Codes (CPP)\AIEEEC.CPP	40	40

13	711	E:\Test Source Codes (CPP)\Brainteaser.CPP	30	30
14	711	E:\Test Source Codes (CPP)\Brainteaser.CPP	16	16
15	711	E:\Test Source Codes (CPP)\Brainteaser.CPP	24	24
16	711	E:\Test Source Codes (CPP)\Brainteaser.CPP	24	24
17	711	E:\Test Source Codes (CPP)\Brainteaser.CPP	32	32
18	711	E:\Test Source Codes (CPP)\Brainteaser.CPP	28	28
19	659	E:\Test Source Codes (CPP)\PROJECTR.CPP	38	38
20	659	E:\Test Source Codes (CPP)\PROJECTR.CPP	23	23
21	659	E:\Test Source Codes (CPP)\PROJECTR.CPP	31	31
22	659	E:\Test Source Codes (CPP)\PROJECTR.CPP	27	27
23	659	E:\Test Source Codes (CPP)\PROJECTR.CPP	30	30
24	659	E:\Test Source Codes (CPP)\PROJECTR.CPP	28	28
25	1331	E:\Test Source Codes (CPP)\REGIST.CPP	56	56
26	1331	E:\Test Source Codes (CPP)\REGIST.CPP	29	29
27	1331	E:\Test Source Codes (CPP)\REGIST.CPP	46	46
28	1331	E:\Test Source Codes (CPP)\REGIST.CPP	42	42
29	1331	E:\Test Source Codes (CPP)\REGIST.CPP	56	56
30	1331	E:\Test Source Codes (CPP)\REGIST.CPP	47	47
31	241	E:\Test Source Codes (CPP)\SUM.H	31	31
32	241	E:\Test Source Codes (CPP)\SUM.H	23	23
33	241	E:\Test Source Codes (CPP)\SUM.H	18	18
34	241	E:\Test Source Codes (CPP)\SUM.H	16	16
35	241	E:\Test Source Codes (CPP)\SUM.H	18	18
36	241	E:\Test Source Codes (CPP)\SUM.H	13	13
37	346	E:\Test Source Codes (CPP)\VERBAL.H	68	68
38	346	E:\Test Source Codes (CPP)\VERBAL.H	19	19
39	346	E:\Test Source Codes (CPP)\VERBAL.H	24	24
40	346	E:\Test Source Codes (CPP)\VERBAL.H	29	29
41	346	E:\Test Source Codes (CPP)\VERBAL.H	28	28
42	346	E:\Test Source Codes (CPP)\VERBAL.H	35	35

5. STATISTICAL ANALYSIS

The Statistical Analysis is the major part of this report for which all efforts have been made in previous section from modelling to compiling to generating data and finally analysing it. It is here where the performance data is processed and represented in graphical form to describe the efficiency of the model that the user has created. Of course, this directly describes the efficiency of the language that user intend to model. In terms of automation where CAT Compiler describes decisions made by individual phases, CAT Statistic Studio goes one step deeper into details by generating statistics. The whole idea behind statistic is to allow user to choose the best decision regarding the model on the basis of space and time complexity by comparing the data depicted by the graphs generated.

This section has been further divided into several sub sections each focusing on particular phase of compilation for instance lexical analysis, syntax analysis and so on and then finally summing up with compilation in totality.

5.1 LEXICAL ANALYSIS

This subsection of the statistical analysis targets the most basic step of any compiler i.e. the lexical analysis where various types of token are identified and recorded in symbol table. Out of all performance data collected, the information regarding lexical analysis is extracted and presented as follows.

5.1.1 LEXICAL TIME VS LOC GRAGH

This representation describes a graph plotted between lexical time and LOC (lines of Codes). On x-axis are various LOC for which codes were compiled while on y-axis lies the lexical compile time that relates to the average time taken to identify tokens for particular LOC. The purpose of this graph is to represent the average time spend in lexical analysis. As the LOC grows, one expect that the lexical time must also increase. However, it is not always the case. The reasoning regarding such cases can be interpreted from the graph as the slop rises and falls. The graph is as follows:

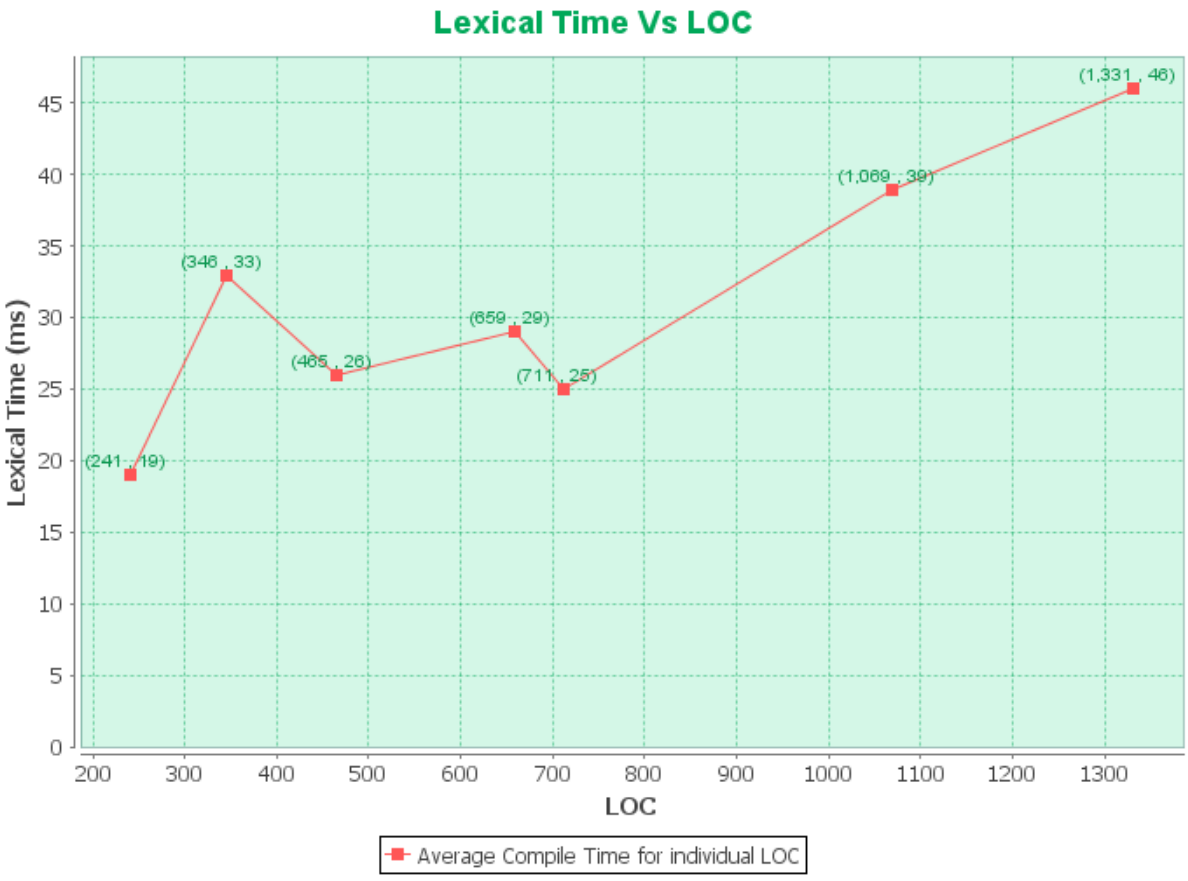
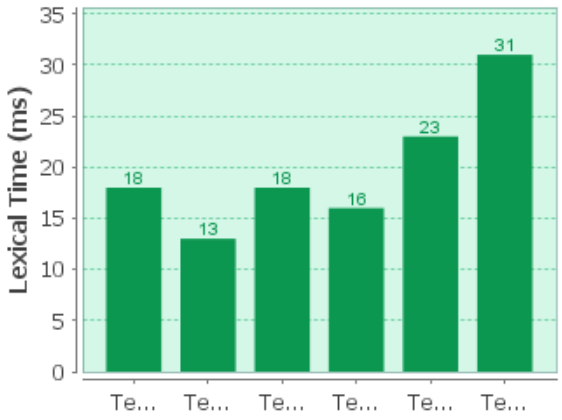


Fig 5.1 - Lexical Time vs LOC graph

5.1.2 LEXICAL TIME GRAGH FOR INDIVIDUAL LOC

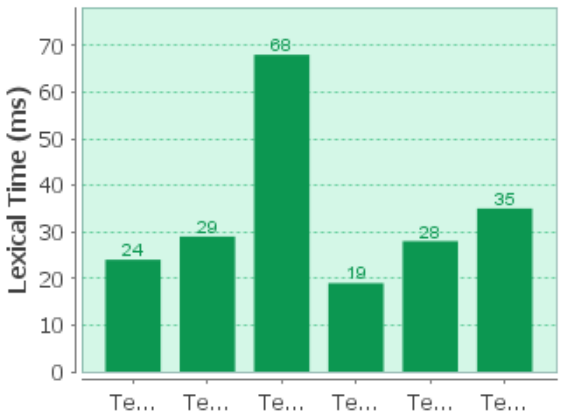
The user may compile a code more than once or perhaps many codes more than once. In such a case it necessary to take the average of the time spend describing the performance corresponding to a particular LOC. This section consider each individual LOC and plots a bar graph of the lexical time spend in each case as well as specifies the average time taken.

Lexical Time for LOC = 241



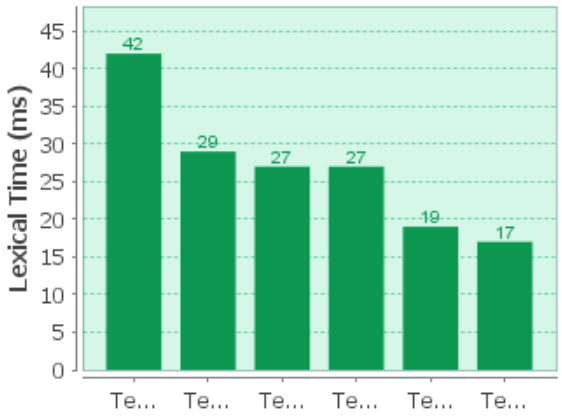
(Average Lexical Time = 19 ms)

Lexical Time for LOC = 346



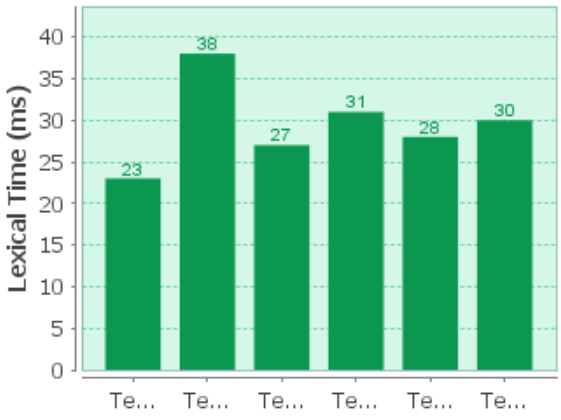
(Average Lexical Time = 33 ms)

Lexical Time for LOC = 465



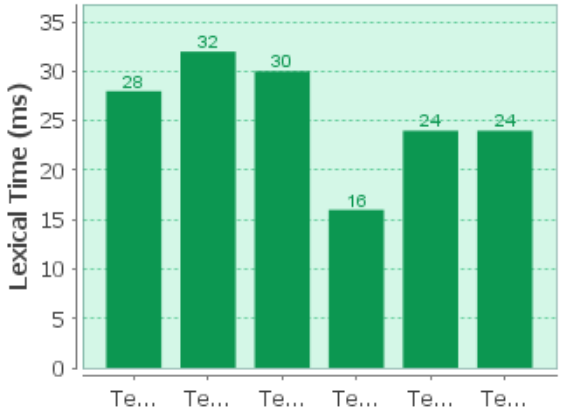
(Average Lexical Time = 26 ms)

Lexical Time for LOC = 659



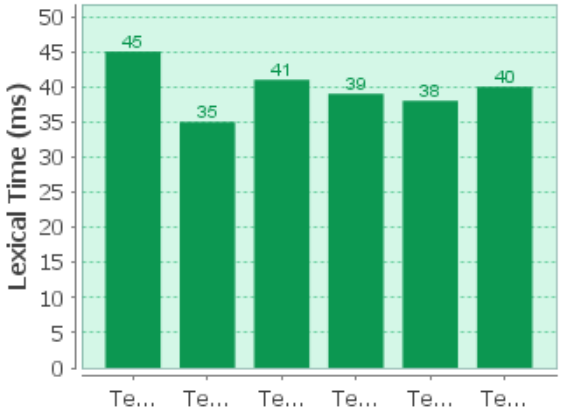
(Average Lexical Time = 29 ms)

Lexical Time for LOC = 711



(Average Lexical Time = 25 ms)

Lexical Time for LOC = 1069



(Average Lexical Time = 39 ms)

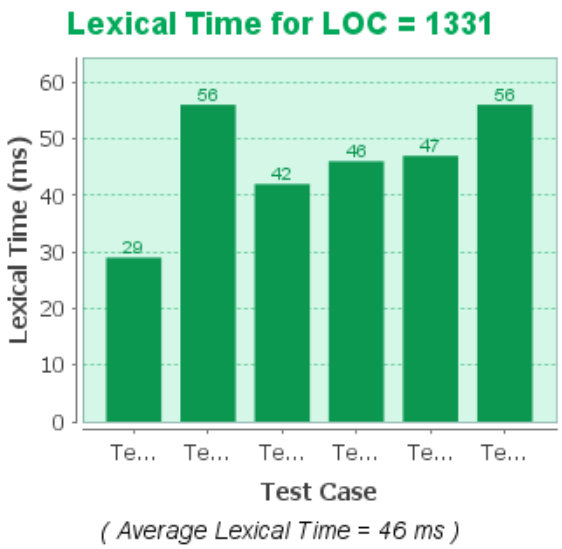


Fig 5.2 - Lexical Time graph for individual LOC

5.2 TOTAL COMPILATION ANALYSIS

This section provides over all statistical information regarding total time required to completely compile source codes covering all phases. This section describes the combined performance of individual phases and is useful in judging the performance of the compilation in totality.

5.2.1 TOTAL COMPILE TIME VS LOC GRAGH

As discussed in previous subsections corresponding to individual phases, this section provides the same information by representing a graph plotted between total compile time and LOC (lines of Codes) rather than just considering each phase. The graph is as follows:

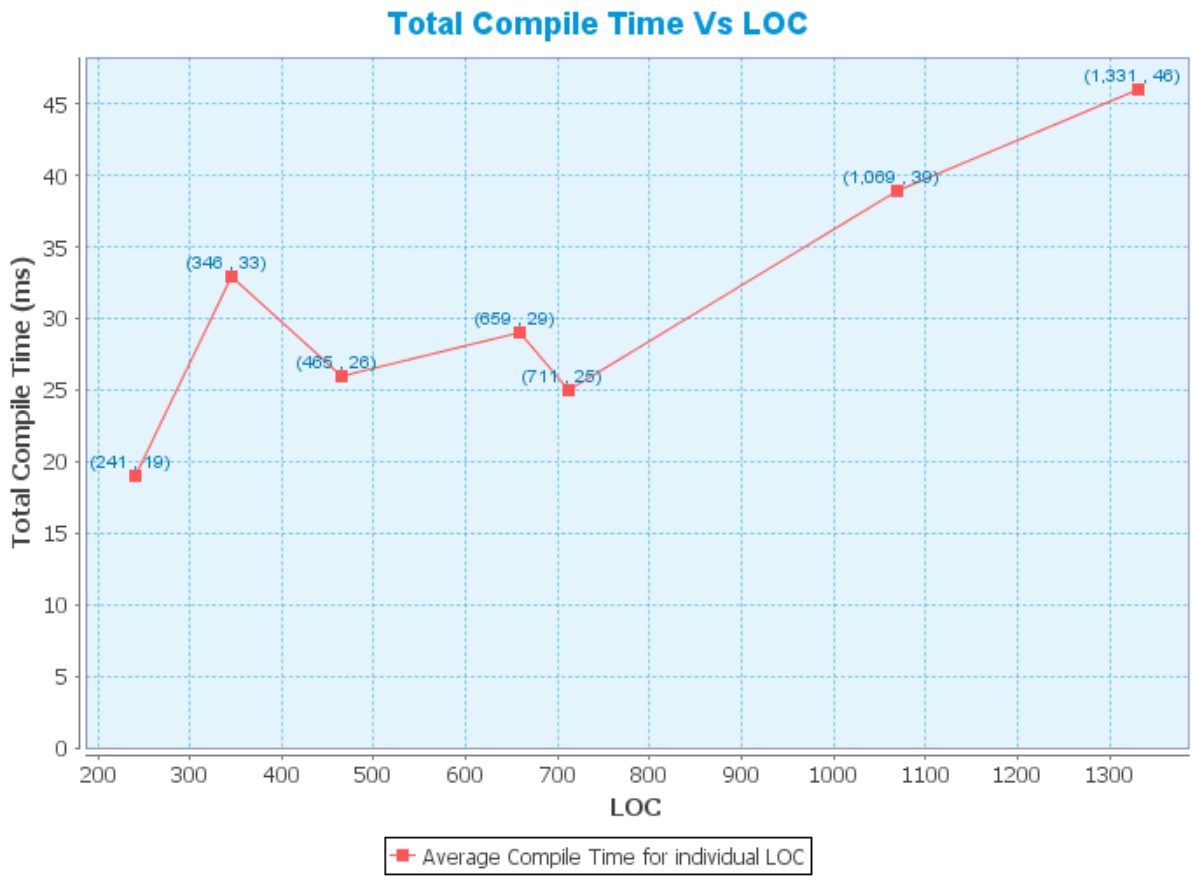
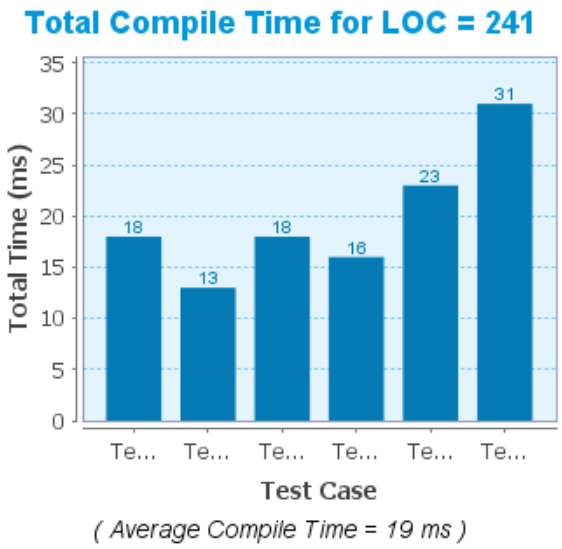


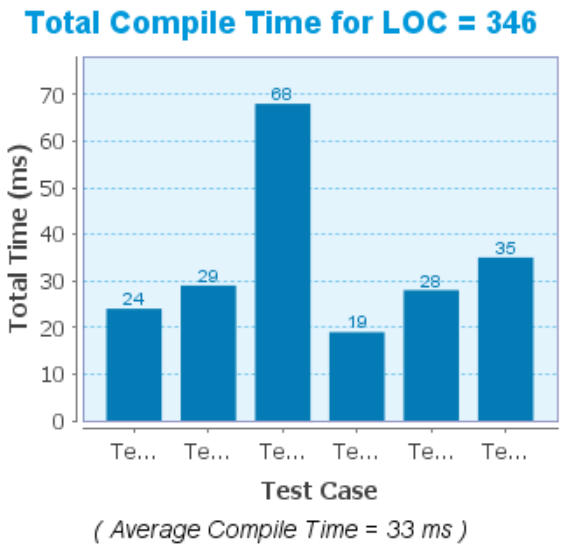
Fig 5.3 - Total Compile Time vs LOC graph

5.2.2 TOTAL COMPILE TIME GRAGH FOR INDIVIDUAL LOC

Below are the bar graphs for individual LOC describing total time taken to compile. The average total time is also specified along with each graph.

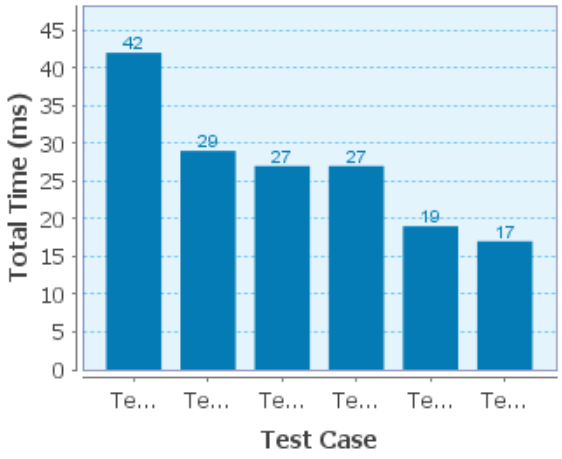


(Average Compile Time = 19 ms)



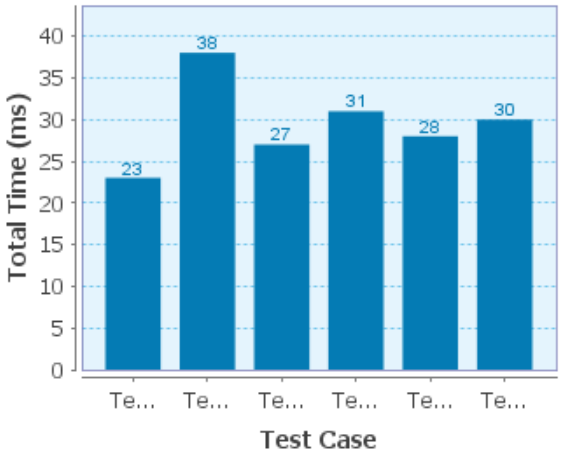
(Average Compile Time = 33 ms)

Total Compile Time for LOC = 465



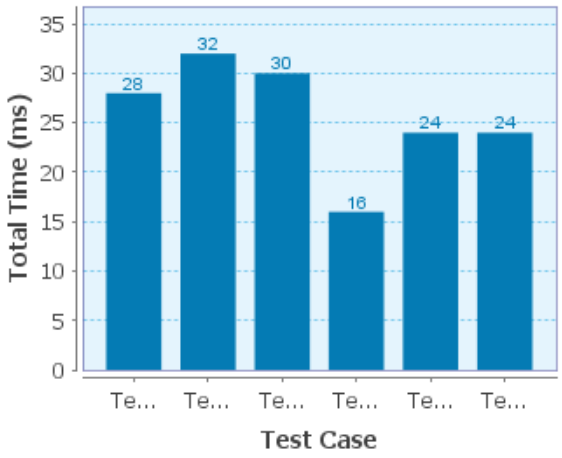
(Average Compile Time = 26 ms)

Total Compile Time for LOC = 659



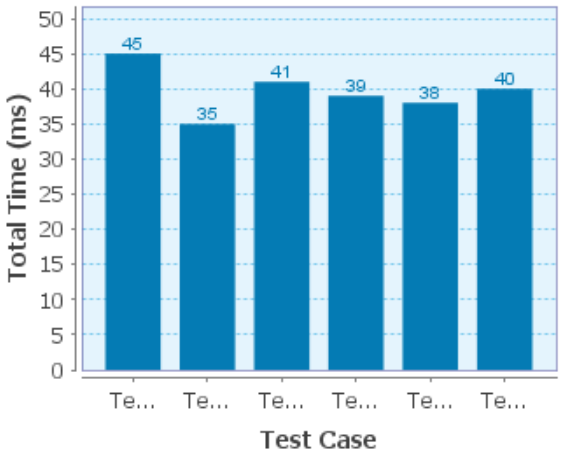
(Average Compile Time = 29 ms)

Total Compile Time for LOC = 711



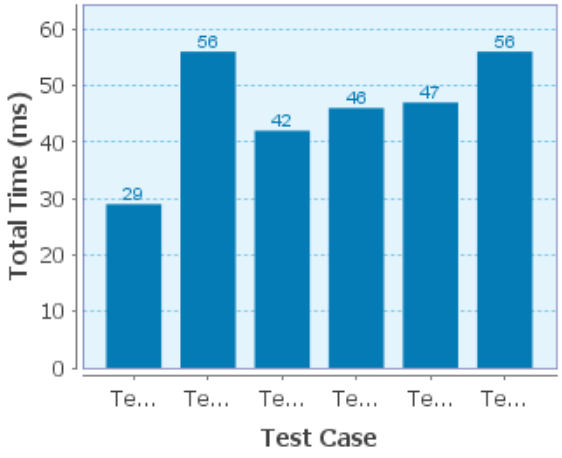
(Average Compile Time = 25 ms)

Total Compile Time for LOC = 1069



(Average Compile Time = 39 ms)

Total Compile Time for LOC = 1331



(Average Compile Time = 46 ms)

Fig 5.4 - Total Compile Time graph for individual LOC