Journal of Science and Technology
ISSN: 2362-1443 (Print) 3059-9490 (Online)
2024, Vol. 4, No. 2 (Special Issue), 65-70

Design and Implementation of a Minimalist Programming Language Stab using Flex, Bison and LLVM

Abhilekh Gautam
Nepal College of Information Technology
Pokhara University
Abhilekh.191304@ncit.edu.np

Bishal Lamichhane
Nepal College of Information Technology
Pokhara University

Ankit Lamichhane
Nepal College of Information Technology
Pokhara University

Aashan Bhattarai
Nepal College of Information Technology
Pokhara University

Amit Shrivastava
Nepal College of Information Technology
Pokhara University
amit.shrivastava@ncit.edu.np

Article History:
Received: 19 July 2024
Revised: 8 August 204
Accepted: 17 December 2024

Keywords— ***Programming language, Flex, Bison, LLVM, Compiler***

Abstract— **We present the development of a minimalist programming language using Flex, Bison, and LLVM (Low Level Virtual Machine). The language supports essential programming constructs, including data types, loops, conditional statements, arrays, and functions. The project involved building a compiler capable of translating high-level language constructs into machine code. Challenges such as error handling and symbol table management were addressed effectively, ensuring the compiler's reliability and functionality. This project demonstrates the successful integration of Flex, Bison, and LLVM in a compiler framework, serving as a valuable educational tool. It has significant implications for Nepalese education, where practical opportunities in compiler construction are limited. By offering a concrete example of language design and implementation, this work aims to enhance the computer science curriculum in Nepalese universities, bridging the gap between theoretical concepts and practical application.**

I. INTRODUCTION

Stab is a minimalist programming language designed to teach compilers. This language uses Flex for lexical analysis, Bison for syntax analysis, and LLVM for target code generation. The main focus for Stab is simplicity, offering support for basic data types, basic looping constructs, if-else statements, arrays, and function definitions.

This paper explores the design and implementation process of Stab. We begin by elucidating the motivations behind our project and underscore the educational significance of offering students a tangible tool for applying theoretical knowledge in a practical setting. We detail the roles of Flex, Bison, and LLVM in the development pipeline, highlighting their contributions to each phase of our compiler's construction.

We also discuss the challenges encountered during the project and the solutions devised to overcome these obstacles. By documenting these experiences, we provide insights into the iterative nature of compiler design and underscore the value of hands-on learning in solidifying theoretical understanding. Practical applications are crucial in computer science education, as they bridge the gap between academic knowledge and real-world implementation [1]. Unfortunately, Nepalese universities usually lack opportunities for students to engage in research projects [2]. Opportunities for hands-on projects that explore language design and compiler construction are also often

limited. This shortfall hampers the development of practical skills crucial for understanding and contributing to advancements in software development. This project aims to equip students at Nepalese universities with essential skills in language design and compiler construction.

II. Literature Review

A compiler is a program that can read a program in one language – the source language translates it into an equivalent program in another language – the target language [3]. Several existing projects and languages have influenced the development of minimalist and educational programming languages, which are relevant to the approach presented in this paper.

One notable example is COOL, the Classroom Object-Oriented Language, developed at Stanford University specifically for undergraduate compiler course projects [4]. COOL is an object-oriented language that supports features such as inheritance [4]. It includes unique constructs like ending loop statements with ”pool” and concluding conditional statements with ”fi” [4], giving it a distinctive syntax for educational purposes.

Another significant project is ChocoPy, a programming language designed for teaching undergraduate courses on programming languages and compilers. ChocoPy, currently used at UC Berkeley, is a restricted subset of Python 3.6, with static type annotations enforcing compile-time type safety [5]. This limits learners to Python-specific feature.In addition, Kaleidoscope, a simple toy language created by the LLVM Clang team is also a source of inspiration. The development of Kaleidoscope is thoroughly documented in the tutorial series ”My First Language Frontend with LLVM Tutorial,” which provides a step-by-step guide to building a language frontend using LLVM [6]. Kaleidoscope employs a hand-written lexer and parser, and the tutorial focuses on introducing compiler construction techniques without enforcing best practices in software engineering. Despite this, it serves as an excellent resource for learning about LLVM and compiler design.

The minimalist language discussed in this paper draws inspiration from these projects but follows a distinct approach. Unlike COOL, which emphasizes object-oriented program- ming principles, and Kaleidoscope, which employs a more manual construction of the lexer and parser, the language presented here prioritizes simplicity and modularity, utilizing tools such as Flex for lexical analysis and Bison for parser generation. Additionally, it leverages the LLVM compiler infrastructure for efficient code generation, aligning with modern compiler development practices. While ChocoPy is constrained as a Python subset, Stab seeks broader flexibility, avoiding Python-specific limitations.

A. Syntactic and Semantic Design of Stab

Stab is designed as a minimalist programming language to facilitate the learning of compiler design and program-ming language implementation. The language’s syntax and semantics are intentionally simplified to support educational purposes effectively.

1) Variable Declaration

Stab requires the data type of a variable to be explicit. So, the variable declaration follows the pattern:

```
data_type variable_name;
```

Variable names must start with an alphabet and cannot have any special symbols other than underscore. Declaring a variable adds the variable to the symbol table of the current scope. Such variables have a garbage value.

2) Control Structures

Control Structures specify the sequencing and interpretation rules for a program or a part of a program [7].

While Loop:

– Syntax: The while loop iterates based on a condition. To create a while loop that runs while the value of a variable x is less than 5, we write:

```
while x < 5 {
// loop body
}
```

Here, x < 5 is the provided condition, in Stab it must be an expression that yields an integer value. – Semantics: Executes the block of statements repeatedly as long as the condition evaluates to true.

Infinite Loop (loop):

Syntax: The loop construct provides an infinite looping mechanism. The following block of code creates an infinite loop in Stab.

```
loop {
// loop body
}
```

– Semantics: Infinitely executes the enclosed block of statements.

For Loop:

Syntax: The for loop iterates over a range of values. To iterate through 1 to 500 in Stab using a for loop, we use the following block of code:

```
for i in 1 to 500 {
// loop body
}
```

Here i is an iteration variable that is added within the scope of the loop.

Semantics: Iterates over the specified range, executing the block of statements for each iteration.

If Statement:

Syntax: The if statement executes based on a condition. An if statement in Stab is written as:

```
if x < 5 {
// if body
}
```

Semantics: Executes the block of statements enclosed if the condition evaluates to true.

If Else Statement:

Syntax: An if else statement in Stab is written as:

```
if x < 5 {
// if body
}
else {
// else body
}
```

Semantics: Only executes one of the two blocks of statement. else block is executed only if the if condition is false. Whenever the if condition satisfies the if block is executed.

3) Functions

Stab supports the definition of functions with parameters and return types.

Syntax: Functions are defined using the fn keyword, followed by the function name. In Stab, functions can take any number of arguments and return either an integer or void. Every Stab program must have a main function which can be defined as:

```
fn main () {
// function body
}
```

In Stab, providing no return type for a function means the function returns void by default. So the above block of code is equivalent to:

```
fn main () -> void {
// function body
}
```

Here, -> void denotes the return type of the function as a void.

Semantics: Defines a reusable block of code that performs specific tasks based on provided parameters and returns a value of the specified type.

4) Input/Output Operations

Stab supports basic input and output operations through built-in functions it provides.

Syntax: Input is read using the input input function and output is printed using print and println functions. To print the text ”Hello World” to the console, we can use any of the following:

```
print("Hello World"); println("Hello, World");
```

The main difference between these two functions is the implicit addition of the newline character by the println function. Similarly to read a value into some variable x from the user, we can use the input function.

```
// declare x int x; input(x);
```

Semantics: Facilitates interaction with the user by capturing input and displaying output.

III. Implementation Detail

A. Overview

The implementation of Stab consists of three main phases – lexical analysis, syntax analysis, and code generation. These phases utilize Flex, Bison, and LLVM, respectively, which facilitate the transformation of high-level source code into machine code.

B. Lexical Analysis

Lexical Analysis transforms the string of characters in a source program into a stream of tokens, where the token is a string with a designated and identified meaning [9]. Lexical analysis involves using a lexer or scanner to scan the source code and break it into pre-defined tokens [10].

1) Lexer: Using flex, we develop a lexer that defines patterns for various tokens, such as keywords, operators, identifiers, and literals. These tokens are then subsequently forwarded to the parser for syntactic analysis.

2) Example: The token for an identifier is identified using a regular expression pattern `[a-zA-Z][a-zA-Z_0-9]*` in the Flex file.

C. Syntactic Analysis

Syntactic analysis recognizes the hierarchical structure of the program [10]. Syntax analysis is carried out using Bison, which constructs a parse tree based on the grammar rules defined for Stab.

3) Parser: The component used to recognize the syntactic structure of the program is called the parser [10]. The parser uses the tokens provided by the lexer to build a parse tree. This tree represents the syntactic structure of the source code.

4) Grammar Rules: The grammar for Stab is defined using Bison as a context free grammar, specifying how different constructs like loops, conditionals, and function declarations are structured.

5) Example: The rule for a while loop specifies that it consists of the while keyword, a condition, and a block of statements. i.e., while expr lcurly stmts rcurly

Once the syntax is verified, details – such as commas, semi-colons, keywords like fn for function definitions, and other syntactic elements—become redundant for later phases. The parser then generates an abstract syntax tree(AST). The AST retains the meaning of the input source code, omitting unnecessary detail. It only contains the necessary information required for code generation and is formed by removing unnecessary information from the parse tree in general [11].

D. Semantic Analysis

Semantic Analysis is a process that analyzes the validity of a meaning created by combining a program's different constituents [8]. In Stab, Semantic analysis ensures that the source program adheres to the language's semantic rules, such as type checking and scope resolution.

6) Type Checking: Ensures that operations are performed on compatible types. Since Stab currently supports only the int type, this step verifies that all expressions evaluate to integers. The Stab compiler totally depends on the LLVM Infrastructure for the guarantees required by a type system.

7) Scope Management: Scope refers to a portion of a program that is the scope of one or more declarations [3]. It is necessary to maintain and verify the scope of variables to ensure they are declared and used correctly [3]. Scopes in Stab are implemented as a linked list. The Stab compiler maintains a symbol table for every scope. A Symbol table is a data structure used by a compiler to keep track of the semantic of variables and identifiers [12]. In Stab, a new scope is created for the following constructs in the language:

a. Function Definition

b. Looping construct - `for`, `while` and `loop`

c. Conditional Statement - `if`, `if-else`

In Stab, the symbol table is implemented as a hashmap with the variable name as the key and its corresponding LLVM's AllocaInst* as the value. An AllocaInst represents an instruction to allocate memory in the stack [13]. Every scope has a pointer to its parent scope, forming a hierarchical structure.

E. Intermediate Representation and Code Generation

In Stab, LLVM is used for the intermediate representation (IR) and final code generation, transforming the high-level constructs into machine code. LLVM is a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link- time, run-time, and in idle time between runs [14].

8) Intermediate Representation (IR): The abstract syntax tree generated by the parser is converted into LLVM IR, a low-level representation suitable for optimization and code generation [15].

Optimization: LLVM performs various optimizations on the IR to enhance performance.

9) Code Generation: The optimized IR is translated into machine code for execution on the target architecture.

F. Error Handling

An important role of the compiler is to report any errors in the source program that it detects during the translation process [3]. Error handling mechanisms are incorporated at each stage to ensure reliable compilation.

10)Detecting Syntax Errors: The parser processes the sequence of tokens passed by the lexer, checking them against the defined grammar rules. If the token sequence doesn't match any of the specified grammar, it is a syntax error. Upon detecting a syntax error, the compiler invokes the error reporting function with the error location and an appropriate error message. The function then writes the error message into the standard output with other information like line number and position.

11)Detecting Semantic Errors: Semantic errors are identified during intermediate code generation, ensuring that all operations and declarations comply with the language rules. Before generating intermediate code for a statement, the compiler checks the data types of variables and function arguments for compatibility. It also verifies that the correct number of parameters is passed in function calls. Like syntax error detection, when a semantic error is detected, the compiler invokes the error reporting function that writes the error message into the standard output.

G. Implementing the print functions

Stab provides two functions - println and print to write to the console. However, both of these functions depend on libc and internally calls the C library's printf function.

The statement

println("The value of x is {}",x);

internally invokes the printf function with arguments,

printf("The value of x is %d\n", x);

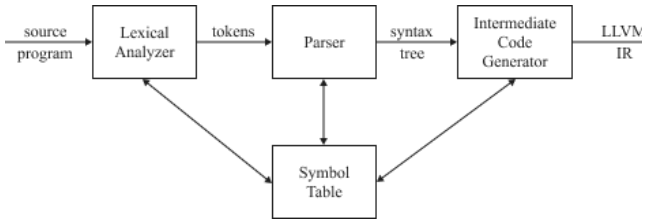

Fig. 1. Interaction of Various Components in Stab Compiler

H. Implementing the input function

Stab provides a function - `input` to read from the standard input. Similar to the printing function, this function also depends on libc internally calls the C library's `scanf` function.

So the statement

```
input(x);
```

internally invokes the `scanf` function with arguments,

```
scanf("%d", &x);
```

I. Compiler Workflow

The Stab compiler takes a program file as its input. The first thing it does after getting the input file is invokes a function, that,

a. Initializes the LLVM Context.

b. Initializes the LLVM Module.

c. Initializes the LLVM Builder Object.

After the initialization is complete, the program file is passed to the lexer. The lexer performs the following operations:

a. Breaks the entire program into the sequence of valid tokens.

b. Maintains the information about the token's location and position in the program file.

The lexer then passes the following information to the parser:

a. The type of the token.

b. The token itself.

The parser based on the information provided by the lexer, performs the following operations:

a. Reports syntactical error in the program file.

b. Generates the abstract syntax tree (AST).

Thus generated AST is used to generate the LLVM IR (Intermediate representation). Semantic checks take place during the generation of the IR. If errors are detected, they are reported back. Once the IR is generated, the compiler then starts gathering the target machine information like CPU Architecture. Once all the information is gathered, the compiler invokes LLVM's function that generates an object file. Once the object file is generated, it is linked against required libraries using the Clang compiler to produce an executable.

IV. Significance of Stab compiler

The development and implementation of Stab offer significant educational benefits, particularly in the context of computer science education. This section highlights the key areas where Stab contributes to the learning experience of students.

A. Practical Understanding of Compiler Theory

One of the primary educational values of Stab is that it provides learners with hands-on experience in compiler construction. By working through the various stages of implementing a compiler—from lexical analysis to code generation—students gain a deeper and more practical understanding of compiler theory. This practical approach demystifies complex concepts and bridges the gap between theoretical knowledge and real- world application.

B. Simplified Learning Curve

Stab's minimalist design, focusing solely on the int type and basic control structures, lowers the initial learning curve for learners. This simplicity allows learners to concentrate on core concepts without being overwhelmed by the intricacies of more advanced language features. By mastering the basics, students can build a strong foundation before moving on to more complex topics.

C. Enhanced Problem-Solving Skills

Engaging with the implementation of Stab encourages learners to develop and enhance their problem-solving skills. As they encounter various challenges during the design and coding processes, they learn to apply logical reasoning, debugging techniques, and optimization strategies. These skills are not only crucial for compiler development but are also broadly applicable across different areas of computer science and software engineering.

D. Exposure to Industry-Standard Tools

By using LLVM in the construction of Stab, learners can gain valuable exposure to industry-standard tools and technologies. Familiarity with these tools equips learners with practical skills that are highly relevant in both academic research and the software development industry. Understanding how to leverage such tools effectively prepares students for future projects and professional opportunities.

E. Contribution to Local Educational Ecosystem

In regions where opportunities to engage in real-world language design projects are limited, Stab serves as a valuable educational resource. By providing a concrete example of language implementation, it helps fill a gap in the local educational ecosystem. This contribution supports the development of a more robust and practical computer science curriculum, enhancing the overall quality of education for students.

V. Conclusion

The development of Stab, a minimalist programming language, provides a practical educational tool for students to understand the intricacies of compiler design and language implementation and a programming language to teach compilers for the teachers. Through the construction of Stab, students are exposed to key concepts such as lexical analysis, syntax parsing, semantic analysis, and code generation using industry-standard tools like LLVM. This hands-on experience bridges the gap between theoretical knowledge and practical application, making abstract concepts more tangible and comprehensible.

The challenges faced during the development of Stab, including defining a clear grammar, ensuring semantic correctness, generating efficient LLVM IR, and providing meaningful error message, were addressed through iterative refinement, comprehensive testing, and leveraging existing documentation and examples. The minimalist design of Stab focuses on core programming constructs, ensures that students can grasp the essential aspects of language design without being overwhelmed by complexity.

By integrating Stab into the curriculum, educational institutions, particularly in regions with limited resources, can significantly enhance the learning experience of their computer science students. Stab not only equips students with practical skills in compiler construction but also fosters innovative thinking and problem-solving abilities that are crucial for their future careers.

Overall, Stab serves as a valuable educational resource, demonstrating that even a minimalist language can provide profound learning opportunities. The project underscores the importance of practical, hands-on learning in computer science education and offers a foundation upon which more advanced language features and concepts can be built.

References

[1] Fantinelli, Stefania & Cortini, Michela & Di Fiore, Teresa & Iervese, Stefano & Galanti, Teresa. (2024). “Bridging the Gap between Theoretical Learning and Practical Application: A Qualitative Study in the Italian Educational Context. Education Sciences.“ 14. 198. 10.3390/educsci14020198.

[2] S. Shakya and D. Rauniar, ”Information Technology Education in Nepal: An Inner Perspective,” *Electronic Journal on Information Systems in Developing Countries*, vol. 8, 2002, doi: 10.1002/j.1681-4835.2002.tb00049.x.

[3] A. V. Aho, Monica S. Lam, R. Sethi, J.D. Ullman “Compilers: Principles, Techniques, and Tools“, 2nd ed. Addison-Wesley, 2007.

[4] A. Aiken, “The COOL Language”, Stanford University, https://theory.stanford.edu/~aiken/software/cool/cool.html, Accessed: May 10, 2024.

[5] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. ChocoPy: A programming language for compilers courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, pages 41–45, Athens, Greece, 2019. Association for Computing Machinery, New York, NY, USA.

[6] LLVM Project, “My First Languge Front end with LLVM Tutorial“, LLVM, https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html, Accessed: Jul. 2, 2024

[7] Fisher, David Allen. *Control structures for programming languages*. Carnegie Mellon University, 1970.

[8] Son, Y., Lee, Y. (2011). ”The Semantic Analysis Using Tree Transformation on the Objective-C Compiler.” In: Kim, Th., et al. Multimedia, Computer Graphics and Broadcasting. MulGraB 2011. Communications in Computer and Information Science, vol 262. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-27204-2 8

[9] Pai T, Vaikunta and Aithal, P. S. 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, 2020.

[10] Wilhelm, Reinhard, Seidl, Helmut, and Hack, Sebastian. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013.

[11] Kim, Jaehyun and Lee, Yangsun. A study on abstract syntax tree for development of a JavaScript compiler. *International Journal of Grid and Distributed Computing*, 11(6):37–48, 2018.

[12] Matta, Jad. Paper on symbol table implementation in compiler design. December 2019. Available online at: https://doi.org/10.13140/RG.2.2. 14217.60005.

[13] LLVM Documentation. `AllocaInst` class documentation. Available online at: https://llvm.org/doxygen/classllvm 1 1AllocaInst.html. Accessed: June 30, 2024.

[14] Lattner, C. and Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75-86, 2004. doi: https://doi.org/10.1109/CGO.2004. 128166510.1109/CGO.2004.1281665.

[15] LLVM Documentation. LLVM Language Reference Manual. Available online at: https://llvm.org/docs/LangRef.html. Accessed: June 30, 2024.