Translating Custom Language to Assembly: A Python-Based Parser, Lexer, and Compiler

1st Kevin Huy Trinh
St. Mary's University
Ronald E. McNair Program
San Antonio, United States
kevintrinh1227@gmail.com

Abstract*—This paper presents a Python-based toolset for translating custom language code into assembly language for a virtual machine. The toolset includes a lexer module for tokenizing the code, a parser module for generating an abstract syntax tree (AST), and a compiler or semantic analyzer module for translating the AST into assembly language instructions. The research focuses on the design and implementation of these components, utilizing top-down recursive parsing. Extensive testing ensures accurate translation and execution of custom language code. The toolset's flexibility enables future enhancements and support for diverse virtual machine architectures. The results demonstrate successful translation, highlighting the power and versatility of the developed toolset. This research advances language processing and compiler design, facilitating the seamless execution of domain-specific languages on virtual hardware platforms.

***Keywords*—Language processing, Custom language, Parser, Lexer, Semantic Analyzer, Virtual machine**

I. INTRODUCTION

A. Research Topic and Background

Computer programming languages are essential tools in the field of software development. They enable programmers to develop a wide range of systems and applications that are used in our daily lives. These languages have allowed innovators to communicate their ideas to machines in a human-readable format. However, computers can only understand instructions in a specific format known as machine code or machine language. Machine code is a low-level language that computers can directly understand and execute and can be exhibited in diverse variations consisting of but not limited to binary, assembly language, hexadecimal, and octal.

B. Research Objective and Significance

The research objective is to develop a comprehensive toolset for translating code written in the custom language into assembly language for a virtual machine. By designing and implementing a robust parser, lexer, and compiler for a general-purpose custom language, this research aims to enable seamless execution of programs on virtual hardware platforms. This toolset will empower developers to efficiently create and execute specialized applications, expanding the possibilities of application development and enhancing software performance. The significance of this research lies in its potential to bridge the gap between high-level custom languages and low-level machine code, offering a practical solution for efficient code translation and execution. By addressing this challenge, this research contributes to the advancement of language processing and compiler design, facilitating the development of domain-specific languages and their seamless execution on virtual hardware platforms.

C. Research Approach and Methodology

The project follows a systematic approach to designing and implementing a comprehensive toolset for translating code written in a custom general-purpose language into assembly language using Python. The methodology involves several key steps. Firstly, the project begins with the design phase, where the lexer module is specified to identify meaningful tokens based on the predefined grammar rules and regular expressions. Once the lexer was developed, a parser module was implemented to generate an abstract syntax tree (AST), representing the hierarchical structure of the code according to the grammar defined in Backus-Naur Form (BNF). Next, the compiler or semantic analyzer module is implemented to traverse and translate the AST into assembly language instructions specific to a virtual machine. Extensive testing is conducted using a comprehensive set of test cases that cover various grammar and semantic aspects of the custom language. The testing phase ensures accurate translation and execution of custom language code into assembly language, validating the effectiveness and reliability of the parser, lexer, and compiler modules. By following this project approach and methodology, a robust and efficient toolset is created for seamless code translation and execution using Python, specifically tailored for the custom language developed.

II. LITERATURE REVIEW

A. Language Design and Parsing

In the process of designing a programming language, a crucial aspect to consider is the language's grammar. The grammar, often defined using a formal notation such as Backus-Naur Form (BNF), dictates the set of rules that determine syntactically valid programs in that language [1]. These rules provide the structured blueprint that a parser will use to interpret the code.

```
<assignment> ::= <type> <identifier> "=" <value>
<type> ::= "STRING" | "INT" | "FLOAT"
<identifier> ::= [a-zA-Z_]\w*
<value> ::= <string> | <number>
<string> ::= "<text>"
<number> ::= <integer> | <float>
<integer> ::= \d+
```

Fig. 1. This figure shows the BNF rules that consist of variable assignment statements and value representation, including STRING, INT, FLOAT types, identifiers, strings, and numbers specific to the project.

Parsing is a fundamental component of a compiler or interpreter, It's the stage that comes after lexical analysis (or 'lexing'), where the input code is divided into meaningful tokens. The parser takes these tokens and, using the language's grammar constructs an Abstract Syntax Tree (AST). The AST represents the hierarchical structure of the program, and this tree-like representation is used in the subsequent stages of the compilation or interpretation process [2].

There are various parsing techniques that can be employed, often categorized as either top-down or bottom-up approaches. Top-down parsers, like Recursive Descent parsers, start at the root of the AST and work their way down, while bottom-up parsers, like shift-reduce parsers, start at the leaves and work their way up [3]. The choice of parsing technique can depend on factors such as the complexity of the language's grammar and the desired efficiency of the parser.

Significant prior work in the field of parsing includes the development of parsing algorithms like Earley's algorithm, and tools like YACC (Yet ANother Compiler-Compiler) or ANTLR (Another Tool for Language Recognition), which are parser generators [4]. These works have contributed to shaping the current landscape of compiler design and have informed the methods used in this project, as discussed in subsequent sections.

B. *Lexical Analysis*

Lexical analysis, also known as lexing, is an integral part of the compilation process, acting as the first phase of translating code. It takes raw source code as input and breaks it into meaningful chunks or tokens. These tokens can include various types such as identifiers, keywords, separators, literals, and operators, among others [5].

One of the primary tools used in the lexing process is regular expressions. Regular expressions provide a means to describe patterns in text, making them ideally suited for identifying the different types of tokens in source code based on their patterns. Lexers often implement finite automata, deterministic or non-deterministic, as a mechanism to recognize these patterns and categorize the input text into the corresponding tokens [5].

There exist numerous tools and techniques for performing lexical analysis, with some of the most prevalent being tools like Lex, Flex, or JLex. These are known as lexer or scanner generators, taking as input a file containing regular expressions and corresponding actions, and outputting code for a lexer that performs the specified actions when it encounters matches for the expressions [6].

Significant work in the field of lexical analysis has provided various strategies and methodologies for tokenizing code. These range from techniques for handling ambiguous token definitions to ways of dealing with language-specific quirks in the lexing process [6]. This existing body of knowledge has significantly shaped the approach taken in this project, as will be discussed in later sections.

C. *Semantic Analysis and Code Generation in Compiler Design and Optimization*

The process of compiler design is an intricate one, involving several stages to transform high-level source code into machine-readable instructions. A compiler takes the tokens generated by the lexical analyzer and, through syntax and semantic analysis, generates an intermediate representation of the code. This intermediate representation is then optimized and finally transformed into machine code [5]. Each stage of the compiler plays a crucial role in generating efficient and correct machine code. Among these stages, the semantic analyzer and code generation stand out for their roles in improving the performance of the resulting program. The semantic analyzer ensures the correct interpretation of the code and performs static checks, while the code generator transforms the intermediate representation into machine code [5]. Compiler optimizations aim to enhance the runtime speed, reduce binary size, or decrease power consumption, all while maintaining the program's original functionality. These optimizations can happen at various levels, including the intermediate code level and the machine code level, and can involve techniques such as dead code elimination, loop optimization, and instruction scheduling [5]. There exist numerous techniques and tools for compiler design, including widely used compilers like GCC, LLVM, and Java compiler. These tools have shaped the field of compiler design and provided robust, efficient mechanisms for translating high-level languages into machine code [5]. Significant prior work in compiler design has led to the development of various methodologies for managing the complexity of translating high-level code into efficient machine code. This research project builds upon these existing techniques to create a compiler tailored to the custom language developed.

D. *Python in Compiler Design*

Python, a high-level, interpreted programming language, is renowned for its simplicity and wide usage. Its straightforward syntax and semantics make it an excellent choice for a myriad of applications, particularly in fields that require rapid development and testing of complex algorithms [7].

A major advantage of Python is its vast array of libraries and tools that facilitate various aspects of programming. For tasks related to language processing, Python provides several built-in libraries for string processing, regular expressions, and file I/O. These tools greatly simplify the process of reading source code, identifying tokens, and writing output files.

Python's capabilities make it particularly well-suited to tasks related to language processing. Its powerful string manipulation features and pattern-matching capabilities simplify the implementation of complex language processing algorithms. Additionally, Python's clear and concise syntax promotes readable and maintainable code, a significant advantage when designing and implementing the complex structures often found in compilers.

In this project, Python was utilized to design and implement the parser, lexer, and compiler for the custom language. The simplicity of Python allowed for rapid prototyping and testing of different language features and compiler designs. Python's rich set of libraries simplified many aspects of the project, from reading and tokenizing the

source code to writing the generated assembly code. Furthermore, the readability of Python code greatly facilitated the process of debugging and refining the compiler or semantic analyzer [7].

III. Design and Implementation

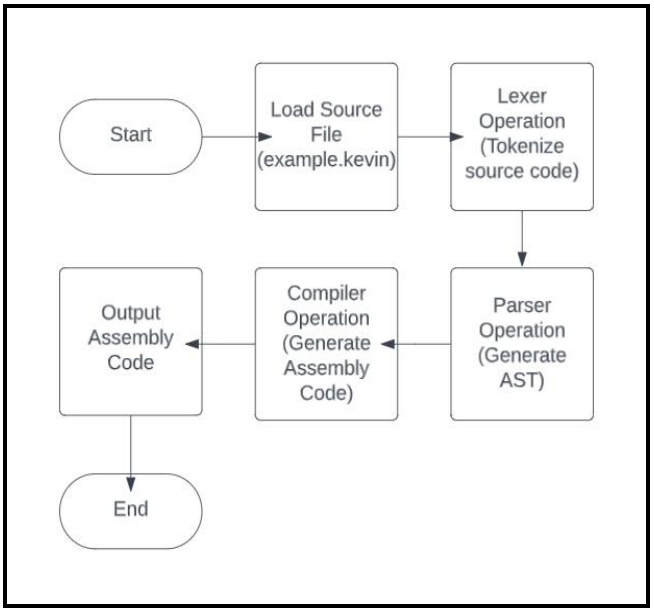

Fig. 2. Flowchart depicting the translation process. The source code from the 'example.kevin' file undergoes lexing, parsing, and compiling, resulting in the output of assembly language.

A. Lexer Design and Implementation

The initial step in translating the custom language into assembly language involved designing and implementing a lexer. The lexer, alternatively referred to as a tokenizer or scanner, is responsible for partitioning the input sequence into token strings. The significance of this component cannot be overstated, as it enables the subsequent elements of the compiler, namely the parser and the code generator, to handle the code in a structured and manageable fashion.

The custom language was read from a file with the extension ".kevin". Each line of code was read individually to ensure that each statement was handled separately, which helped maintain the order of execution of the code.

The first task of the lexer was to split the input line into individual words or components. This was accomplished by slicing the string at each whitespace character and storing the resulting substrings in a list. This method enabled me to separate individual elements of the code like keywords, identifiers, operators, and values, each of which plays a crucial role in the meaning of the code.

Following the initial splitting of the input line, each item in the list was then tokenized. Tokenization involved categorizing each substring into a type that could be understood by the subsequent stages of the semantic analyzer. For instance, keywords like "if", "else", and "while" were recognized and classified, and identifiers were separated from their associated values.

The design and implementation of the lexer were not without their challenges. One of the primary challenges was ensuring that the lexer accurately recognized all components of the code, especially with respect to more complex constructs like multi-character operators or identifiers with special characters. Resolving this issue required thorough testing and fine-tuning of the regular expressions used for tokenization.

Another challenge was handling errors in the input code. While the lexer aimed to be robust and handle as many scenarios as possible, there were cases where the input code did not conform to the expected structure. This necessitated the design of error-handling mechanisms to inform the user about the nature of the error and where it occurred in the code [5].

Despite these challenges, the successful implementation of the lexer provided a solid foundation for the rest of the compiler. The lexer served as a bridge, translating the free-form structure of the custom language into a more rigid and easily processed format that could be used by the next stages of the compiler.

B. Parser Design and Implementation

After lexing, the next step in translating the custom language into assembly language was to parse the tokenized output. The parser's primary role is to check the code for syntactic correctness and generate an abstract syntax tree (AST) to capture the hierarchical relationship between different parts of the code.

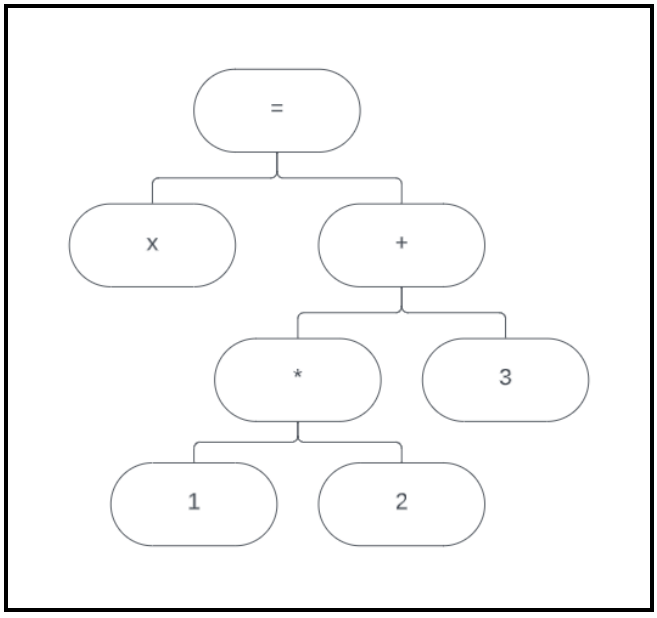

Fig. 3. Example of a simple Abstract Syntax Tree (AST) for a simple math expression "1 * 2 + 3". The AST illustrates the hierarchical structure of the expression, demonstrating the multiplication and addition operations along with their corresponding values.

In the context of the project, a top-down parsing technique known as Recursive Descent Parsing was implemented, as mentioned earlier. The parser takes the list of tokens produced by the lexer as input and recursively matches the tokens against the grammar rules of the custom language. Each token is examined to determine its type (e.g., keyword, operator, identifier), and a corresponding node is created in the AST. The parser ensures that the tokens

comply with the grammar rules, thereby confirming the syntactic correctness of the input code [5].

The creation of the AST was a crucial part of the parsing process. This tree-like data structure allowed me to capture the hierarchical relationship between different parts of the code. For instance, in an assignment statement, the variable being assigned a value would be a parent node, with the assigned value or expression being a child node.

The design and implementation of the parser were not without challenges. Handling syntax errors in the input code was a significant challenge. Unclosed brackets or missing semicolons could disrupt the parsing process and lead to an incorrect AST. To address this, error-handling mechanisms were implemented in the parser to detect syntax errors and report them to the user, indicating the type and location of the error in the code [5].

Another challenge was ensuring that the AST correctly represented the hierarchical structure of the code, especially for complex constructs like nested if-else statements or complex expressions. However, through careful design and extensive testing, it was ensured that the parser correctly built the Abstract Syntax Tree (AST) for a wide range of code constructs.

The successful design and implementation of the parser using Recursive Descent Parsing represented a significant milestone in the project. With the Abstract Syntax Tree (AST) in place, the next stage of the process, the compiler, could be initiated.

C. Compiler Design and Implementation

The last crucial component in the translation pipeline of the custom language is the compiler. Its role was to translate the abstract syntax tree (AST) generated by the parser into assembly language instructions that could be executed by the target virtual machine.

The design of the compiler was intimately tied to the specifics of both the source language (the custom language) and the target language (the assembly language for the virtual machine). For each type of node in the Abstract Syntax Tree (AST), a corresponding rule was defined in the compiler to govern its translation into assembly code [8].

The compiler was implemented in Python and worked by traversing the AST generated by the parser. For each node encountered during this traversal, the compiler produced the corresponding assembly code according to the translation rules defined.

The process of implementing the compiler posed several challenges. One major challenge was dealing with language constructs that have no direct equivalent in the target assembly language. For instance, high-level control structures (like loops or conditional branches) had to be translated into sequences of low-level jumps and comparisons.

Another challenge was managing the allocation and deallocation of memory on the virtual machine. A strategy had to be devised to efficiently handle memory management and ensure the correct execution of the generated assembly code.

Despite these challenges, the implementation of a compiler capable of translating a wide range of custom language constructs into assembly language was successfully achieved. This marked the final step in the process of translating code written in the custom language into a form that could be executed by a virtual machine, thus fulfilling the main objective of this research project.

IV. Testing and Results

A. Testing Procedures

A thorough testing process is indispensable in the development of any language processing tool, and this project was no exception. The objective of this testing phase was to authenticate the functionality of the toolset and identify potential areas for improvement.

The custom language was put through a rigorous set of test cases to check its syntax and semantics. The test cases varied from simple programs that evaluated individual language features to complex programs that integrated multiple features.

During the testing of the lexer, each generated token was printed out. This allowed for a detailed visual verification process, ensuring that the tokenized output adhered to the syntax of the custom language.

Similarly, for the parser, tests were executed to confirm that it could build an accurate AST from a range of code constructs and accurately identify syntax errors.

The compiler, which translates the AST into assembly language instructions, underwent a similarly exhaustive testing procedure. The generated assembly code was executed on the target virtual machine, and the output was then compared with the expected results to verify the translation process's accuracy.

B. Results

Although the custom language is smaller in scale compared to full-fledged programming languages, the testing phase yielded promising results, reaffirming the toolset's effectiveness in translating code into assembly language.

Both the lexer and parser exhibited resilience and accuracy across diverse code constructs, successfully identifying and reporting errors.

The compiler effectively translated the Abstract Syntax Tree (AST) into assembly language instructions, producing expected results when executed on the target virtual machine. While these outcomes highlight the successful translation process, testing also identified areas for further enhancements. Certain complex programs revealed discrepancies in the execution of the assembly code, indicating potential improvement areas in the compiler's design.

In summary, the testing phase confirmed the efficacy of the lexer, parser, and compiler in translating the custom language into assembly language. Despite the smaller scale of the project, the results provide a solid foundation for potential future expansions and refinements to the toolset.

V. Discussion

A. Interpretation of Results

The results from the testing phase provided several insights into the functionality and performance of the lexer, parser, and compiler. Notably, the toolset demonstrated promising capabilities in translating the custom language into assembly language, despite operating on a smaller scale compared to full-fledged programming languages. This could be indicative of Python's robustness in building language processing tools and its potential in the development of domain-specific languages.

Compared to existing research or applications, this project reinforces the utility of using high-level languages like Python to construct language processing tools, especially for smaller, custom languages. The efficiency and readability of Python code played a key role in the successful implementation and testing of the toolset.

B. Advantages and Disadvantages

One significant advantage of this system is its specificity to the custom language, enabling seamless translation and execution without the need for extensive modifications or adjustments typically associated with standard compilers.

However, this specificity poses a challenge as the toolset's efficiency is limited to the custom language, lacking the universality of traditional compilers. Adapting it to other languages or more complex programming constructs will require significant future development efforts.

C. Potential Applications

Despite being developed on a smaller scale, the toolset has potential applications in educational and research settings. It can be used as a practical teaching tool for students learning about language processing, compilers, and assembly language. It also opens avenues for further research into language design, potentially sparking the development of other custom languages with unique features.

VI. Conclusion and Future Work

A. Testing Procedures

This research aimed to explore the design and implementation of a custom programming language and its translation toolset, which includes a lexer, parser, and code generator. The language processing pipeline was developed using Python's robust capabilities to enable the translation of the custom language into assembly language. Through rigorous testing, the system's effectiveness and reliability were verified. Despite operating on a smaller scale compared to full-fledged programming languages, the encouraging results established a strong foundation for future enhancements.

B. Implications

The successful implementation of this project contributes to our understanding of language processing tools and their development. It shows the potential for creating custom, domain-specific languages, and emphasizes Python's utility in this domain. Moreover, the upshot from this research might provide insights for future efforts in the design and implementation of programming languages and their respective compilers.

C. Testing Procedures

A thorough testing process is indispensable in the development of any language processing tool, and this project was no exception. The objective of this testing phase was to authenticate the functionality of the toolset and identify potential areas for improvements.

The compiler, which translates the AST into assembly language instructions, underwent a similarly exhaustive testing procedure. The generated assembly code was executed on the target virtual machine, and the output was then compared with the expected results to verify the translation process's accuracy.

Acknowledgment

I personally would like to extend my profound appreciation to my mentor, Dr. Art Hanna, whose specialized knowledge in code translators was invaluable throughout the course of this project. His expert guidance, constant support, and insightful feedback were instrumental in shaping this research and the development of the toolset.

Special thanks to the McNair program at St. Mary's University, which provided essential resources and a supportive environment that made this research possible. Their contribution to this project was fundamental in navigating the complexities of language processing and compiler design. Lastly, I am grateful for the supportive academic environment provided by St. Mary's University and my peers. This research would not have been possible without the nurturing atmosphere, dedicated faculty, and diverse academic resources available at the institution.

References

[1] D. D. McCracken, V. Profile, E. D. Reilly, and O. M. A. Metrics, "Backus-Naur Form (BNF): Encyclopedia of computer science," DL Books, https://dl.acm.org/doi/abs/10.5555/1074100.1074155

[2] Bangare, S. L., et al. "Code Parser For Object Oriented Software Modularization" International Journal of Engineering Science and Technology, vol. 2, no. 12, 2010, pp. 7262-7265

[3] Cooper, K. D., & Torczon, L. (2010). Introduction to Parsing Comp 412 [PowerPoint slides]. Rice University. Retrieved from https://homepage.cs.uri.edu/faculty/hamel/courses/2013/spring2013/csc502/presentations/parsing-presentation.pdf

[4] Fodor, P. 2022. Programming Language Syntax [PowerPoint slides]. Stony Brook University. Retrieved from https://www3.cs.stonybrook.edu/~pfodor/courses/CSE260/_L02_Syntax.pdf

[5] A. W. Appel and M. Ginsburg, *Modern Compiler Implementation in C*. Cambridge: Cambridge Univ. Press, 2010.

[6] E. J. Berk, "JLex: A lexical analyzer generator for Java (TM)," Princeton University, https://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html#SECTION1 (accessed Jul. 4, 2023).

[7] A. Sharma, F. Khan, D. Sharma, and S. Gupta, "Python: The Programming Language of Future," IJIRT, vol. 6, no. 12, May 2020, ISSN: 2349-6002

[8] P. Stanley-Marbell, "Sal/Svm: an assembly language and virtual machine for computing with non-enumerated sets," in VMIL '10: Virtual Machines and Intermediate Languages, October 2010, Article No.: 1, pp. 1-10, doi: 10.1145/1941054.1941055, published on 17 October 2010.