International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 10 Issue: 01 | Jan 2023 www.irjet.net p-ISSN: 2395-0072

LEXICAL ANALYZER

Rushikesh Lakhotiya[1], Mayuresh Chavan[2], Satwik Divate[3], Soham Pande[4]

[1,2,3,4] *Student, Dept. of Artificial Intelligence and Data Science, Vishwakarma Institute of Technology, Pune, Maharashtra, India*

Abstract - *An The process of turning a string of letters into a string of tokens is known as lexical analysis, commonly referred to as lexing or tokenization. These tokens may be keywords, identifiers, constants, operators, or other language-specific symbols.*

The word lexical is obtained from the native word i.e. lexeme which means tokens.

The process of lexical analysis usually includes reading each character of the input one by one, grouping characters into tokens, and passing these tokens to a parser or other program for further processing.

Lexical analysis is often first step in the operation of compiling or interpreting a program. It is also used in natural language processing, information retrieval, and other fields where it is necessary to identify and classify the elements of a body of text.

In general, lexical analysis involves breaking up a stream of text into a sequence of tokens, which can then be further processed and analyzed by other programs. It is an important step in the compilation and interpretation of programming languages, as well as in the processing of natural language.

***Key Words*: Lexical Analyzer, Lexeme, Compiler, Syntax analysis, Deterministic Finite Automata, Regular Expression, Compiler, Tokens, Parallel Tokenization, Multi-core Machines.**

1. INTRODUCTION

The initial step in the compilation procedure is the lexical analyzer. This phase, also referred to as a lexical scanner, scans the input string without going back and reading each symbol more than once before fully processing it. The primary responsibility of lexical analyzer is to take input characters and generate the output of the token sequence that the parser utilises for lexical analysis. The lexical analyzer receives input characters from the parser and reads them until it can recognize the next token.

Tokens are generated from lexemes by a lexical analyzer. Internally, the tokens are frequently represented as distinct integers or an ordered type. In order to distinguish between the multiple name or numeric tokens in this example, the lexeme is necessary in addition to the token itself.

The lexical analyzer normally functions independently and only uses one or two subprocesses and global variables to interact with rest of the compiler. Every time the parser requires a new token, it calls the lexical analyzer, which then delivers both the token and the lexeme that goes with it.

The lexical anlyzer can be replaced or modified without im pacting the remaining compiler, because the real input is concealed from the parser. The lexical analyzer normally functions independently and only uses one or two subprocesses and global variables to interact with the rest of compiler.

When the parser requires a new token, it invokes the lexical analyzer, which then returns the token and its lexeme. The lexical analyzer may be changed or replaced without having an impact on the remainder of the compiler since actual input process is concealed from the parser.

In this paper we study about the role of lexical analysis in the overall process of compiling or interpreting a program, Techniques for defining the tokens that a lexical analyzer should recognize, such as using regular expressions along with the implementation.

2. LITERATURE REVIEW

Daniele Paolo Scarpazza et al. [1] A parallel regexp-based tokenization technique has been suggested that makes use of the substantial thread- and data-level parallelism offered by multi-core architecture. It is derived from the Deterministic Finite Automation (DFA) model, which was created for branch elimination and SIMDization in prediction-like applications.

Umarani Srikanth [2] The suggested approach divides the source programme into a predetermined number of blocks using a dynamic block splitter algorithm to carry out lexical analysis concurrently. By swiftly scanning through large dictionaries on multiple core IBM Cell processors for a string, the Aho-Corasick method tokenizes data.

Swagat Kumar Jena et al. [3] According to the method, each block of the source file is divided into M lines, with

the possible exception of last block, and each block is stored in memory as a separate file. Then, N lexical programme threads are created, and lexical analysis is carried out simultaneously for each file using N lexical programme threads.

Amit Barve et al. [4] stated The method based on an open source automatic lexer generator Flex and exploiting the concept of processor affinity and partitioning code written in C/C++ programming language based on for-loop looping structures.

Amit Barve et al. [5] said, The method is based on setting the pivot points, which divides the code into a predetermined block count equal to the amount of available CPUs. Considerations were made for white space characters, various topologies, and pivot components based on lines.

Amit Barve et al. [6] Modernized version the approach provided by Amit Barve et al. is concluded when of developing fast parallel lexers for multi-core processors. A proposed algorithm stores block indicators of source code in a text file that will later be read. Based on the read indicators, processes are branched and assigned to different CPUs using processor affinity. The algorithm's efficiency is increased by assigning operations to an available processor only when a process is formed. This method eliminates the need to wait for a process to be allocated to a processor.

Amit Barve et al., (2015) [7] explained an enhanced version for parallel lexical analysis algorithm. Furthermore, if the number of CPUs rises, the speed is seen as being higher. As a result, this approach can improve compilation time even more. The author asserts that the memory block-based approach exceeds the results of his earlier work, which used a round-robin CPU scheduling technique to run lexical analysis in parallel and the highest speed achieved is 6.84. The speed will be increased when number of CPU rises and also improves the overall compilation time. Daniele Paolo Scarpazza et al., (2007) investigated the importance of the efficiency of the cell processor system when it is used for the implementation of Deterministic Finite Automata based string matching process algorithms.

Daniele Paolo Scarpazza et al., (2008) [8] the results of their experiment indicate that the Cell is the perfect candidate for managing security requirements. One cell processor has eight processing units, but only two of them have the processing ability to process a net connection with a data rate of even more than 10 Gbps. Using the Aho-Corasick string searching algorithm, developed optimized string matching solutions for the Cell processor and the result showed a throughput of 40 Gbps per processor when The speed for bigger dictionaries is somewhere between 1.6 to 2.2 Gbps per processor, and the dictionaries are tiny enough to fit into local memory space of the processing cores.

Wuu Yang et al (2002) [9] identified the issue of the longest-match rule's applicability and proposed a model. The method consists of two steps: the first is to determine the regular set of token patterns produced by non-deterministic finite automaton while the automaton processes components of an input regular set, and the second is to determine whether a regular set and a free language have any non-trivial intersections with a set of equations. To enable parallel procedure model in the C programming language, Russell et al. (1992) created additions to the parallel procedural language and a runtime environment. In order to reduce the need for expensive process control blocks to be implemented, a novel method for nesting parallel process contexts in multiple stack frames is used in the run-time framework and the performance data for two parallel programs utilizing their proposed system is provided.

Xiaoyan Lai., (2014) [10] begins an innovative implementation approach for syntactic analysis, interpretative execution, and lexical analysis. The experimental analysis provides integrity and reliability of compilation system. Amit Barve et al., (2012) presented a new approach of implementing lexical analyzer to run in parallel which is based on an open source automatic lexer generator Flex and exploiting the concept of processor affinity. It is measured to be a simple and faster process by partitioning code written in C/C++ programming language based on for-loop looping structures. Work reasonably illustrates the benefit of multi-core architecture machines in accelerating the process of lexical analysis tasks. Thomas Reps et al., (1998) described the compilation domain, where tokenization process can always be carried out in time linear in the input size, while most of the standard tokenization algorithm explains that, in the worst case, the scanner can exhibit quadratic behavior for some sets of token definition

3. METHODOLOGY

A) Proposed System

There are several different approaches to performing lexical analysis, but a common methodology involves the following steps:

i. Read each character of the input one at a time.

ii. Identify the next token in the input. This may involve looking for patterns in the characters, such as sequences of digits that form a number, or strings of letters that form a keyword or identifier.

iii. Extract the token from the input.

iv. Classify the token based on its type (e.g., keyword, identifier, constant, operator).

v. Pass the token to a parser or other program for further processing.

Some lexical analyzers also include additional steps, such as removing comments or white space from the input, or performing preprocessing on the input before tokenization. It is also possible to use regular expressions or finite automata to perform lexical analysis. Regular expressions are a way of describing patterns in strings, and can be used to identify and extract tokens from the input. Finite automata are mathematical models that can be used to recognize patterns in input and are often used in the implementation of lexical analyzers.

B) Flowchart

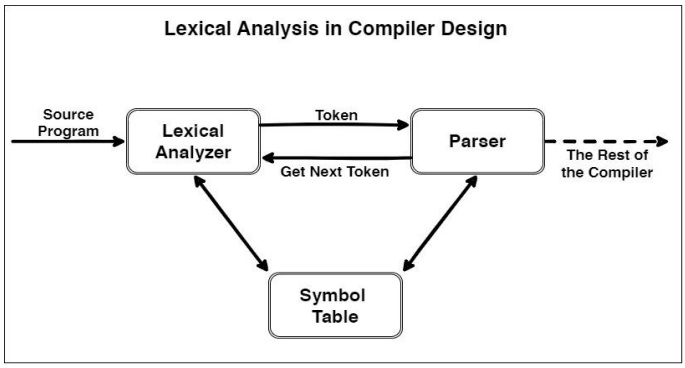

Fig -1: Lexical Analyzer with the Parser

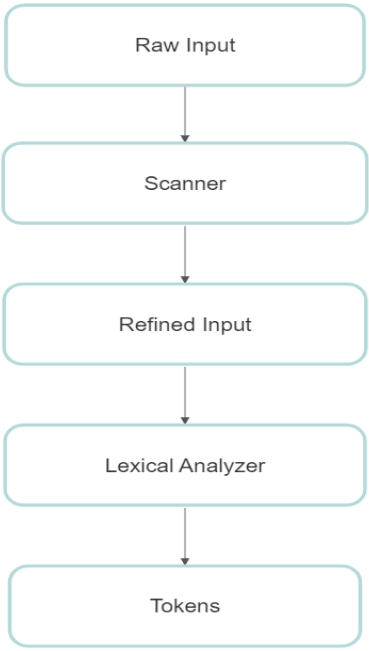

Fig -2: Dividing into Tokens

4. RESULTS AND DISCUSSIONS

```c
#include <stdio.h>
int main() {

    int a = 10;

    printf("Value of a is %d", a);

    return 0;
}

```

Fig -3: Sample Input Program

In fig.3 : A sample C language program passed to the lexical analyzer.

```
Lexical Analyzer using C
By Group 3 VIT Pune

Keywords : int int return

Identifiers : include stdioh main a 10 printfValue of a is d a 0

Math Operators : %

Logical Operators : < >

Numerical Values : 1 0 0

Others : ( ) { ( ) }
Program ended with exit code: 0
```

Fig -4: Output

In fig.4: The sample C program given as input is converted into tokens i.e. Keywords, Identifiers, Mathematical Operators, Logical Operators, Numerical Values, Other (Separators).

5. LIMITATIONS

- The presence of an illegal character, often at the start of a token, results in a lexical mistake.
- Some of the regular expressions are quite challenging to comprehend.
- The lexer and also its token descriptions require more work to build and test.

6. CONCLUSION

The goal of this study was to conduct a thorough examination of recent research on lexical analyzer implementation methodologies. It is known from the review that various software tools for lexical analyzers have been developed in the past that are ideally suited for serial execution. The different stages of the compilation process must be updated to accommodate multi-core architecture technologies as a result of the rise of multi-core architecture systems in order to attain a parallelism in compilation tasks and thereby minimize the time of compilation. An extensive analysis provides a deeper insight towards the lexical analyzer. Among the results

recorded in the reviewed articles, it is observed, some of the high-level trends in scanner generation are the adaptation of parallel processing in lexical analysis tasks by using multi-core processor affinity principle to increase the efficiency of the compiler's runtime compared to the sequential execution of lexical analysis tasks on a single processor system.

7. FUTURE SCOPE

The development of computing power is moving rapidly towards massive multi-core platform due to its power and performance benefits. System software, including compilers, should be designed for parallel processing in order to take full use of multi-core technology. Implementing more pattern matching algorithms into the program.

REFERENCES

[1] Aho, A. V., Lam, M. S., & Sethi, R. (2009). Compilers Principles, Techniques and Tools, 2nd ed, PEARSON Education.

[2] Lesk, M. E., & Schmidt, E. (1975). Lex: A lexical analyzer generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hills, New Jersey.

[3] Mickunas, M. D., & Schell, R. M. (1978, December). Parallel compilation in a multiprocessor environment. In Proceedings of the 1978 annual conference (pp. 241-246).

[4] Kitchenham, B. (2004). Procedures for performing systematic reviews. Keele, UK, Keele University, 33(1), 1-26.

[5] Webster, J., & Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a literature review. MIS quarterly, 26(2), 8-23.

[6] Glesner, S., Forster, S., & Jager, M. (2005). A program result checker for the lexical analysis of the gnu c compiler. Electronic Notes in Theoretical Computer Science, 132(1), 19-35.

[7] Barve, A., & Joshi, B. K. (2012, September). A parallel lexical analyzer for multi-core machines. In 2012 CSI Sixth International Conference on Software Engineering (CONSEG) (pp. 1-3). IEEE.

[8] Barve, A., & Joshi, B. K. (2013). Automatic C Code Generation for Parallel Compilation. International Journal on Advanced Computer Theory and Engineering (IJACTE), 2(4), 26-28.

[9] Omori, Y., Joe, K., & Fukuda, A. (1997, August). A parallelizing compiler by object-oriented design. In Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97) (pp. 232-239). IEEE.

[10] Barve, A., & Joshi, B. K. (2016). Fast parallel lexical analysis on multi-core machines. International Journal of High-Performance Computing and Networking, 9(3), 250- 257.

[11] Scarpazza, D. P., Villa, O., & Petrini, F. (2007, March). Peak-performance DFA-based string matching on the Cell processor. In 2007 IEEE International Parallel and Distributed Processing Symposium (pp. 1-8). IEEE

[12] Jena, S. K., Das, S., & Sahoo, S. P. (2018). Design and Development of a Parallel Lexical Analyzer for C Language. International Journal of Knowledge-Based Organizations (IJKBO), 8(1), 68-82.

[13] Clapp, R. M., & Mudge, T. N. (1992). Parallel language constructs for efficient parallel processing. University of Michigan, Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science. IEEE, 230-241.

[14] Li, D. C., Cai, X. C., Han, C. Y., & Liu, Y. X. (2012). The Research and Analysis of Lexical Analyzer in Prolog Compiler. In Applied Mechanics and Materials (Vol. 229, pp. 1733-1737). Trans Tech Publications Ltd.

[15] Maliavko, A. A. (2018, October). The Lexical and Syntactic Analyzers of the Translator for the EI Language. In 2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE) (pp. 360-364). IEEE.

[16] Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., & Zhu, H. (2019). Hyperscan: a fast multi-pattern regex matcher for modern cpus. In 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19) (pp. 631-648).

[17] Becchi, M., & Crowley, P. (2013). A-dfa: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation. ACM Transactions on Architecture and Code Optimization (TACO), 10(1), 1-26.

[18] Aithal, P. S., & Pai T, V. (2016). Concept of Ideal Software and its Realization Scenarios. International Journal of Scientific Research and Modern Education (IJSRME), 1(1), 826-837.

[19] Ingale, Varad, Kuldeep Vayadande, Vivek Verma, Abhishek Yeole, Sahil Zawar, and Zoya Jamadar. "Lexical analyzer using DFA." International

Journal of Advance Research, Ideas and Innovations in Technology, www. IJARIIT. com.

[20] Chandra, Arunav, Aashay Bongulwar, Aayush Jadhav, Rishikesh Ahire, Amogh Dumbre, Sumaan Ali, Anveshika Kamble, Rohit Arole, Bijin Jiby, and Sukhpreet Bhatti. Survey on Randomly Generating English Sentences. No. 7655. EasyChair, 2022.

[21] Manjramkar, Devang, Adwait Gharpure, Aayush Gore, Ishan Gujarathi, and Dhananjay Deore. "A Review Paper on Document text search based on nondeterministic automata." (2022).

[22] Vayadande, Kuldeep, Neha Bhavar, Sayee Chauhan, Sushrut Kulkarni, Abhijit Thorat, and Yash Annapure. Spell Checker Model for String Comparison in Automata. No. 7375. EasyChair, 2022.

[23] ayadande, Kuldeep, Ram Mandhana, Kaustubh Paralkar, Dhananjay Pawal, Siddhant Deshpande, and Vishal Sonkusale. "Pattern Matching in File System." International Journal of Computer Applications 975: 8887.

[24] VAYADANDE, KULDEEP. "Simulating Derivations of Context-Free Grammar." (2022).

[25] Vayadande, Kuldeep B., Parth Sheth, Arvind Shelke, Vaishnavi Patil, Srushti Shevate, and Chinmayee Sawakare. "Simulation and Testing of Deterministic Finite Automata Machine." International Journal of Computer Sciences and Engineering 10, no. 1 (2022): 13-17.