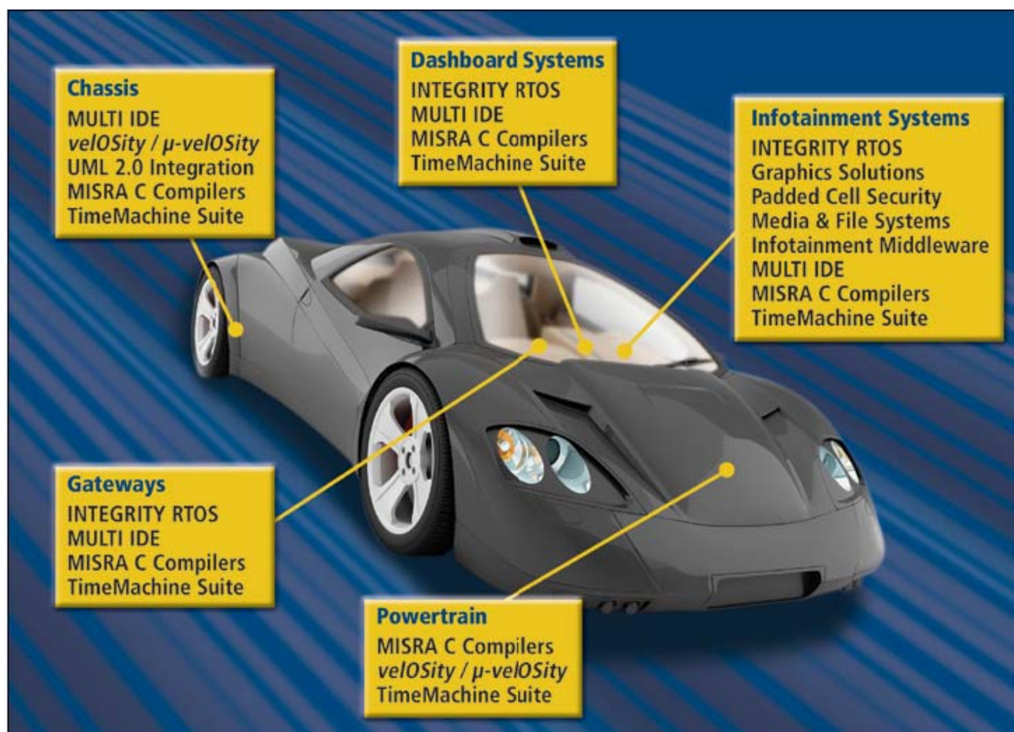
Institute of Technology, Carlow

B.Sc (Honour) in Software Engineering

Research Report

For

MISRA C Code Compliance Checker Project



Student Name: Mingjun Zhou

Student ID: C00094981

Supervisor: Dr. Christophe Meudec

Date: 01/12/2009

Table of Content

1 Introduction 4
2 Background 4
2.1 C Standardization 4
2.2 C Language insecurities [4] 4
2.2.1 The programmer makes mistakes 5
2.2.2 The programmer misunderstands the language 5
2.2.3 The compiler doesn't do what the programmer expects 5
2.2.4 The compiler contains errors 6
2.2.5 Run-time errors 6
2.3 What is MISRA? 6
2.4 What is MISRA C? 7
2.5 Alternatives Languages and Tools 7
2.6 Categories of MISRA C Rules: 8
2.7 Example of Rules: [10] 8
2.8 Version of MISRA-C for using 9
3 Similar tools: 10
3.1 MISRA C Rule Checker SQMlint 10
3.1.1 Overview 10
3.1.2 Message format: 10
3.1.3 Number of Rules can be inspected: 11
3.1.4 MISRA Rules Supported 11
3.1.5 Operating Environment 12
3.1.6 Target Device [16]: 12
3.1.7 Position 12
3.1.8 Report output files 12
3.1.9 How to use 13
3.2 Pc-lint 13
3.2.1 Overview[19] 13
3.2.2 Example of using 14
3.2.3 PC-Lint MISRA-C Rule checking statistics 15
3.2.4 How to integrate into IDE 17
3.2.5 How to use 18
3.3 Conclusion for Similar tools: 19
4 Parsing in general 20
4.1 Why does this project need parsing? 20
4.2 Parsing in English sentence 20
4.3 Parsing technique used in Computer Science field 21
4.4 Parser [22] 22
4.4.1 Lexical analyser [25] 22
4.4.2 Token [25] 22
4.4.3 Types of parsers 23
4.5 Overview of Parsing Process 23

5 Parser generator tools: 23
5.1 Lex/yacc 23
5.1.1 Background 23
5.1.2 How Lex & yacc works 24
5.2 ANTLR 24
5.2.1 Background 25
5.2.2 How ANTLR works 25
5.3 Parser generator choice for project 25
5.4 A C grammar for ANTLR 26
6 Conclusion 27
Reference 28

1 Introduction

The C programming language is used for real-time embedded application field for many reasons. For example, language flexibility, easy associate with hardware, low memory requirements, and so on. However, that's only the one side of C language. On the other side, while the fully experienced programmers enjoying the convenient which C language provided for them, there are also a lot of problems left for those inexperienced programmers who would misunderstand the language, or make mistake when they are doing the program. Typically, for those real-time embedded applications which used in motor industry have considerably more safety-related requirement. If some problems could ever happen, that will cost life. So in this critic situation, there indeed need some standard to guide all the programmers to follow. That's why "MISRA C" [1] was introduced to this issue.

This project is to build up a tool for checking whether the c code following the guideline which provided by MISRA C. It will help the programmer to check their code and also guide them to build up the good coding style.

2 Background

2.1 C Standardization

A standard specification of C was established by the American National Standards Institute (ANSI) committee in 1983. In 1989, after a long time and dramatic hard working, this standard was completed and approved as ANSI X3.159-1989 "Programming Language C." [2] This standard is often referred as "ANSI C" or "C89".

> *"In 1990, the ANSI C standard (with a few minor modifications) was adopted by the International Organization for Standardization as ISO/IEC 9899:1990. This version is sometimes called C90. Therefore, the terms "C89" and "C90" refer to essentially the same language."*[3]

The contents in those two standardizations are identical, but the only slight difference between them is the section numbering.

The MISRA C standard follows the section numbering of the ISO standard.

2.2 C Language insecurities [4]

C programming language is well known as the kind of language that is easy to get into, but hard to control well. Similar with any programming language, there are numerous of problems can happen while using C language. Those problems are categorised below:

2.2.1 The programmer makes mistakes

The mistakes are made by programmers can be as simple as mistyping a variable name, or as complicated as misunderstanding an algorithm.

First, The C language is a very flexible programming language, it gives the widely flexibility and controllability to programmer, thus, program can be either written as well structured and clean, easy-to-understanding code, or written as perverse and extremely hard-to-understanding code. Obviously, the first style code is what people expect to see in a safety-related system.

Second, the syntax of C is easy to make a mistyped mistake be a valid code. For instance, the type in assignment ('=') instead of logical comparison ('= ='), the result is almost always valid but clearly, it's wrong.

Third, theoretically, the principle of C is to assume that programmers would know what they are doing. In other words, if a programmer makes a compatible mistake, and he might not get any warning at all. For example, intends to store a floating-point data type number in an integer for using to represent a Boolean value.

2.2.2 The programmer misunderstands the language

For those inexperienced programmers, there are largely a number of areas in C language that are easily misunderstood. Operator precedence, for instance, is well defined rule in C language, but these rules are still very complicated to fully understand even to an experienced programmer, therefore, there is also a big chance to make wrong assumptions about operator precedence in a particular expression.

2.2.3 The compiler doesn't do what the programmer expects

> "There are many areas of the C language which are not completely defined, and so behaviour may vary from one compiler to another. In some cases the behaviour can vary even within a single compiler, depending on the context. Altogether the C standard, in Annex G, lists 201 issues that may vary in this way. This can present a sizeable problem with the language, particularly when it comes to

portability between compilers. However, in its favour, the C standard does at least list the issues, so they are known."[4]

2.2.4 The compiler contains errors

"Because there are aspects of the C language that are hard to understand, compiler writers have been known to misinterpret the standard and implement it incorrectly. Some areas of the language are more prone to this than others. In addition, compiler writers sometimes consciously choose to vary from the standard."[4]

2.2.5 Run-time errors

"C is generally poor in providing run-time checking. This is one of the reasons why the code generated by C tends to be small and efficient, but there is a price to pay in terms of detecting errors during execution. C compilers generally do not provide run-time checking for such common problems as arithmetic exceptions (e.g. divide by zero), overflow, validity of addresses for pointers, or array bound errors."[4]

2.3 What is MISRA?

Fig.1 MISRA Logo [5]

"MISRA, The Motor Industry Software Reliability Association, is collaboration between vehicle manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety-related electronic systems in road vehicles and other embedded systems."[6]

The MISRA project started in 1990; the mission of this project was "To provide assistance to the automotive industry in the application and creation within vehicle systems of safe and reliable software." [6]

"The original project was part of the UK Government's "SafeIT" programme, but now is self-supported." [7]

There are few examples of MISRA Publications:

- November 1994: Development guidelines for vehicle based software (The MISRA Guidelines)
- April 1998: Guidelines for the use of the C language in vehicle based software (MISRA C)
- October 2004: MISRA-C:2004 –Guidelines for the use of the C language in critical systems(MISRA C2)

2.4 What is MISRA C?

From the section of 2.2, very easily, we notice that a great care should be taken whenever using C as a programming language within safety-related systems. Surely, because of the issues mentioned above, many of concerns have to be involved. Full C language is not suitable for programming safety-related systems.

However, at the same time, C language is sophisticated language. After years and years, it has been well analyzed and fully practiced. Thus, its shortage has been defined and understood well. Furthermore, there are numerous tools available for checking C source code and warning the programmer of the occurrence of problem. There are many constraints rules in those tools for checking whether the c code is suitable for safety-related system. Those constraints rules or called guidelines are often referred as a 'subset of C language'.

MISRA C is one of those guidelines to aid the development of safety related systems in 'C' in the automotive world was produced by MISRA.

> *"As part of these activities, MISRA C was first published in 1998. The intention was to provide a "restricted subset of a standardized structured language" as required in the 1994 MISRA Guidelines for automotive systems being developed to meet the requirements of Safety Integrity Level (SIL) 2 and above."*[8]

So far there are two versions of MISRA C were well published, there are:

MISRA C 1998: there are 127 Rules in total in which included 93 'required' rules and 34 'advisory' rules.

MISRA C 2004: it's a upgrade version of 1998's, there are 141 Rules in total in which included 121 'required' rules, 20 'advisory' rules, and there are also 15 old rules were deleted.

2.5 Alternatives Languages and Tools

Beside MISRA C, there are other languages generally recognised to be more suitable than C. Ada is an example of others.

Ada is designed for highly reliable, real-time and embedded system.

The features of Ada are: [9]

- Run-time error checking
- Program library mechanism
- Object-oriented Programming
- Strong typing
- Generics
- Standard interrupt handling mechanism
- Language Level Tasking/Synchronisation
- Fixed Point Types

2.6 Categories of MISRA C Rules:

There are 17 categories in MISRA C, as following:

- Environment
- Character Sets
- Comments
- Identifiers
- Types
- Constants
- Declarations and Definitions
- Initialisation
- Operators
- Conversions
- Expressions
- Control Flow
- Functions
- Pre-processing Directives
- Pointers and arrays
- Structures and Unions
- Standard Libraries`

2.7 Example of Rules: [10]

Rule 33 (required): The right hand operand of a `&&` or `||` operator shall not contain side effects.

There are some situations in C code where certain parts of expressions may not be evaluated. If these sub-expressions contain side effects then those side effects may or may not occur, depending on the values of other sub expressions. The operators which can lead to this problem are `&&`, `||` and `?:`. In the case of the first two (logical operators) the evaluation of the right-hand operand is conditional on the value of the lefthand operand. In the case of the `?:` operator, either the second or third operands are evaluated but not both. The conditional evaluation of the right hand operand of

one of the logical operators can easily cause problems if the programmer relies on a side effect occurring. The `?:` operator is specifically provided to choose between two sub-expressions, and is therefore less likely to lead to mistakes.

For example:

```
if ( ishigh && ( x == i++ ) )   /* Incorrect */
if ( ishigh && ( x == f(x) ) )  /* Only acceptable if f(x) is known to
                                   have no side effects */
```

Rule 49 (advisory): Tests of a value against zero should be made explicit, unless theoperand is effectively Boolean

Where a data value is to be tested against zero then the test should be made explicit. The exception to this rule is data which is representing a Boolean value, even though in C this will, in practice, be an integer. This rule is in the interests of clarity, and makes clear the distinction between integers and logical values.

For example, if x is an integer, then:

```
if ( x != 0 ) /* Correct way of testing x is non-zero */
if ( x )      /* Incorrect, unless x is effectively Boolean data
              (e.g. a flag) */
```

Rule 50 (required): Floating point variables shall not be tested for exact equality or inequality.

The inherent nature of floating point types is such that comparisons of equality will often not evaluate to true even when they are expected to. In addition the behaviour of such a comparison cannot be predicted before execution, and may well vary from one implementation to another. For example the result of the test in the following code is unpredictable:

```
F_32 x, y;    /* some calculations in here */
if ( x == y )
{ /* ... */ }
```

2.8 Version of MISRA-C for using

Decision on version use for my project will be MISRA-C 1998, and the reasons are pretty simple and clear:

This is academic project so that there is a limitation for studying resources, the MISRA-C 1998 is the only book I can find and downloaded from internet for free. Even thought I have found all the rules for version 2004, for me, there is not much different working on and learning knowledge from doing the old one.

3 Similar tools:

3.1 MISRA C Rule Checker SQMlint

3.1.1 Overview

SQMlint is a software produced by Renesas Technology company. It is a tool to statically check if the C source code compliant MISRA C rules. Example: [11]

```
typedef unsigned short UINT16;
extern volatile UINT16 port1 = 0;
extern volatile UINT16 port2 = 0;
void func(void);
void func(void)
{
    while(port1 != 0) {
        if (port2 == 0) {
            break;
        }
    }
}
```

As example shows above, when a program is inspected by SQMlint, a report message will show down below:

[MISRA(58) Complaining : test.c, 10] 'break' statement shall not be used (except in a 'switch')

3.1.2 Message format:

There are two levels MISRA C Rule:

1. Complaining

When any part of the source code deviates from MISRA C rules

2. Warning

When any part of the source code is likely to deviate from MISRA C rules

Therefore, the message format for those two levels would look like as follow:

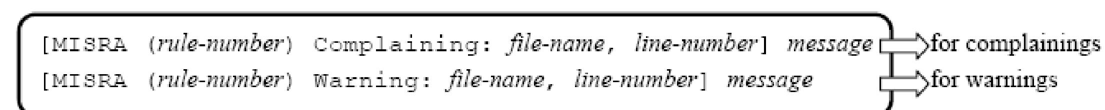


Fig. 2: Format of Report Message [12]

3.1.3 Number of Rules can be inspected:

Rule classification	Number of rules inspected (number of rules inspectable by SQMlint / total number of rules)
Required	67/93
Advisory	19/34
Total	86/127

Fig. 3: Rules can be inspected [13]

3.1.4 MISRA Rules Supported

O: Inspectable; O*: Inspectable, subject to limitations; X: Outside the scope of inspection

Rules	Yes/No	Rules	Yes/No	Rules	Yes/No	Rules	Yes/No	Rules	Yes/No	Rules	Yes/No
1	O	26	X	51	O*	76	O	101	O	126	O
2	X	27	X	52	X	77	O	102	O	127	O
3	X	28	O	53	O	78	O	103	O		
4	X	29	O	54	O*	79	O	104	O		
5	O	30	X	55	O	80	O	105	O		
6	X	31	O	56	O	81	X	106	O*		
7	X	32	O	57	O	82	O	107	X		
8	O	33	O	58	O	83	O	108	O		
9	X	34	O	59	O	84	O	109	X		
10	X	35	O	60	O	85	O	110	O		
11	X	36	O	61	O	86	X	111	O		
12	O	37	O	62	O	87	X	112	O		
13	O	38	O	63	O	88	X	113	O		
14	O	39	O	64	O	89	X	114	X		
15	X	40	O	65	O	90	X	115	O		
16	X	41	X	66	X	91	X	116	X		
17	O*	42	O	67	X	92	X	117	X		
18	O	43	O	68	O	93	X	118	O		
19	O	44	O	69	O	94	X	119	O		
20	O	45	O	70	O*	95	X	120	X		
21	O*	46	O*	71	O	96	X	121	O		
22	O*	47	X	72	O*	97	X	122	O		
23	X	48	O	73	O	98	X	123	O		
24	O	49	O	74	O	99	O	124	O		
25	X	50	O	75	O	100	X	125	O*		

Fig. 4: MISRA C rules supported by SQMlint V.1.03 [14]

3.1.5 Operating Environment

"IBM PC/AT compatibles (Windows Vista®, Windows® XP, Windows® 2000)

SQMlint cannot be used alone; it adds MISRA C rule inspection functions to the Renesas C compiler installed in the computer."[15]

3.1.6 Target Device [16]:

- SuperH RISC engine Family (32-bit RISC)
- M32R Family (32-bit RISC)
- M16C Family (32/16-bit, 16-bit)
- R8C Family (16-bit)
- H8SX Family (32-bit)
- H8S Family (16-bit)
- H8 Family (16-bit, 8-bit)

3.1.7 Position

The SQMlint checks source code before the code processed by compiler. So the code generated by the compiler is unaffected by MISRA C rule checking.

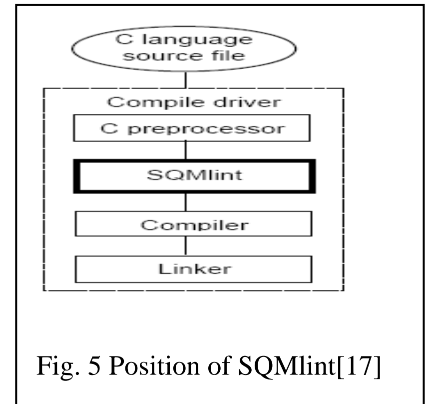


Fig. 5 Position of SQMlint[17]

3.1.8 Report output files

The result of MISRA C rule checking can be output and save into a CSV (Comma Separated Value) format file. And this type of file can be available in most spreadsheet applications.

In the report file, each header for each column is output to the first line. The inspection results will follow line by line. Example of an output report file as below:

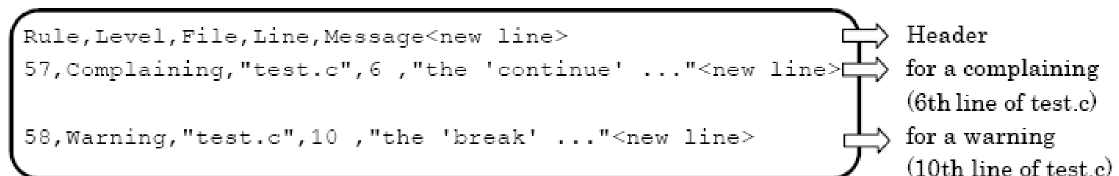


Fig. 6 format of output file [12]

3.1.9 How to use

For different compiler package, the command for SQMlint use will be slightly different. For example, if using C Compiler Package for M16C family, the command line will be like this:

***Example*:**

" *nc30 test.c -c –misra_all –misra_report report.csv*
nc308 test.c -c –misra_all –misra_report report.csv

The explanation for these commands is :
SQMlint inspects the test.c for all of the supported MISRA C rules and outputs the result to report.csv. The test.c also is processed by the compiler." [18]

3.2 Pc-lint

3.2.1 Overview[19]

PC-lint is a static code analysis software tool for C/C++ programming language. It has produced and developed by Gimpel Software since 1985. PC-lint can thoroughly check C/C++ source code for bugs, glitches, in consistencies, non-portable constructs and other sources of problems by using K&R and ANSI standards for C. So the program developer can find and fix the bugs efficiently.

Before continuing introducing this tool, there is little concept, to avoid confusing, has to be mentioned first, there is another name for PC-lint as well which is called FlexeLint. PC-Lint/FlexeLint is the same software package. But the Gimpel Software distributes it in one of two ways.

> *"For the PC market (Windows, MS-DOS, OS/2) the product is distributed in binary executable format. For all other platforms, it is distributed in shrouded C source code format and is known as FlexeLint."*[19]

Version 9.00 is the newest release of PC- lint. Furthermore, PC-lint can be integrated into IDEs as an external tool, and the warning message can be output to a format which IDE can recognize and process, for example, the notepad format will be shown in my description.

In this research report, only PC-lint version 8.00 will be chosen and described. Because it's a commercial product and this version is the one found online for free. The steps for how to integrate it into Borland C++ builder 6.0(BCB) and how to use it will be described later.

3.2.2 Example of using

The example program used here is called BADCODE.C from website.

```
/*-----------------------------------------------------------
BADCODE.C

Copyright Keil - An ARM Company

This source file is full of errors.  You may use µVision
to compile and correct errors in this file.
-----------------------------------------------------------*/

#incldue <stdio.h>

void main (void, void)
{
unsigned i;
long fellow;

fellow = 0;

fer (i = 0; i < 1000; i++)
    {
    printf ("I is %u\n", i);

    fellow += i;
    printf ("Fellow = %ld\n, fellow);
    printf ("End of loop\n")
    }
}
```

Fig. 7: example source code from web [20]

The red colour highlight shows the coding mistakes. First the output with 12 errors from C51 complier will be shown:

```
C51 COMPILER
Copyright (c) KEIL - An ARM Company.  All rights reserved.

*** ERROR 315 IN LINE 10 OF BADCODE.C:
     unknown #directive 'incldue'
*** ERROR 159 IN LINE 12 OF BADCODE.C:
     'typelist': type follows void
*** WARNING 206 IN LINE 19 OF BADCODE.C:
     'fer': missing function-prototype
*** ERROR 267 IN LINE 19 OF BADCODE.C:
     'fer': requires ANSI-style prototype
*** ERROR 141 IN LINE 19 OF BADCODE.C:
     syntax error near ';'
*** ERROR 141 IN LINE 19 OF BADCODE.C:
     syntax error near '000'
*** ERROR 202 IN LINE 19 OF BADCODE.C:
     '000': undefined identifier
*** ERROR 141 IN LINE 19 OF BADCODE.C:
     syntax error near ')'
*** WARNING 206 IN LINE 21 OF BADCODE.C:
     'printf': missing function-prototype
*** ERROR 103 IN LINE 24 OF BADCODE.C:
     '<string>': unclosed string
*** ERROR 305 IN LINE 24 OF BADCODE.C:
     unterminated string/char const
*** ERROR 141 IN LINE 25 OF BADCODE.C:
     syntax error near 'printf'

C51 COMPILATION COMPLETE.  2 WARNING(S),  10 ERROR(S)
```

Fig. 8 the error report detected by C51 compiler [20]

Then, the output with 26 errors from PC-Lint will be shown as follow:

```
--- Module:        badcode.c
badcode.c     10     Error  16:
                            Unrecognized  name
badcode.c     10     Error  10:
                            Expecting  end  of  line
badcode.c     12     Error  66:
                            Bad  type
badcode.c     12     Error  66:
                            Bad  type
badcode.c     19     Info  718:
                            fer  undeclared,  assumed  to  return  int
badcode.c     19     Info  746:
                            call  to  fer  not  made  in  the  presence  of
                            a  prototype
badcode.c     19     Error  10:
                            Expecting  ','
badcode.c     19     Error  26:
                            Expected  an  expression,  found  ';'
badcode.c     19     Warning  522:
                            Expected  void  type,  assignment,
                            increment  or  decrement
badcode.c     19     Error  10:
                            Expecting  ';'
badcode.c     19     Error  10:
                            Expecting  ';'
badcode.c     21     Info  718:
                            printf  undeclared,  assumed  to  return  int
badcode.c     21     Info  746:
                            call  to  printf  not  made  in  the  presence  of
                            a  prototype
badcode.c     23     Info  737:
                            Loss  of  sign  in  promotion  from  long  to
                            unsigned  long
badcode.c     23     Info  713:
                            Loss  of  precision  (assignment)  (unsigned
                            long  to  long)
badcode.c     24     Error  2:
                            Unclosed  Quote
badcode.c     25     Error  10:
                            Expecting  ','
badcode.c     26     Error  10:
                            Expecting  ','
badcode.c     26     Error  26:
                            Expected  an  expression,  found  '}'
badcode.c     26     Warning  559:
                            Size  of  argument  no.  2  inconsistent  with
                            format
badcode.c     26     Warning  516:
                            printf  has  arg.  type  conflict  (arg.  no.  2
                            --  pointer  vs.  unsigned  int)  with  line  21
badcode.c     27     Warning  550:  fellow  (line  15)  not  accessed

---  Global  Wrap-up

Warning  526:  printf  (line  21,  file  badcode.c)  not  defined
Warning  628:  no  argument  information  provided  for  function
                    printf  (line  21,  file  badcode.c)
Warning  526:  fer  (line  19,  file  badcode.c)  not  defined
Warning  628:  no  argument  information  provided  for  function
                    fer  (line  19,  file  badcode.c)
```

Fig. 9 the errors and warning message detected by PC –Lint [20]

As shown, the PC-Lint can detect more errors than normal C compiler.

3.2.3 PC-Lint MISRA-C Rule checking statistics

	# of	total	percentage
Required	76 of	93	82%
Advisory	18 of	34	53%
Total	**94 of**	**127**	**74%**

Fig. 10 PC-Lint MISRA-C Rule checking statistics [19]

The table above shows that, up to December 2001, PC-lint can detect 74% overall and 82% of the “Required” rules. There are 12 rules are not detected by PC-Lint or likely the compiler that will have to be checked by manually in code review. The following are the compliance Matrix for using PC-Lint.

MISRA	Required	Tools			Tool	REQUIRED
Rule	Advisory	PC-Lint	Compiler	Manual	checked	NOT CHECKED
1	R	Yes			Yes	
2	A			YES	NO	
3	A	Yes +			Yes	*+
4	A			YES	NO	
5	R	Yes			Yes	
6	R	*****	YES		YES	
7	R	Yes			Yes	
8	R	*****	YES		YES	
9	R	Yes			Yes	
10	A			YES	NO	
11	R	Yes			Yes	
12	A	Yes			Yes	
13	A	Yes			Yes	
14	R	Yes			Yes	
15	A			YES	NO	
16	R			YES	NO	*****
17	R	Yes			Yes	
18	A	Yes			Yes	
19	R			YES	NO	*****
20	R	Yes			Yes	
21	R	Yes +			Yes	*+
22	A	Yes			Yes	
23	A	Yes			Yes	
24	R	Yes			Yes	
25	R	Yes			Yes	

Fig. 11 Rule 1-25[19]

MISRA	Required	Tools			Tool	REQUIRED
Rule	Advisory	PC-Lint	Compiler	Manual	checked	NOT CHECKED
26	R	Yes			Yes	
27	A	Yes +			Yes	
28	A	Yes +			Yes	
29	R	Yes			Yes	
30	R	Yes			Yes	
31	R	Yes			Yes	
32	R	Yes +			Yes	*+
33	R	Yes +			Yes	*+
34	R			YES	NO	*****
35	R	Yes			Yes	
36	A			YES	NO	
37	R	Yes			Yes	
38	R	Yes			Yes	
39	R	Yes			Yes	
40	A	Yes +			Yes	
41	A			YES	NO	
42	R	Yes			Yes	
43	R	Yes			Yes	
44	A	Yes +			Yes	
45	R	Yes			Yes	
46	Y	Yes			Yes	
47	A	Yes +			Yes	
48	A	Yes +			Yes	
49	A	Yes +			Yes	
50	R	Yes			Yes	

Fig. 12 Rule 26-50[19]

MISRA	Required	Tools			Tool	REQUIRED
Rule	Advisory	PC-Lint	Compiler	Manual	checked	NOT CHECKED
51	A	Yes			Yes	
52	R	Yes			Yes	
53	R	Yes			Yes	
54	R	Yes			Yes	
55	A	Yes +			Yes	
56	R	Yes			Yes	
57	R	Yes +			Yes	*+
58	R	Yes +			Yes	*+
59	R	Yes +			Yes	*+
60	A	Yes +			Yes	
61	R	Yes			Yes	
62	R	Yes			Yes	
63	A	Yes +			Yes	
64	R	Yes			Yes	
65	R	Yes +			Yes	*+
66	A			YES	NO	
67	A			YES	NO	
68	R	Yes +			Yes	*+
69	R	Yes			Yes	
70	R		Note 1			*****
71	R	Yes			Yes	
72	R	Yes			Yes	
73	R	Yes +			Yes	*+
74	R					*****
75	R	Yes			Yes	

Fig. 13 Rule 51-75 [19]

MISRA	Required	Tools			Tool	REQUIRED
Rule	Advisory	PC-Lint	Compiler	Manual	checked	NOT CHECKED
76	R	Yes +			Yes	*+
77	R	Yes			Yes	
78	R	Yes			Yes	
79	R	Yes			Yes	
80	R	Yes			Yes	
81	A			YES	NO	
82	A			YES	NO	
83	R	Yes			Yes	
84	A	Yes			Yes	
85	A			YES	NO	
86	A			YES	NO	
87	R	Yes +			Yes	*+
88	R	Yes			Yes	
89	R	Yes			Yes	
90	R			YES	NO	*****
91	R	Yes +			Yes	*+
92	A	Yes			Yes	
93	A			YES	NO	
94	R	Yes			Yes	
95	R	Yes			Yes	
96	R	Yes			Yes	
97	A	Yes			Yes	
98	A	Yes +			Yes	*+
99	R			YES	NO	*****
100	R	Yes +			Yes	*+

Fig. 14 Rule 76-100[19]

MISRA	Required	Tools			Tool	REQUIRED
Rule	Advisory	PC-Lint	Compiler	Manual	checked	NOT CHECKED
101	A			YES	NO	
102	A			YES	NO	
103	R	Yes			Yes	
104	R			YES	NO	*****
105	R			YES	NO	*****
106	R	Yes			Yes	
107	R	Yes			Yes	
108	R	Yes			Yes	
109	R		YES	YES	YES	*****
110	R	Yes +			Yes	*+
111	R	Yes			Yes	
112	R	Yes			Yes	
113	R			YES	NO	*****
114	R	Yes			Yes	
115	R			YES	NO	*****
116	R			YES	NO	*****
117	R	Yes *++			Yes *++	*++
118	R	Yes			Yes	
119	R	Yes			Yes	*+
120	R	Yes			Yes	*+
121	R	Yes			Yes	
122	R	Yes			Yes	
123	R	Yes			Yes	
124	R	Yes			Yes	
125	R	Yes			Yes	
126	R	Yes			Yes	
127	R	Yes			Yes	

Fig. 15 Rule 101-127[19]

Note:

"Yes++ can be checked but requires specific setting up with Pc-Lint

Yes+ == no Specific MISRA-Rule Message

****** == REQUIRED RULE Not checked by PC-Lint Required manual checking"* [19]

3.2.4 How to integrate into IDE

There are many IDEs that PC-lint can be integrated, for example, Visual Studio 2005/2008, Visual C, etc. I only tried it in Borland C++6.0 successfully, procedures show as follow:

1. Download PC-lint 8.0 and unzip it to local directory, for example, C:\Lint. This can be change, but the BCBLint .bat file need to be modified correspondingly.
2. Open BCB 6.0 and choose Tools→Configure Tools and choose Add in the popup window.

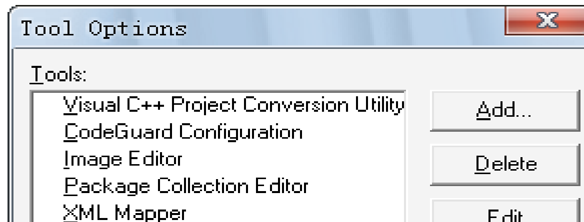


Fig. 16 Configure Tools

3. Enter follow information:

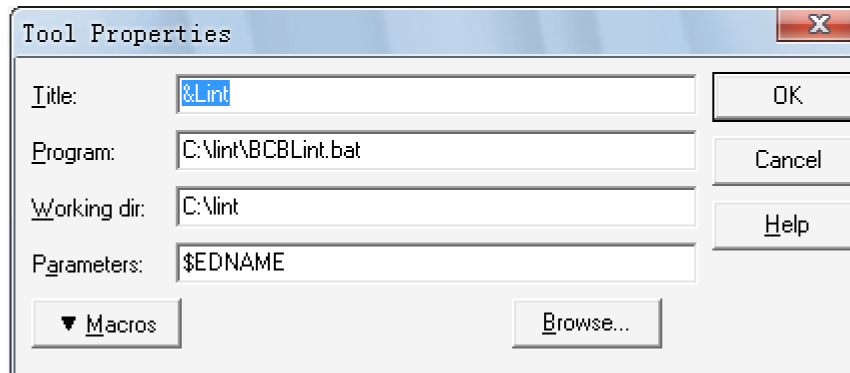


Fig. 17 Enter Information

Title: &Lint
Program: C:\Lint\BCBLint.bat
Working Dir: E:\Lint
Parameters: $EDNAME

4. Click OK, and Close this window.
5. The Lint Tool will appear in the Tools options

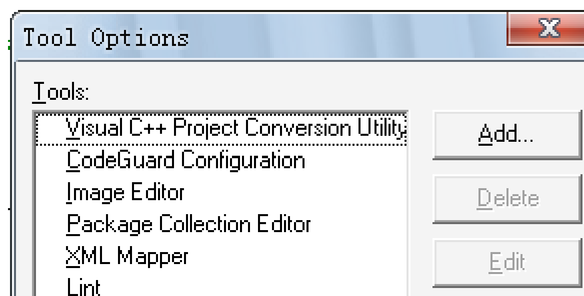


Fig .18 Added tool appear in the Tools option

6. The std.lnt file in C:\Lint\lnt directory need to be modified to as follow:

```
// Borland C++ Builder, -si4 -sp4, lib-owl5.lnt lib-w32.lnt
// Standard lint options
au-misra.lnt //standard Misra C lint option
co-cb.lnt
lib-ow15.lnt lib-w32.lnt
options.lnt -d_FLAT_ -u_SMALL_ -si4 -sp4
// depends on the local file directory
-i C:\Program Files\Borland\CBuilder6\Include;
-i C:\Program Files\Borland\CBuilder6\Include\Vcl;
-i C:\Program Files\Borland\CBuilder6\Imports;
-i C:\Program Files\Borland\CBuilder6\Projects;
//end
```

7. The contents in BCBLint.bat file is:

```
Lint-nt -i"c:\Lint\lnt" std.lnt -os(D:\errorlog.txt) -w2 -u %1
//error messages output to a new created notepad file.
C:\windows\notepad.exe "d:\errorlog.txt"
```

3.2.5 How to use

After above procedure, the “how to use” is turned to pretty straightforward and simple. Open a testing project or a C/C++ file.

1. Choose Lint from Tools menu.

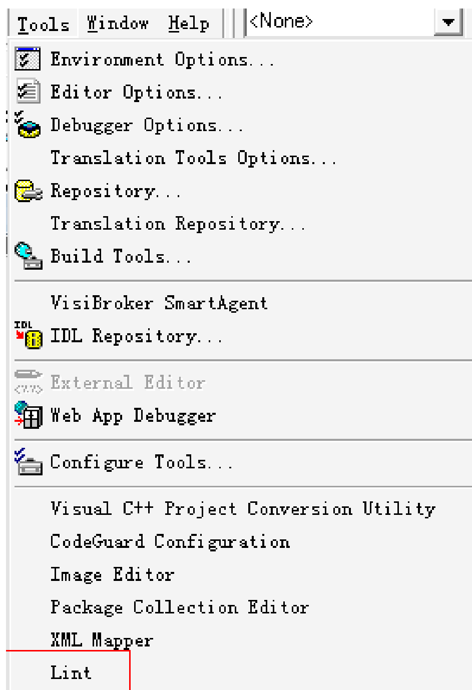


Fig. 19 Location of new added Lint tool

2. The Lint will be executed and message file will be created and display on the screen. The error message from the example program Fig. 7 as follow:

```
--- Module:   C:\badcode.c

#incldue <stdio.h>
C:\badcode.c  11  Error 16: Unrecognized name
C:\badcode.c  11  Error 10: Expecting end of line

void main (void, void)
C:\badcode.c  13  Error 66: Bad type
C:\badcode.c  13  Error 66: Bad type

fer (i = 0; i < 1000; i++)
C:\badcode.c  20  Error 10: Expecting ','
C:\badcode.c  20  Error 26: Expected an expression, found ';'
C:\badcode.c  20  Warning 522: Expected void type, assignment, increment or decrement [MISRA Rule 53]
C:\badcode.c  20  Error 10: Expecting ';'
C:\badcode.c  20  Error 10: Expecting ';'

  printf ("Fellow = %ld\n, fellow);
C:\badcode.c  25  Error 2: Unclosed Quote

  printf ("End of loop\n")
C:\badcode.c  26  Error 10: Expecting ','

  }
C:\badcode.c  27  Error 10: Expecting ','
C:\badcode.c  27  Error 26: Expected an expression, found '}'
C:\badcode.c  27  Warning 626: argument no. 2 inconsistent with format
C:\badcode.c  27  Warning 516: 'Symbol printf()' has arg. type conflict (arg. no. 2 -- basic) with line 22 [MISRA Rule 72]

}
C:\badcode.c  28  Warning 550: Symbol 'fellow' (line 16) not accessed
```

Fig. 20 error message from tested BADCODE.C code

Depending on the option we use here, the output information would look different compare with the original example. The MISRA C rule has been included and highlighted in my test

The modification can be done by changing the “options.lnt”, in order to ignore some error detect. The “-w2” options as show in the “BCBLint.bat” means only display error and warning message.

Note, there is a “PC-Lint.pdf” file in the folder that shows all information in fully details.

3.3 Conclusion for Similar tools:

Since the limitation of time, I couldn’t study those tools very well. Base on what I have done, I got conclusion for those tools.

- I couldn’t try SQMlint since it’s a commercial product and there is no free release. But I found PC- lint and tried it.
- From the MISRA C supported point of view, 94 compare with 86 in totals, PC-lint support more Rules than SQMlint.
- For the message format, since MISRA C rule checker SQMLint is a specific tool made for MISRA C, so its message is more clear, detailed and easy to read. PC-Lint, however, it provides more other error checking besides MISRA C, so its format is very simple and unclear.
- It seems like once PC-lint is integrated into IDEs, it would be very easy to implement. But, the configuration is a little trick and difficult, user has to spend more time to study. Furthermore, PC-lint can be executed in command line. This

is similar with SQMlint.

- The SQMlint has to be used with those compliers which already mentioned earlier, but PC-lint doesn’t care about the type of compliers. It checks source code statically.

From those two tools I have learned something useful for my own project:

My tool will concentrate on MISRA C-1998 rules. It will provide easy approach to execute the C source code in command line format, and provide friendly and readable feedback message. Ideally, like PC-lint, it can be integrated into other IDEs.

4 Parsing in general

4.1 Why does this project need parsing?

The reason why the parsing technology will be involved in this project is that the input C source code needs to be analyses and checked whether it compliance the MISRA C 1998 Guidelines or not. For achieving this goal the existing C grammar must be properly parsed and the new rules for MISRA C rules grammar for my target tool need to be made in order to parsing the input source code. Then, depending on the result of parsing, the software can compare each line of the source code and see if there is any code against the MISRA C guideline. The software should know what it going to check, and what exactly the rules it is going to follow. What is right coding and what is wrong. Finally, it should give correct feedback to the people whoever using this tool.

4.2 Parsing in English sentence

Parsing is, in general speaking, a process that analysis the structure of the statements in a human or artificial language depending on the set of grammars which defines possible structures. The diagram blow shows the structure of the sentence “The dog chased the black cat.”

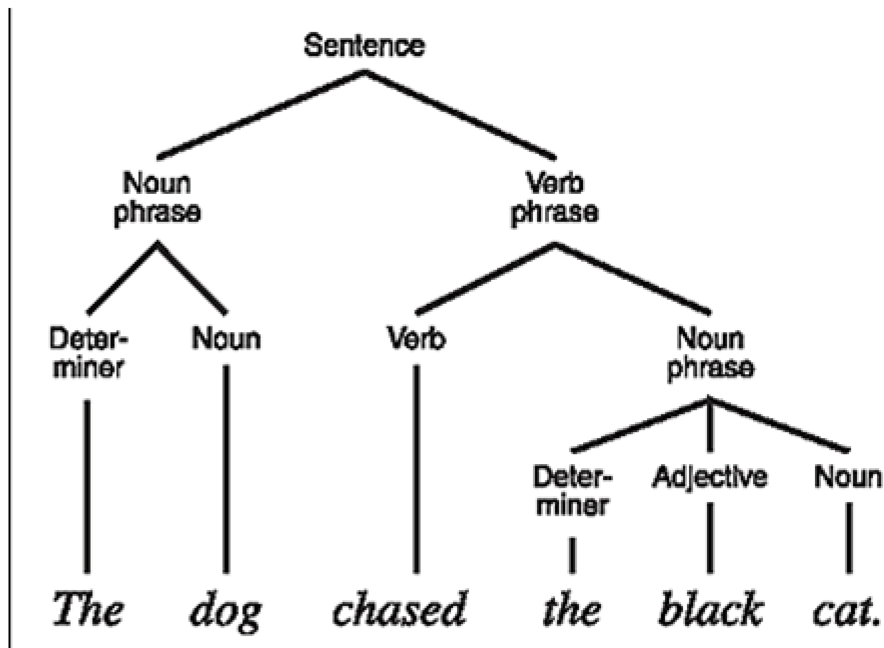


Fig. 7 Parsing: Structure of a sentence [21]

4.3 Parsing technique used in Computer Science field

"In computer science and linguistics, parsing, or more formally, syntactic or grammatical analysis, is the process of analysing a text, made of the sequence of tokens (for example, words), to determine its grammatical structure based on a given formal grammar." [22]

Parsing can be done either top-down or bottom up.

- Top-down parsing

  "Top-down parsing is a strategy of analysing unknown data relationships by hypothesising general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. It occurs in the analysis of both natural languages and computer languages.

  Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse-trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules." [23]

- Bottom-up parsing

"Bottom-up parsing (also known as shift-reduce parsing) is a strategy for analyzing unknown data relationships that attempts to identify the most fundamental units first, and then to infer higher-order structures from them. It attempts to build trees upward toward the start symbol. It occurs in the analysis of both natural languages and computer languages."[24]

4.4 Parser [22]

In computer science, a parser is a component of either an interpreter or a compiler, that process a set of texts as input and identifies the structure of these texts depending on a given grammar so that can extract information.

The parser generally creates tokens from the sequence of input text by using a separate lexical analyser.

"Parsers may be programmed by hand or may be (semi-)automatically generated (in some programming languages) by a tool (such as Yacc) from a grammar written in Backus-Naur form."[22]

4.4.1 Lexical analyser [25]

Lexical analyser, or called lexers, is a program that performs lexical analysis which is the process of converting a sequence of characters into a sequence of tokens.

4.4.2 Token [25]

A token, also as a lexeme, is a sequence of characters that can be considered as a unit according to function which gives them meaning; it is either a word or an operator.

For example, the following C programming language expression:

Sum =4+5;

It can be tokenised into a following table:

Lexeme	Token type
Sum	Identifier
=	Assignment operator
4	Number
+	Addition operator
5	Number
;	End of statement

The Tokens in a Language are frequently defined by regular expressions which are understood by a lexical analyser generator such as lex. A regular expression specifies a set of strings to be matched. It contains text characters and operator characters.

4.4.3 Types of parsers

Same as types of parsing techniques, there also are two types of parsers:

- Top-down parsers (LL Parser)
- Bottom-up parsers(LR Parser)

4.5 Overview of Parsing Process

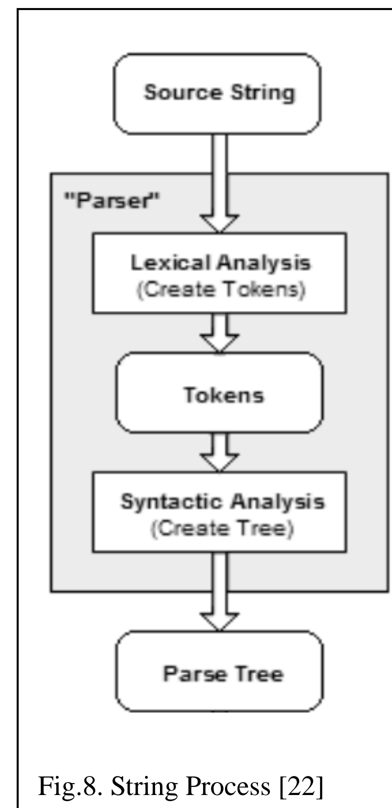


Fig.8. String Process [22]

For computer language parsing, there are commonly two levels of grammar: lexical and syntactic.

The first stage is doing lexical analysis, which is also called the token generation, by splitting the input character stream into meaningful symbols defined by a grammar of regular expressions.

The second stage is parsing or syntactic analysis which is checking whether the tokens form an allowable expression or not. And create a tree for those tokens.

The final stage, the token tree will go to a Parse tree which will optionally output text. [22]

5 Parser generator tools:

5.1 Lex/yacc

5.1.1 Background

"Lex and yacc were both developed at Bell Laboratories in the 1970s. Yacc was the first of the two, developed by Stephen C. Johnson. Lex was designed by Muike Lesk and Eric Schmidt to work with yacc." [26]

Lex refers to Lexical Analysis.

Yacc refers to Yet another Compiler Compiler.

5.1.2 How Lex & yacc works

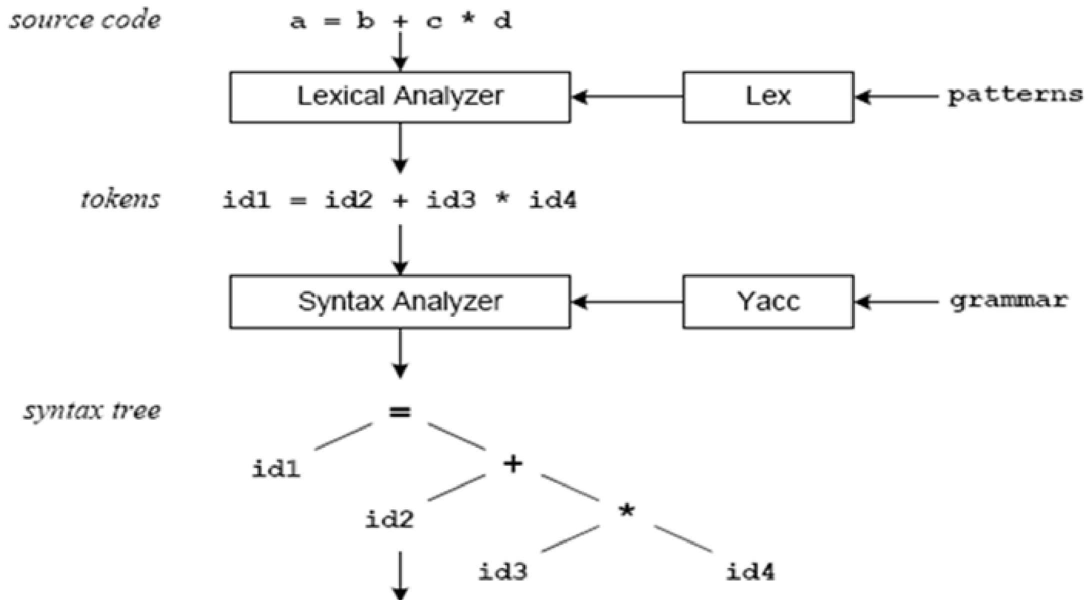


Fig. 9 Compilation Sequence [27]

From the diagram above, we can see the sequence of process. [27]

At the first layer, or stage, lex can generate C code for a lexical analyser based on the patterns we put in it. And lexical analyser will process and convert the input string to tokens. Those tokens are numerical representations of input strings, and simplify processing. The lexical analyser enters those identifiers in a symbol table when it finds them from the input stream. Besides identifiers, the symbol table could also contain other information such as data type (integer or real) and location of the variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

The second layer, or stage, yacc will generate C code for a syntax analyser or parser based on the grammar we put in it. Syntax analyser will analyse tokens from the lexical analyser according on the grammar rules, and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. In this diagram, the operator precedence and associativity are apparent in the syntax tree.

5.2 ANTLR

5.2.1 Background

ANTLR refers to ANother Tool for Language Recognition. [28]

It is the second generation parser generator. The first generation was called Purdue Compiler Construction Tool Set (PCCTS) which first developed in 1989. Both of them were designed and implemented by Professor Terence Parr of the University of San Francisco. [28]

"ANTLR uses Extended Backus-Naur (EBNF) grammars which can directly express optional and repeated elements.ANTLR supports LL(*) parsing which can allow program infinitely lookahead for selecting the rule alternative that matches the portion of the input stream being evaluated.

*An LL(k) parser is a top-down parser that parses from left to right, constructs a leftmost derivation of the input and looks ahead k tokens when selecting between rule alternatives. The * means any number of lookahead tokens."* [28]

ANTLR also has Lexical analyser and syntax analyser parts, called Lexer and Parser respectively.

5.2.2 How ANTLR works

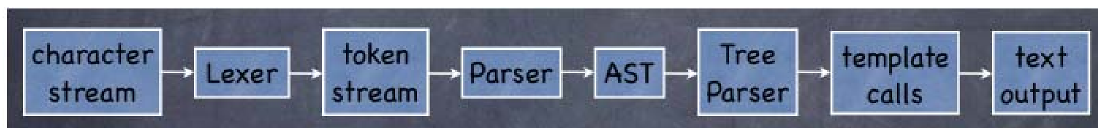


Fig. 10 ANTLR Working approach [29]

Character stream is input into Lexer, Lexer then converts the stream of characters to a stream of tokens based on the lexer grammar. The tokens, after, go into Parser. Parser then constructs those tokens to an AST (Abstract Syntax Tree). The AST will be delivered to Tree Parser where it gets process. And finally, text will optionally output by using StringTemplate which is a library that supports using templates with place holders for outputting text(ex. Java source code)

5.3 Parser generator choice for project

The parser generator tool I am going to choice for my project is ANTLR. Even though, lex/yacc is seemed to be a better choice for C. But from my point of view, ANTLR has those following advantages for me:

- ANTLR is used in windows system environment, yacc is designed for using in UNIX OR Linux.
- ANTLR provide friendly GUI for help programmer visualize what we are building.
- ANTLR generates recursive parsers and provide good error reporting[30]
- Open source and free, ANTLR users are worldwide, it's easier to identify and correct bugs for this parser
- My programming language target generated would be in Java.

5.4 A C grammar for ANTLR

The only C grammar resource for ANTLR has been found online is:

> *"ANSI C ANTLR v3 grammar Which translated from Jutta Degener's 1995 ANSI C yacc grammar by Terence Parr in July 2006."*[31]

There are hundreds lines of code, here can only show part of the grammar, and one of them looks like:

"

```
storage_class_specifier
    : 'extern'
    | 'static'
    | 'auto'
    | 'register'
    ;
```

" [31]

This segment of code describes that there are four terminal words, as also can be called 'Key words' in C, 'extern', 'static', 'auto', 'register' were defined to a identifier called 'storage_class_specifier'.

There is another example of the grammar:

"

```
Fragment
LETTER
    :   '$'
    |   'A'..'Z'
    |   'a'..'z'
    |   '_'
    ;
```

" [31]

The name "LETTER" is a Non-terminal word represents any combination of the legal word that we can write in C programming language. In this case it could contain any number of '$' or uppercase 'A' to 'Z' or lower case 'a' to 'z' or '_' no matter what the sequence they are.

6 Conclusion

After researching, I got some basic idea about what I am going to do, and studied the relative technical knowledge. For my project, I have to use third party tool called “ANTLR” for generating parser tree to analysis the input C source code. The ANSI C grammar has been found, but need more studying and understanding. Furthermore, I need more understudying about MISRA C 1998 guideline and trying to work out how to change the theory to grammar. Finally, using this grammar and my tool to check the input C source code, and output the feedback information to user.

7 Reference

[1] “Guidelines for the use of the C Language in Vehicle-Based Software,” published in 1998 by MISRA, www.misra.org.uk

[2] ANSI X3.159-1989, Programming languages - C, American National Standards Institute,1989

[3] "ANSI C." Wikipedia, The Free Encyclopedia. 6 Nov 2009 http://en.wikipedia.org/wiki/ANSI_C

[4] p2-p4, The Motor Industry Software Reliability Association, Guidelines For The Use Of The C Language In Vehicle Based Software, April 1998

[5] "MISRA" triangle logo is registered trademarks of MIRA Ltd, held on behalf of the MISRA Consortium.

[6] Frequently asked questions about MISRA. www.misra.org.uk

[7] An Introduction to MISRA C++ University of Warwick, Chris TappKeylevel Consultants Ltd, 28 th June 2007 28th.

[8] “Introduction to MISRA C, http://www.misra-c2.org/, MISRA

[9] p7-p8. The Contrast Between MISRA C and Ada 95, Julian Day, MISRA C Forum, Octorber 2002

[10] p39, p46. Section 7: Rules, The Motor Industry Software Reliability Association, Guidelines For The Use Of The C Language In Vehicle Based Software, April 1998

[11] p2, MISRA C Rule Checker SQMlint V.1.03 User’s Manual, Rev.1.00 Aug.1.2006, Renesas Technology

[12] p15, MISRA C Rule Checker SQMlint V.1.03 User’s Manual, Rev.1.00 Aug.1.2006, Renesas Technology

[13] Renesas Technology Corp, 2003-2009, http://eu.renesas.com/fmwk.jsp?cnt=number_of_misra_c_rules_that_can_be_inspected.htm&fp=/products/tools/coding_tools/extension_software_compilers/misra_c_rule_checker_sqmlint/child_folder/&title=Number%20of%20MISRA%20C%20rules%20that%20can%20be%20inspected

[14] Renesas Technology Corp, 2003-2009,
http://eu.renesas.com/fmwk.jsp?cnt=misra_c_rules_supported.htm&fp=/products/tools/coding_tools/extension_software_compilers/misra_c_rule_checker_sqmlint/child_folder/&title=MISRA%20C%20Rules%20Supported
[15] Renesas Technology Corp, 2003-2009,
http://eu.renesas.com/fmwk.jsp?cnt=operating_environment.htm&fp=/products/tools/coding_tools/extension_software_compilers/misra_c_rule_checker_sqmlint/child_folder/&title=Operating%20Environment

[16] Renesas Technology Corp, 2003-2009,
http://eu.renesas.com/fmwk.jsp?cnt=misra_c_rule_checker_sqmlint_tools_product_landing.jsp&fp=/products/tools/coding_tools/extension_software_compilers/misra_c_rule_checker_sqmlint/

[17] p3, MISRA C Rule Checker SQMlint V.1.03 User's Manual, Rev.1.00 Aug.1.2006, Renesas Technology

[18]p4, MISRA C Rule Checker SQMlint V.1.03 User's Manual, Rev.1.00 Aug.1.2006, Renesas Technology

[19] Reference Maunal for PC-lint/FlexeLint for C and C++, Software Version 8.00 and later Document Version 8.00, Gimpel Software,July, 2001

[20] http://www.keil.com/pclint/example.asp

[21] Business Glossary, Barron's Educational Series, Inc.
http://www.allbusiness.com/glossaries/parsing/4949237-1.html, Copyright © 2006, 2003, 2000, 1998, 1996, 1995, 1992, 1989, 1986 by Barron's Educational Series, Inc. Reprinted by arrangement with Publisher.

[22] http://en.wikipedia.org/wiki/Parsing

[23] http://en.wikipedia.org/wiki/Top-down_parsing

[24] http://en.wikipedia.org/wiki/Bottom-up_parsing

[25] http://en.wikipedia.org/wiki/Lexical_analysis

[26] p4, lex & yacc, John R. Levine,Tony Mason & Doug Brown, O'REILLY, 1995

[27] p4, A COMPACT GUIDE TO LEX & YACC, Tom Niemann ,

[28] http://en.wikipedia.org/wiki/ANTLR

[29] ANTLR 3, R. Mark Volkmann, Partner/Software Engineer Object Computing, Inc. (OCI)
http://jnb.ociweb.com/jnb/jnbJun2008.html#References

[30] http://www.bearcave.com/software/antlr/antlr_expr.html

[31]ANSI C ANTLR v3 Grammar, Terence Parr, July 2006
http://www.antlr.org/grammar/1153358328744/C.g