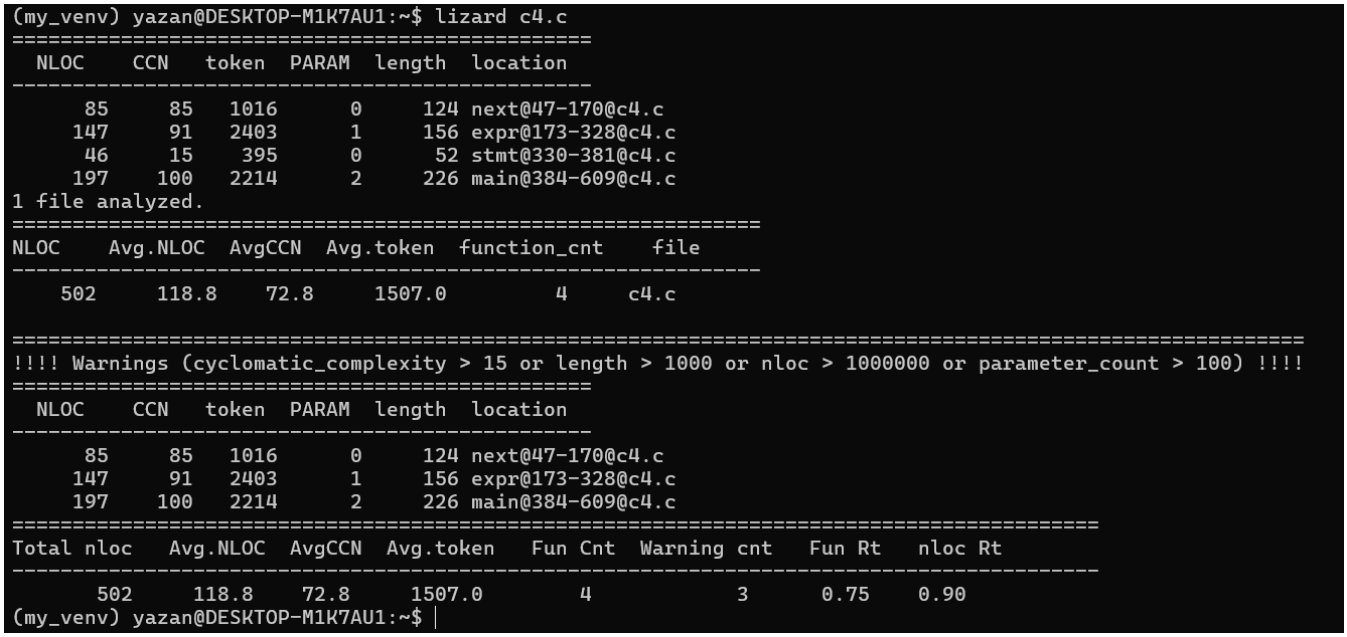
The Lizard tool is a static code analysis tool that measures metrics about the source code such as complexity and size. By using the Lizard tool, we obtained valuable information about the number of lines in each function and their respective cyclomatic complexity.

The image below represents the details of the commented file:



Each of the columns in the analysis can be understood as follows:

- **NLOC**: Number of Lines of Code (only executable lines), excluding comments, blank lines, and non-executable statements.
- **CCN**: Cyclomatic complexity, which indicates the number of independent paths through the code.
- **Token**: The count of tokens used within each function (such as keywords, variables, and operators).
- **PARAM**: The number of parameters a function takes.
- **Length**: The total number of lines from the start to the end of a function, including both executable and non-executable lines.

3.1: The Lines of Code (LOC) for the entire program and individual functions along with number of functions and their sizes in (LOC) are as follows:

- The entire code consists of 502 executable lines.
- Individual Functions:
    - Next function: 85 lines.
    - Expr function: 147 lines.
    - Stmt function: 46 lines.
    - Main function: 197 lines.

3.2: The Cyclomatic Complexity (CCN) for each function is as follows:

- Average Cyclomatic Complexity: The average cyclomatic complexity for the entire code is 72.8, indicating an average of approximately 73 possible execution paths.
- Individual Functions:
    - Next function: Cyclomatic complexity of 85.
    - Expr function: Cyclomatic complexity of 91.
    - Stmt function: Cyclomatic complexity of 15.
    - Main function: Cyclomatic complexity of 100.

3.3: Number of Global Variables and Their Usage

- The code contains several global variables, each serving a specific purpose in managing the state of the compiler. These global variables are as follows:

    ***char *p, lp, data:*

        - p: Tracks the current position in the source code.
        - lp: Points to the starting position of the current line (line pointer).
        - data: Points to the data/bss segment.

    ***int *e, *le, id, sym, tk, ival, ty, loc, line, src, debug:*

        - e: Tracks the current position in the emitted (intermediate) code.
        - le: Marks the end position in the emitted code.
        - id: Stores the currently parsed identifier.
        - sym: The symbol table, which stores identifiers and their properties.
        - tk: Represents the current token.
        - ival: Holds the current token value.
        - ty: Represents the current expression type.
        - loc: Stores the local variable offset.
        - line: Tracks the current line number.

- src: A flag to enable or disable printing of source and assembly code.
- debug: A flag to enable or disable printing of executed instructions.

- These global variables are used throughout the code to manage various aspects of the compiler, including token parsing, symbol table management, code generation, and debugging. They help maintain the state of the compilation process and ensure efficient handling of data and control flow within the program.

3.4: Number of Unique Tokens and Their Frequency

- The code defines several tokens, which are used to represent various elements in the compiled code, and these tokens are categorized using an enum.
- Unique Tokens:
  - Keywords (8 tokens):
    - *Char, Else, Enum, If, Int, Return, Sizeof, While.*
  - Operators (23 tokens):
    - *Assign, Cond, Lor, Lan, Or, Xor, And, Eq, Ne, Lt, Gt, Le, Ge, Shl, Shr, Add, Sub, Mul, Div, Mod, Inc, Dec, Brak.*
  - Special Tokens (6 tokens):
    - *Num, Fun, Sys, Glo, Loc, Id.*
- Total Unique Tokens: 37 unique tokens in total.
- Frequency: The frequency of each token depends on the input source code being parsed, and it varies based on the code structure and its usage of these tokens.

The tokens that were identified from the source code of c4 itself along with the number of times they were used is as follows:

354 =	;	201 ==	;	194 if	;	189 tk	;	152 e
151 *	;	148 else	;	121 *++	;	96 next	;	70 -
69 1	;	65 ty	;	65 ++	;	62 p	;	61 i
58 id	;	58 a	;	56 sp	;	51 printf	;	51 int
40 return	;	37 expr	;	36 line	;	36 PSH	;	35 +
33 INT	;	30 while	;	30 t	;	27 0	;	23 ||
23 exit	;	23 d	;	23 !=	;	22 &&	;	21 pc

19 Class ; 18 PTR ; 18 Inc ; 17 poolsz

17 ival ; 17 char ; 17 IMM ; 17 <= ; 16 Val

15 sizeof ; 14 >= ; 13 CHAR ; 13 Assign ; 13 >

11 argv ; 11 Type ; 11 Mul ; 11 LI ; 11 LC

10 data ; 10 ! ; 8 pp ; 8 argc

8 SUB ; 8 Num ; 8 Int ; 8 Char ; 8 *--

7 bp ; 7 b ; 7 Tk ; 7 Shl ; 7 MUL

7 Loc ; 7 Id ; 7 ADD ; 7 < ; 7 &

6 stmt ; 6 malloc ; 6 le ; 6 cycle ; 6 bt

6 BZ ; 6 Add ; 5 sym ; 5 n ; 5 lp

5 SI ; 5 SC ; 5 Lt ; 5 JMP ; 5 And

5 ADJ ; 5 9 ; 4 void ; 4 src ; 4 memset

4 loc ; 4 fd ; 4 enum ; 4 Xor ; 4 XOR

4 Sub ; 4 Or ; 4 Lan ; 4 LEV ; 4 HVal

4 HType ; 4 HClass ; 4 Eq ; 4 EXIT ; 4 EQ

4 Dec ; 4 DIV ; 4 Cond ; 4 A ; 4 2

4 – ; 3 | ; 3 idmain ; 3 debug ; 3 While

3 Sys ; 3 Shr ; 3 SHR ; 3 SHL ; 3 OR

3 OPEN ; 3 Ne ; 3 Name ; 3 NE ; 3 Mod

3 MOD ; 3 Lor ; 3 Le ; 3 LT ; 3 LEA

3 LE ; 3 JSR ; 3 Idsz ; 3 Hash ; 3 Gt

3 Glo ; 3 Ge ; 3 GT ; 3 GE ; 3 Fun

3 ENT ; 3 Div ; 3 Brak ; 3 BNZ ; 3 AND

3 5 ; 3 3 ; 3 / ; 3 ** ; 2 ~ ; 2 z

2 read	;	2 open	;	2 memcmp	;	2 lev	;	2 close
2 _	;	2 ^	;	2 Z	;	2 Sizeof	;	2 Return
2 READ	;	2 PRTF	;	2 MSET	;	2 MCMP	;	2 MALC
2 If	;	2 FREE	;	2 Enum	;	2 Else	;	2 <<
2 6	;	2 %	;	1 x	;	1 s	;	1 main
1 free	;	1 f	;	1 X	;	1 F	;	1 >>
1 8	;	1 7	;	1 4	;	1 256	;	1 16
1 15	;	1 147	;	1 128	;	1 1024	;	1 10

3.5: Number of Branches, Loops, and Their Nesting Levels

- Branches represent the control flow of the program, such as `if-else` statements. These are counted as one branch, regardless of the number of conditions.
    - Next function: ~30 branches.
    - Expr function: ~50 branches.
    - Stmt function: ~10 branches.
    - Main function: ~40 branches.
- Loops are used to iterate over data or repeatedly execute code, such as `while` loops for token parsing and processing.
    - Next function: ~10 loops.
    - Expr function: ~5 loops.
    - Stmt function: 1 loop.
    - Main function: ~12 loops.
- Nesting Levels: The code typically features 2-3 levels of nesting deep, depending on the function. This indicates that the functions contain a moderate amount of nested control structures like loops and branches.

3.6: Memory Usage Patterns (Stack vs. Heap Allocation)

- **Stack Memory Usage:**
    - Local Variables: Variables declared inside functions (e.g., int t, *d; inside expr function) are allocated on the stack. Each function call creates a stack frame that contains these variables, and the space is automatically freed when the function returns.

    - Function Call Stack: The use of the stack pointer (sp) for function calls, where return addresses and local data are pushed onto the stack, represents typical stack usage.
- **Heap Memory Usage:**
    - Dynamic Memory Allocation: The code uses malloc for dynamic memory allocation, reserving memory at runtime which must be manually managed. For example, malloc(poolsz) is used to allocate memory for symbols, data, and the stack. This memory does not automatically free itself; explicitly releasing it is required using free().