Compiler Design Using Context-Free Grammar

Arpit Patil, Saloni Khedekar, Omkar Vyavhare, Rahul Dound

ABSTRACT

Compiler design is a crucial aspect of computer science, as it enables the translation of high-level programming languages into machine-readable code. One approach to compiler design is to use context-free grammars (CFGs) to specify the syntax of a programming language. In this research paper, we will explore the use of CFGs in compiler design and their benefits and limitations. We will first provide an overview of CFGs and their role in compiler design, including a description of the formal definition of a CFG and the processes involved in creating a CFG for a programming language. Additionally, we will examine the challenges and limitations of using CFGs in compiler design, including issues of ambiguity and efficiency, and discuss potential solutions and alternatives.

Keywords

Compiler Design, Context-Free Grammar, Python, Lexing, Parsing.

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. In order to translate these programs into machine code, compilers must be able to accurately parse and understand the syntax and structure of the source language. 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. 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. In this research paper, we will delve into the role of CFGs in compiler design and discuss the various techniques and algorithms that are used to implement them. We will also explore the challenges and limitations of using CFGs in compiler design, including issues of ambiguity and efficiency, and discuss potential solutions and alternatives examining the role of CFGs in compiler design, we aim to provide a better understanding of how these tools work and how they can be used to effectively parse and understand programming languages.

2. LITERATURE REVIEW

Noam Chomsky [1] 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.

Alfred Aho and Jeffrey Ullman [2] in their book, provide a thorough overview of the use of CFGs in compiler design, including the various techniques and algorithms that are used to implement them. 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. Aho and Ullman also address the challenges and limitations of using CFGs in compiler design, including issues of ambiguity and efficiency. The solutions to these challenges, such as using disambiguation rules and incorporating additional formalisms, such as attribute grammar, to improve the expressiveness and precision of CFGs.

Steven Johnson[3] is known for his work on the development of the Yacc parser generator, which is used to automatically generate parsers from context-free grammars (CFGs). Yacc, which stands for "Yet Another Compiler Compiler," is a widely used tool in compiler design and has been influential in the development of many programming languages. Johnson's work on Yacc has had a significant impact on the field of compiler design, as it provides a convenient and efficient way of implementing CFGs in compiler design. Using Yacc, programmers can specify the syntax of a programming language using a CFG and then automatically generate a parser that can check the syntax and semantics of programs written in that language.

Andrew Appel [4] who is known for his work on the development of the Tiger programming language and the accompanying compiler, which was used as a case study to demonstrate the use of context-free grammars (CFGs) in compiler design. In his work on the Tiger compiler, Appel demonstrated how CFGs can be used to effectively define the syntax of a programming language and generate parse trees that can be used to check the syntax

and semantics of programs written in that language. He also addressed the challenges and limitations of using CFGs in compiler design, including issues of ambiguity and efficiency, and discussed potential solutions and alternatives.

Frank DeRemer and Tom Pennello [5] are computer scientists who are known for their work on bottom-up parsing and the development of the LL(k) and LR(k) parsing algorithms, which are widely used in compiler design. Bottom-up parsing involves constructing a parse tree from the leaves and expanding it towards the root, using the rules of context-free grammar rules the input program into its underlying syntactic structure. The LL(k) and LR(k) algorithms are two common techniques for implementing bottom-up parsing and have been widely used in compiler design due to their ability to handle left recursion and efficiently parse large programs.

3. METHODOLOGY

For this project, Python is used to design a compiler and our own programming language. To design our own programming language and compiler following techniques are used:

1. Lexical analysis: This is the process of breaking the source code into a sequence of tokens, which are the basic units of the programming language.

2. Syntax analysis: This is the process of checking the source code for correct syntax, using grammar rules.

3. Semantic analysis: This is the process of checking the source code for meaning and ensuring that it is semantically correct.

4. Intermediate code generation: This is the process of generating an intermediate representation of the source code, which is easier for the compiler to work with.

5. Code optimization: This is the process of improving the efficiency of the generated code by making it run faster or use fewer resources.

6. Code generation: This is the process of generating machine code from the intermediate representation of the source code.

3.1 Proposed System

This Compiler design using a Context-Free Grammar project helps to create a custom programming language using our own set of rules and syntax. We have declared our own Context Free Grammar i.e. our own set of rules and have developed our own compiler to compile that programming language and display the desired output.

Overall, the process of a compiler is breaking down the input program into its individual tokens, generating a parse tree to represent the syntax of the program, check the semantics of the program, generating machine code, and assembling the machine code into a form that can be executed. These steps are typically carried out by different components of the compiler, such as the lexer, parser, semantic analyzer, code generator, and assembler.

Steps in designing the compiler and language:

Step 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.

Step 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.

Step 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. We used a parsing algorithm, such as top-down or bottom-up parsing, to construct the parse tree. To implement the parser, we write code that can apply the rules of the CFG to the input program and generate the parse tree.

Step 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. To implement the semantic analyzer, we write a code that can check the semantics of the program and report any errors that are found.

Step 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. To implement the code generator, we write code that can traverse the parse tree and generate the corresponding machine code.

Step 6: Test and debug your compiler: Once we have implemented the various components of our compiler, it will be important to thoroughly test it to ensure that it is functioning correctly and producing the desired output. For that, we have debugged our compiler to fix any issues that arise.

3.2 Flowchart

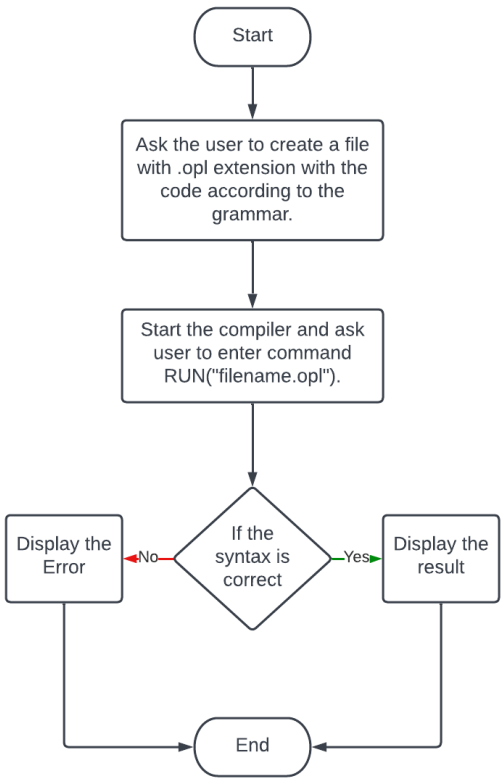

Fig 1. Flowchart of the project

4. RESULTS AND DISCUSSIONS

All the above process is used in designing this language's grammar and compiler.

The user needs to first create a file with the program as per the given syntax and rules. Then need to save it with the .opl extension the user needs to start the compiler and enter the command RUN("filename.opl").

After doing this, if the syntax and rules are correct then the desired output will get displayed.

```
PRINT("Fibonacci Sequence is: ")
    VAR num1 = 0
    VAR num2 = 1
    PRINT(num1)
    PRINT(num2)

    FOR i = 0 TO 10 THEN
        VAR num3 = num1 + num2
        PRINT(num3)
        VAR num1 = num2
        VAR num2 = num3
    END
```

Fig 2. Program to print Fibonacci series

```
compiler >>> RUN("fibonacci.opl")
Fibonacci Sequence is:
0
1
1
2
3
5
8
13
21
34
55
89
```

Fig 3. The output of the program

```
statements  : NEWLINE* statement (NEWLINE+ statement)* NEWLINE*

statement   : KEYWORD:RETURN expr?
            : KEYWORD:CONTINUE
            : KEYWORD:BREAK
            : expr

expr        : KEYWORD:VAR IDENTIFIER EQ expr
            : comp-expr ((KEYWORD:AND|KEYWORD:OR) comp-expr)*

comp-expr   : NOT comp-expr
            : arith-expr ((EE|LT|GT|LTE|GTE) arith-expr)*

arith-expr  : term ((PLUS|MINUS) term)*

term        : factor ((MUL|DIV) factor)*

factor      : (PLUS|MINUS) factor
            : power

power       : call (POW factor)*

call        : atom (LPAREN (expr (COMMA expr)*)? RPAREN)?

atom        : INT|FLOAT|STRING|IDENTIFIER
            : LPAREN expr RPAREN
            : list-expr
            : if-expr
            : for-expr
            : while-expr
            : func-def
```

Fig 4. Grammar of the programming language

```
list-expr   : LSQUARE (expr (COMMA expr)*)? RSQUARE

if-expr     : KEYWORD:IF expr KEYWORD:THEN
              (statement if-expr-b|if-expr-c?)
            | (NEWLINE statements KEYWORD:END|if-expr-b|if-expr-c)

if-expr-b   : KEYWORD:ELIF expr KEYWORD:THEN
              (statement if-expr-b|if-expr-c?)
            | (NEWLINE statements KEYWORD:END|if-expr-b|if-expr-c)

if-expr-c   : KEYWORD:ELSE
              statement
            | (NEWLINE statements KEYWORD:END)

for-expr    : KEYWORD:FOR IDENTIFIER EQ expr KEYWORD:TO expr
              (KEYWORD:STEP expr)? KEYWORD:THEN
              statement
            | (NEWLINE statements KEYWORD:END)

while-expr  : KEYWORD:WHILE expr KEYWORD:THEN
              statement
            | (NEWLINE statements KEYWORD:END)

func-def    : KEYWORD:FUN IDENTIFIER?
              LPAREN (IDENTIFIER (COMMA IDENTIFIER)*)? RPAREN
              (ARROW expr)
            | (NEWLINE statements KEYWORD:END)
```

Fig 5. Grammar of the language

5. LIMITATIONS

- Only Limited programs can be implemented using this language.
- 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.

6. Conclusion

In this project, we understood that there are many different approaches to compiler design, and context-free grammars play a central 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.

7. Future Scope

- Improved parsing algorithms: Currently, most compilers use variations of the top-down or bottom-up parsing algorithms. However, there is room for improvement in these algorithms, as well as the development of new parsing algorithms that are more efficient and effective.
- Language extensions: As programming languages continue to evolve, there is a need for compilers that can handle new language constructs and features. Research in this area could focus on extending context-free grammar to support these new language features.

8. References

[1] Kuldeep Vayadande, Aditya Bodhankar, Ainkya Mahajan, Diksha Prasad, Shivani Mahajan, Aishwarya Pujari and Riya Dhakalkar, “Classification of Depression on social media using Distant Supervision”, ITM Web Conf. Volume 50, 2022

[2] Kuldeep Vayadande, Rahebar Shaikh, Suraj Rothe, Sangam Patil, Tanuj Baware and Sameer Naik,” Blockchain-Based Land Record SysteM”, ITM Web Conf. Volume 50, 2022.

[3] Kuldeep Vayadande, Kirti Agarwal, Aadesh Kabra, Ketan Gangwal and Atharv Kinage,” Cryptography using Automata Theory”, ITM Web Conf. Volume 50, 2022

[4] Samruddhi Mumbare, Kunal Shivam, Priyanka Lokhande, Samruddhi Zaware, Varad Deshpande and Kuldeep Vayadande,”Software Controller using Hand Gestures”, ITM Web Conf. Volume 50, 2022

[5] Preetham, H. D., and Kuldeep Baban Vayadande. "Online Crime Reporting System Using Python Django."

[6] Vayadande, Kuldeep B., et al. "Simulation and Testing of Deterministic Finite Automata Machine." *International Journal of Computer Sciences and Engineering* 10.1 (2022): 13-17.

[7] Vayadande, Kuldeep, et al. "Modulo Calculator Using Tkinter Library." *EasyChair Preprint* 7578 (2022).

[8] VAYADANDE KULDEEP. “Simulating Derivations of Context-Free Grammar”(2022).

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

[10] Vayadande, Kuldeep, Ritesh Pokarne, Mahalakshmi Phaldesai, Tanushri Bhuruk, Tanmay Patil, and Prachi Kumar. "Simulation Of Conway's Game Of Life Using Cellular Automata." *SIMULATION* 9, no. 01 (2022).

[11] Gurav, Rohit, Sakshi Suryawanshi, Parth Narkhede, Sankalp Patil, Sejal Hukare, and Kuldeep Vayadande. "Universal Turing machine simulator." *International Journal of Advance Research, Ideas and Innovations in Technology, ISSN* (2022).

[12] 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.

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

[14] Vayadande, Kuldeep B., and Surendra Yadav. "A Review paper on Detection of Moving Object in Dynamic Background." *International Journal of Computer Sciences and Engineering* 6, no. 9 (2018): 877-880.

[15] 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.

[16] Vayadande, Kuldeep, Harshwardhan More, Omkar More, Shubham Mulay, Atharva Pathak, and Vishwam Talnikar. "Pac Man: Game Development using PDA and OOP." (2022).

[17] Preetham, H. D., and Kuldeep Baban Vayadande. "Online Crime Reporting System Using Python Django."

[18] Vayadande, Kuldeep. "Harshwardhan More, Omkar More, Shubham Mulay, Atahrv Pathak, Vishwam Talanikar,"Pac Man: Game Development using PDA and OOP"." *International Research Journal of Engineering and Technology (IRJET), e-ISSN* (2022): 2395-0056.

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

[21] 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.