Design of efficient Programming Language with Lexer using '$'-prefixed identifier

Priya Gupta[1], L S Yaswanth Kumar[2], J V V M S D Santosh[2], D Yashwanth Kumar[2], Chokkari Dinesh[2], Mukkoti Maruthi Venkata Chalapathi[3, *]

[1] Atal Bihari Vajpayee School of Management and Entrepreneurship, Jawaharlal Nehru University, New Dehli, India
[2] School of Engineering, Jawaharlal Nehru University, New Delhi, India
[3] School of Computer Science and Engineering, VIT-AP University, Amaravati, Andhra Pradesh, India

Abstract

An identifier which starts with '$'is known as '$'-prefixed identifier and this type of identifiers are used in our research paper to improve the lexical analysis phase. This paper talks about a new programming language with '$'-prefixed identifier that features a novel approach for optimizing the lexer for efficient lexical analysis which can be applied to any existing language. This approach is used to classify identifiers and keywords using '$'-prefixed variables, which significantly reduces the time taken and number of iterations required during the tokenization process, leading to improved overall performance. This type of language structure allows for fast lookup and matching of tokens. We conducted a series of experiments to evaluate the performance of our lexer and compared it with a regular lexer. Our results show that our approach leads to significant improvements in time complexity and number of iterations for identifying whether the token is an identifier or a keyword, resulting in faster compilation times and improved overall performance. Our language has reduced the amount of time taken by 7-10% and 45-50% in terms of iterations. Our language and lexer represent a significant step forward in the design and implementation of high-performance programming languages by reducing the number of iterations and time taken to identify whether a token is a keyword or an identifier.

1. Introduction

Compilers are essential tools in the field of computer science, as they allow users to write and execute programs in high-level programming languages [1]. To translate these programs into machine code, compilers must be able to accurately parse and understand the syntax and structure of the source language [2]. One key aspect of this process is the use of context-free grammar (CFGs) to define the syntax of programming languages. CFGs are a formal way of specifying the syntactic structure of a language, using a set of rules and symbols to define the allowed sequences of tokens (such as keywords, variables, and operators) in a program [5]. These grammars can be used to generate parse trees, which represent the hierarchical structure of a program and can be used to check its syntax and semantics [8]. In this research

paper, we will delve into the role of CFGs and regular expressions in compiler design and discuss a novel approach to improve the performance of Lexer. We will also explore the challenges and limitations of using this method for a general-purpose language. We aim to provide a better understanding of how this approach works and how they can be used to effectively classify identifiers in any general programming language. We have introduced a new lexical analyzer method to reduce the number of iterations while classifying the identifiers in the source code. Usually a lexical analyzer takes 25-30% of compilation time to perform lexical analysis [4]. As the lexical analyzer takes 30% of compilation time, we need to reduce that in order to decrease the overall compilation time. We have introduced '$'-prefixed identifiers in our language so that whenever a '$'is encountered it will be directly classified as an identifier immediately without performing a search over the list of keywords.

Noam Chomsky [2] made significant contributions to the field of formal languages and the concept of context-free grammar (CFGs). His work laid the foundation for the use of CFGs in compiler design and other areas of computer science. Chomsky's work on CFGs has had a significant impact on the field of compiler design, as they provide a formal way of specifying the syntactic structure of a language. Many compiler design techniques and algorithms, such as top-down and bottom-up parsing, are based on the use of CFGs to generate parse trees and check the syntax and semantics of programs [3].

Aho and Ulman in their book have provided a thorough overview of the use of CFGs in compiler design, including the various techniques and algorithms that are used to implement them [2]. The role of CFGs in defining the syntax of programming languages and the importance of generating accurate parse trees to check program syntax and semantics. They have also addressed the challenges and limitations of using CFGs in compiler design, including issues of ambiguity and efficiency [8].

Since time and computation power plays a vital role in construction of compilers for a programming language and lexical analysis takes almost 30% of the time of the compilation, we thought of improving the lexical phase of the compilers using '$'-prefixed compilers.

Organization - Section II talks about Literature Review i.e, about the previously published works. Section III talks about Methodology i.e, how the proposed method reduced the compilation time. Section IV talks about Proposed System i.e, how the method is supposed to be implemented. Section V talks about Results i.e, show the significant difference between the regular lexer and proposed lexer. Section VI talks about Limitations of the proposed method. Section VII talks about Advantages of the proposed method. Section VIII talks about Conclusion of the paper. Section IX talks about Future Scope i.e, how the proposed work can have significant impact in certain fields.

2. Literature Review

For improving lexical analysis phase Various tools are used for automatic generation of tokens and are more suitable for sequential execution of the process. Recent advances in multi-core architecture systems have led to the need to re-engineer the compilation process to integrate the multi-core architecture through which we obtain parallelization in the recognition of tokens in multiple cores optimally, thus reducing compilation time [20], [21]. They have also improved on handling errors of handling the errors, due to insertion, deletion, substitution, letter sequencing and typing in the lexical analysis phase of the compiler [11], [13], [14].

Fuzzy keywords, their fuzzy regular expressions and minimized fuzzy deterministic automata are constructed. The issue of membership of fuzzy keywords is successfully

tackled with the help of an algorithm. Full implementation of fuzzy lexical analyzer is also described. But this just improves the handling errors for the lexical analysis but it doesn't improve the time taken for the lexical analysis phase of the compiler [12], [15], [16]. Most of the research done so far deals with the improvement of other phases of compiler rather than lexical analysis. Very little work is done on the lexical phase that too on efficient use of multi core architecture and parallelization of the lexical analysis phase using the multi-core architecture, but it doesn't improve the time complexity for the analysis phase [17], [18],[19].

The construction of compilers for a programming language requires significant time and computation power. One major bottleneck is the lexical analysis phase, which typically accounts for almost 30% of the compilation time. To address this, $-prefixed identifiers were proposed to improve the efficiency of lexical analysis. In this paper the authors have tried working on optimizing the time complexity for lexical analysis phase and obtained substantial results with a language having $-prefixed identifiers which will lead to classification of Identifiers and keywords faster than the traditional lexer's.

3. Methodology

Lexical Analyzer accepts the preprocessor's output, which is in a pure high-level language and handles file inclusion and macro expansion, as input. It extracts the text from the source programme and organizes the characters into lexemes (groups of characters that "go together"). A token is assigned to each lexeme. The lexical analyzer can comprehend regular expressions used to define tokens. It also eliminates comments, white space, and lexical errors (such as incorrect letters) [7].

The time complexity of a lexer in general is O(nk), where n is the length of the input text and k is the number of keywords in the Language. The regular expression used to identify the keywords takes O(k) to aim to provide a better understanding of how these tools work and how they can be used to effectively parse and understand programming languages. Lexical Analysis Time to build and the find all method takes O(n) time to find all the matches in the input text. Therefore, the overall time complexity is O(nk).

If we have a language which has '$'-prefixed identifiers then we can iterate over the input text from the beginning of a keyword which is '$'to until an arrival of space character or any character other than {Letter, Digit} and ' ', Then we can end the iteration and say that the identifier exists in the buffer. For example, "VAR $varName = 10" is an input string then we iterate over the input string and from the arrival of '$', we can directly classify the following pattern as Identifier. Generally, when a pattern matching with can be an identifier or Keyword. So, when a similar pattern was found then the lexeme needs to be compared with all the keywords and if not

.found in the keywords list it will be identified as an Identifier otherwise it will be classified as a Keyword. So, to skip this additional matching with the keywords we have introduced '$'-prefixed identifiers. When a '$'is encountered then the following pattern will be classified as an Identifier without being compared with the keywords. In this way, the identification of the Identifiers is easily done without using regular expressions.

Fig:1 explains how a token is classified whether it's a keyword or an identifier and Fig:2 explains how our proposed method decides whether a token is a keyword or an identifier. This is a significant improvement over a lexer that would need to iterate over each character of the input text and check it against a list of keywords, which would have a time complexity of O(n) and identifying the keywords takes O(k) time. The use of regular expressions is eliminated by using a '$'-prefix identifiers structure language and following the above procedure for tokenization.

A. Mathematical Approach

Fig:1 and Algorithm:1 shows us how a lexer of a Normal Programming language takes decision to classify the keywords and identifiers which will take NL number of Iterations.

Algorithm 1 Normal Lexer

Require: *Token*
    if Token in KEYWORDS **then**
        return KeywordToken
    else
        return IdentifierToken
    end if
    It takes 'n' number of iterations to classify identifier and 'n' Iterations to classify keywords, where 'n' is number of keywords in Programming Language.

Algorithm 2 Proposed Lexer

Require: *Token*
    if Token [0] = '$' **then**
        return IdentifierToken
    else if token in KEYWORDS **then**
        return KeywordToken
    end if
    It takes 1 iteration to classify identifier and 'n' iterations to classify keywords, where 'n' is number of keywords in Programming Language

---

$$NL = (Number\ of\ Keywords + Number\ of\ Identifiers) \times Total\ Number\ of\ Keywords\ in a\ Programming\ Language$$

Whereas Fig:2 and Algorithm:2 us how our concept of lexing will classify the keywords and identifiers which will take PL number of Iterations.

$$PL = (Number\ of\ Keywords \times Total\ Number of\ Keywords\ in\ a\ Programming\ Language) + Number\ of\ Identifiers$$
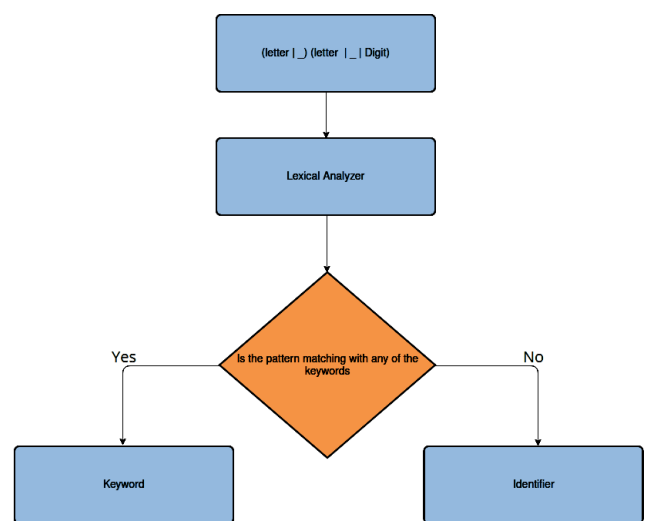

Fig. 1. GENERAL LEXICAL ANALYZER

B. Method of Programming

Method of programming is based on how the programming process is further implemented. Types of methods of programming used for implementation:

1) Procedural Programming: To improve the modularization and reusability of code, procedures or blocks of code can be decomposed into smaller tasks. The entire program is made up of all the procedures. Each of these operations can be implemented as a separate process for a calculator program that performs addition, subtraction, multiplication, division, square root, and comparison. Each procedure would be called in the main program based on the user's selection. This language would involve defining functions that encapsulate reusable blocks of code and using them to implement the desired program logic [9].
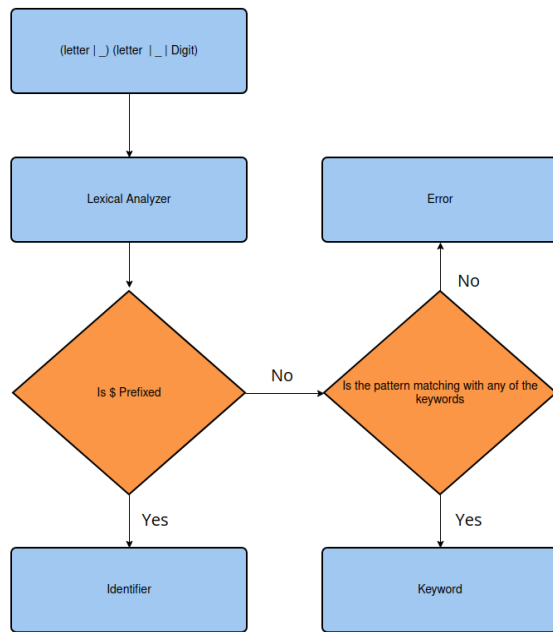
Fig. 2. PROPOSED LEXICAL ANALYZER

2) Functional Programming: Here, the issue or the desired outcome is divided into workable components. Each unit is independent and capable of carrying out its own duty. The entire solution is then created by sewing these sections together [9].
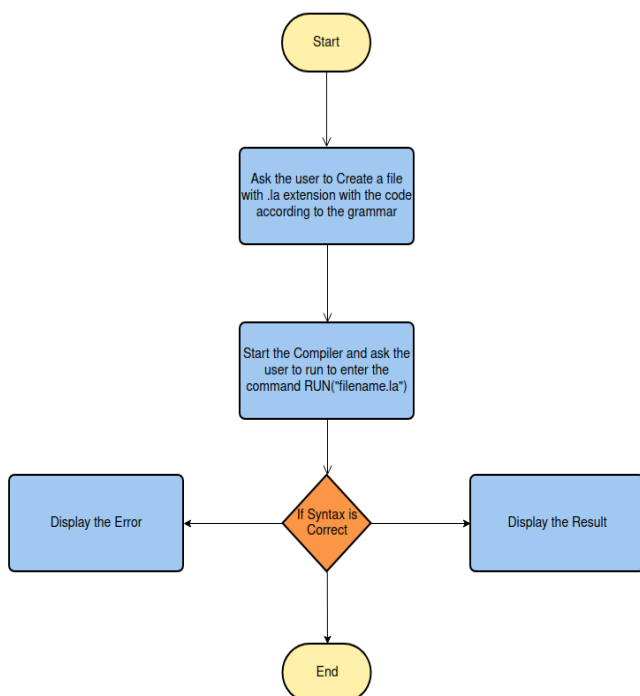

Fig. 3. FLOWCHART OF A PROGRAMMING LANGUAGE

4. Proposed System

This Compiler design using a Context-Free Grammar project helps to create a custom programming language using its own set of rules and syntax. A Context Free Grammar was declared, and a compiler was developed to compile a specific programming language and display the desired output.

Steps in designing the compiler and language:

1) Define the syntax and semantics of the programming language: This will involve deciding on the structure and rules of the language, including the types of variables, operators, and control structures it will support [10].

2) Implement the lexer: To break down the input in a sequence of tokens by writing code that can match the various tokens in the language and define functions to handle the lexing process [6] proposed lexer was implemented.

3) Implement the parser: The parser is responsible for generating a parse tree from the input program, using the rules of the CFG defined in step I. Parsing algorithm, such as top down or bottom-up parsing, to construct the parse tree was used [7].

4) Implement the semantic analyzer: The semantic analyzer is responsible for checking the semantics of the program, including verifying the types of variables and ensuring that the program follows the rules of the language [8].

5) Implement the code generator: The code generator is responsible for translating the parse tree into machine-code that can be executed by the computer [2].

6) Test and debug your compiler: Once the various components of the compiler are implemented, it will be important to thoroughly test it to ensure that it is functioning correctly and producing the desired output [9].

Fig:3 explains the final step of the compilation process and Fig:4 shows the grammar used to implement the programming language which has lexer which accepts normal identifiers as a general-purpose programming as well as $-prefixed identifiers.

```
1   statements : NEWLINE* statement (NEWLINE+ statement)* NEWLINE*
2
3   statement  : KEYWORD:RETURN expr?
4              : KEYWORD:CONTINUE
5              : KEYWORD:BREAK
6              : expr
7
8   expr       : KEYWORD:VAR IDENTIFIER EQ expr
9              : comp-expr ((KEYWORD:AND|KEYWORD:OR) comp-expr)*
10
11  comp-expr  : NOT comp-expr
12             : arith-expr ((EE|LT|GT|LTE|GTE) arith-expr)*
13
14  arith-expr : term ((PLUS|MINUS) term)*
15
16  term       : factor ((MUL|DIV) factor)*
17
18  factor     : (PLUS|MINUS) factor
19             : power
20
21  power      : call (POW factor)*
22
23  call       : atom (LPAREN (expr (COMMA expr)*)? RPAREN)?
24
25  atom       : INT|FLOAT|STRING|IDENTIFIER
26             : LPAREN expr RPAREN
27             : list-expr
28             : if-expr
29             : for-expr
30             : while-expr
31             : func-def
32
33  list-expr  : LSQUARE (expr (COMMA expr)*)? RSQUARE
34
35  if-expr    : KEYWORD:IF expr KEYWORD:THEN
36               (statement if-expr-b|if-expr-c?)
37             | (NEWLINE statements KEYWORD:END|if-expr-b|if-expr-c)
38
39  if-expr-b  : KEYWORD:ELIF expr KEYWORD:THEN
40               (statement if-expr-b|if-expr-c?)
41             | (NEWLINE statements KEYWORD:END|if-expr-b|if-expr-c)
42
43  if-expr-c  : KEYWORD:ELSE
44               statement
45             | (NEWLINE statements KEYWORD:END)
46
47  for-expr   : KEYWORD:FOR IDENTIFIER EQ expr KEYWORD:TO expr
48               (KEYWORD:STEP expr)? KEYWORD:THEN
49               statement
50             | (NEWLINE statements KEYWORD:END)
51
52  while-expr : KEYWORD:WHILE expr KEYWORD:THEN
53               statement
54             | (NEWLINE statements KEYWORD:END)
55
56  func-def   : KEYWORD:FUN IDENTIFIER?
57               LPAREN (IDENTIFIER (COMMA IDENTIFIER)*)? RPAREN
58               (ARROW expr)
59             | (NEWLINE statements KEYWORD:END)
60
```

Fig. 4. GRAMMAR USED FOR PROPOSED
PROGRAMMING LANGUAGE

Fig:5, Fig:6, Fig:7 shows that there's improvement in time consumed by the lexical analyzer in proposed programming language when the identifiers are prefixed (with '$') and not prefixed for different number of identifiers in the program script.

From Fig:8, Fig:9, Fig:10 one can see significant improvement in Number of iterations taken to identify

whether a given lexeme is a keyword token or an identifier token.

5. Results and Discussion
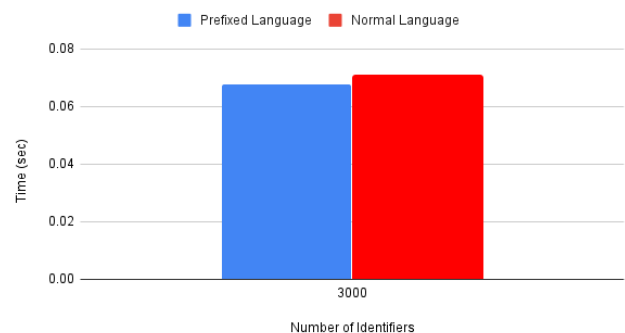
Fig. 5. NO OF IDENTIFIERS VS TIME
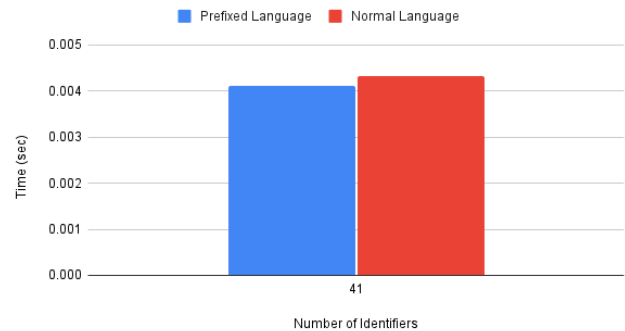

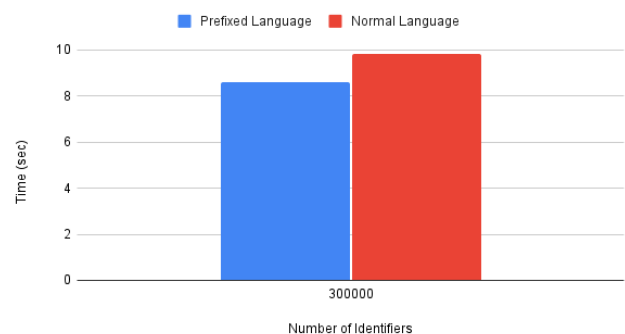**Fig. 6.** NO OF IDENTIFIERS VS TIME

Fig. 7. NO OF IDENTIFIERS VS TIME

Let's say that to classify 'varName' as a keyword or an identifier, if it's in non-prefixed identifier language then it would take k number of iterations, where k is number of keywords in the programming language. whereas in the proposed prefixed language it doesn't need to be compared with any keywords if it's prefixed with '$' so authors take that as 1 iteration.
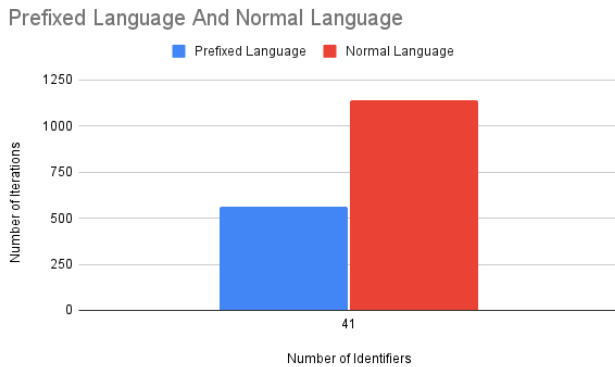

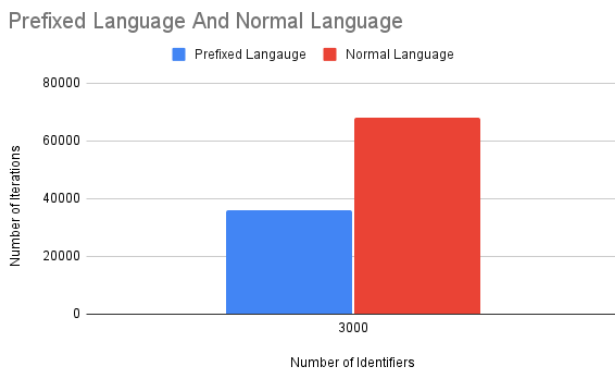**Fig. 8.** NO OF IDENTIFIERS VS ITERATIONS
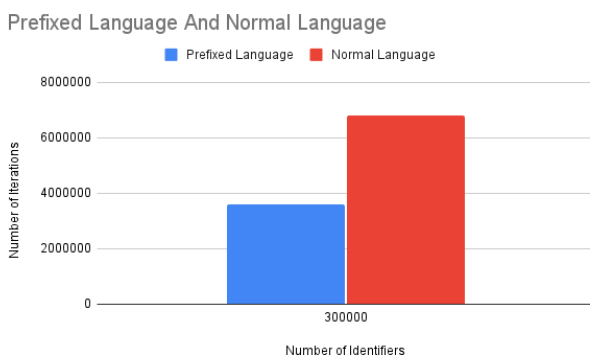

Fig. 9. NO OF IDENTIFIERS VS ITERATIONS

Fig. 10. NO OF IDENTIFIERS VS ITERATIONS

So, in this way authors are able to improve time taken and Iterations taken to classify identifiers and keywords when identifiers are $-prefixed.

- All concepts of a high-level programming language such as Classes, Structures, Inheriting, polymorphism etc. are not implemented in it.
- Ambiguity: Context-free grammar is sometimes ambiguous, which means that they allow multiple parse trees to be constructed for the same input string. This can lead to difficulty in determining the intended meaning of the input and can make it harder to generate the correct code.
- Limited readability: The use of $-prefixed identifiers can make the code less readable and harder to understand for new developers who are not familiar with the language.
- Reserved words of the programming language as identifiers unlike other languages which do not allow such flexibility can be used.
- Number of iterations to identify an identifier in a programming language is significantly decreased which will lead to decreasing the time taken for lexical analysis when more keywords are there in a programming language.
- The adaptation of such structured language is comparatively easier.
- The concept proposed to make lexer efficient can be adapted to any existing general purpose programming language.

6. Advantages

- We can use reserved words of the programming language as an identifier unlike other languages which do not allow such flexibility.

- Number of iterations to identify an identifier in a programming language is significantly decreased which will lead to decreasing the time taken for lexical analysis when more keywords are there in a programming language.

- The adaptation of such structured language is comparatively easier.

- The concept that we used to make lexer efficient can be adapted to any existing general purpose programming language.

7. Conclusion

In this research, we understood that there are many different approaches to compiler design, and context free

grammars play a vital role in many of them. By using context-free grammar to define the structure of a programming language, compilers can parse and analyze the source code to ensure that it is syntactically correct and semantically meaningful. This is a critical step in the compilation process, as it enables the compiler to generate efficient machine code that can be executed by the computer. We divide the compiler into different phases such as Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Code Optimization, Target Code Generation to optimize the compiler in each level to obtain a compiler which is ideal in terms of stability, performance, time-complexity etc.

We can optimize the phase of Lexical Analysis by adding an extra '$' character for every occurrence of an Identifier in the source code, which will lead to a significant improvement in time and Number of iterations taken to identify keywords and identifiers.

This concept which we used on Lexer of our own Programming Language can be applied to any existing programming language.

8. Future Scope

A programming language that uses $prefixed identifiers to improve lookup time has the potential to be useful in a variety of contexts where fast and efficient token lookup is important. Here are a few potential applications:

1) Game development: A language with fast and efficient variable lookup could be useful in the field of game development, where performance is critical. For example, you could use a $-prefixed variable to represent a game object or character, making it easy to access and modify these values as needed.

2) Scientific computing: A language with fast and efficient variable lookup could be useful in the field of scientific computing, where complex simulations and calculations require fast and efficient access to variables. For example, you could use a $-prefixed variable to represent a physical parameter or simulation variable, making it easy to access and modify these values as needed [6].

3) Performance-oriented applications: If the language is designed to be highly performant and efficient, it may find a niche in applications where speed and resource usage are critical, such as scientific computing or high frequency trading.

4) Unique selling point: If the language is able to differentiate itself from other languages in some way beyond just its use of $-prefixed identifiers, such as through its ease of use, unique features, or compatibility with existing systems, it may find a wider audience and broader scope.

Overall, the future scope of a language like this would depend on its specific features and use cases, as well as its adoption and support by the developer community. However, the

potential for improved performance and efficiency in certain domains could make it a compelling option for developers looking for a more streamlined and efficient programming experience. The potential applications of a programming language that uses $prefixed identifiers are wide-ranging and varied. The specific future scope will depend on the features and capabilities of the language, as well as its adoption and support by the developer community.

9. References

[1]. Appel, A. W. (2004). Modern compiler implementation in C. Cambridge university press.

[2]. Aho, A. V., Sethi, R., & Ullman, J. D. (2007). Compilers: principles, techniques, and tools (Vol. 2). Reading: Addison-wesley.

[3]. Sanju, V. (2016, March). An exploration on lexical analysis. In 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT) (pp. 253-258). IEEE. DOI: 10.1109/ICEEOT.2016.7755127

[4]. Chhabra, J., Chopra, H., & Vats, A. (2014). Research paper on Compiler Design. International Journal of Innovative Research in Technology, 1(5), 151-153.

[5]. Haili Luo The Research of Applying Regular Grammar to Making Model for Lexical Analyzer, Proceedings of IEEE 6th International Conference on Information Management, Innovation Management & Industrial Engi-neering, pp 90-92 , 2013. DOI: 10.1109/ICIII.2013.6703245

[6]. Sanju, V. (2016, March). An exploration on lexical analysis. In 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT) (pp. 253-258). IEEE. DOI: 10.1109/ICEEOT.2016.7755127

[7]. Wu, X., Mernik, M., Bryant, B. R., & Gray, J. (2009). Implementation of Programming Languages Syntax and Semantics. In Encyclopedia of Information Science and Technology, Second Edition (pp. 1863-1869). IGI Global DOI: 10.4018/978-1-60566-026-4.ch293

[8]. Aho, A. V., & Ullman, J. D. (1973). The theory of parsing, translation, and compiling (Vol. 1, p. 309). Englewood Cliffs, NJ: Prentice-Hall.

[9]. Abubakar, B. S., Ahmad, A., Aliyu, M. M., Ahmad, M. M., & Uba, H. U. (2021). An Overview of Compiler Construction. Int. Res. J. Eng. Technol., 8(3), 578-590.

[10]. Chen, H., Ching, W. M., & Hendren, L. (2017, June). An ELI-to-C compiler: design, implementation, and performance. In Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (pp. 9-16). https://doi.org/10.1145/3091966.3091969

[11]. Pai T, V., & Aithal, P. S. (2020). A Systematic Literature Review of Lexical Analyzer Implementation Techniques in Compiler Design. International Journal of Applied Engineering and Management Letters (IJAEML), 4(2), 285 301.

[12]. Bhosale, V., & Chaudhari, S. R. (2015). Fuzzy Lexical Analyser: Design and Implementation. International Journal of Computer Applications, 123(11). DOI:10.5120/ijca2015905567

[13]. Marowka, A. (2004, July). Analytic comparison of two advanced c language-based parallel programming models. In Third International Symposium on Parallel and Distributed

Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (pp. 284-291). IEEE. DOI:10.1109/ISPDC.2004.11

[14]. 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 (COMP- SAC'97) (pp. 232-239). IEEE. Doi:10.1109/CMPSAC.1997.624802

[15]. 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. https://doi.org/10.1504/ijhpcn.2016.076270

[16]. Barve, A., & Joshi, B. K. (2015). Improved Parallel Lexical Analysis using OpenMP on Multi-core Machines. Procedia Computer Science, 49, 211-219. https://doi.org/10.1016/j.procs.2015.04.246

[17]. 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. https://doi.org/10.4018/IJKBO.2018010105

[18]. Barve, A., & Joshi, B. K. (2012, December). Parallel lexical analysis on multi-core machines using divide and conquer. In 2012 Nirma University International Conference on Engineering (NUiCONE) (pp. 1-5). IEEE. DOI: 10.1109/NUICONE.2012.6493218

[19]. Srikanth, G. U. (2010, June). Parallel lexical analyzer on the cell processor. In 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion (pp. 28-29). IEEE. https://doi.org/10.1109/SSIRI-C.2010.16

[20]. Barve, A., Khomane, S., Kulkarni, B., Ghadage, S., & Katare, S. (2017, December). Parallelism in C++ programs targeting objects. In 2017 International Conference on Advances in Computing, Communication and Control (ICAC3) (pp. 1-6). IEEE. DOI: 10.1109/ICAC3.2017.8318759

[21]. Marowka, A. (2008, December). Towards high-level parallel programming models for multicore systems. In 2008 Advanced Software Engineering and Its Applications (pp. 226-229). IEEE. https://doi.org/10.1109/ASEA.2008.9