Title Slide

Good morning, everyone! My name is Aayushya Lakkadwala, and I'm here to present *BoinkLang*—a lightweight, interpreted programming language developed by my team, *Quad Binary*.

BoinkLang is designed to explore language design and interpreter implementation, offering a hands-on approach to understanding how programming languages work internally.

"Let's dive in and explore what makes BoinkLang unique!

Table of Contents

To understand our project, we will look at following points: Introduction, Problem Statement, Implementation Overview, Features, Challenges Faced, Results & Future Work, Conclusion.

Introduction Slide

Before we delve into the technical details, let me introduce you to *BoinkLang*. It is a lightweight, interpreted programming language developed in Go, designed to explore interpreter design and language processing.

The primary motivation behind BoinkLang is to provide a simple yet effective environment for learning key concepts like tokenization, parsing, and evaluation. By focusing on core programming constructs such as functions and control flow, BoinkLang bridges the gap between theoretical compiler design and practical implementation.

Moreover, this project was created as a hands-on experiment to better understand how interpreters work and how code is processed internally. Through this, we aim to offer aspiring language designers a structured way to explore compiler fundamentals while also deepening expertise in Go programming.

Now, let's take a closer look at the problem we aimed to solve with BoinkLang.

Problem Statement Slide

In modern programming, most languages abstract away their internal processing to provide a seamless development experience. While this is beneficial for productivity, it often makes it difficult to understand how code execution works under the hood.

Key processes like tokenization, parsing, and execution remain hidden, making it challenging for developers—especially beginners—to grasp how interpreters process and evaluate code.

Additionally, most interpreters are designed with a strong focus on efficiency, which often results in complex implementations that are difficult to modify or experiment with.

This is where *BoinkLang* comes in. It offers a lightweight and modular design, breaking down language processing into clear, structured steps. By doing so, it allows users to explore, modify, and experiment with interpreters without unnecessary complexity.

Now, let's move on to how BoinkLang is implemented and how it processes code.

Implementation Overview Slide

Now that we understand the problem BoinkLang aims to solve, let's look at its implementation.

BoinkLang is built using *Go*, a language known for its simplicity, strong standard library, and efficient performance. One of the key advantages of using Go is its built-in garbage collection, which handles memory management, allowing us to focus on the core logic of the interpreter. Additionally, Go's static typing helps catch errors early, making development more robust and efficient.

BoinkLang follows a structured execution process consisting of four main components:

1. **Tokenizer (Lexer):** Converts the source code into tokens, breaking it down into meaningful symbols.
2. **Parser:** Takes these tokens and constructs an *Abstract Syntax Tree* (AST) to represent the program's structure.
3. **Evaluator:** Interprets the AST and executes the corresponding operations.
4. **Symbol Table:** Manages variable bindings, ensuring proper scope handling and execution consistency.

BoinkLang also features an interactive REPL (Read-Eval-Print Loop) mode, which allows users to test and execute code in real-time, making it an excellent tool for learning and experimentation.

Now that we understand how BoinkLang processes code, let's move on to its core features.

Features Slide

BoinkLang is designed to be simple yet powerful, supporting essential programming constructs that make it both accessible for beginners and useful for experienced developers experimenting with language design.

Some of its key features include:

1. **Variables:** Users can define and store values, enabling data manipulation.
2. **Control Flow:** Supports conditional statements like if-else, allowing decision-making in code execution.
3. **Functions:** Enables modular programming by allowing users to define reusable blocks of code.

4. **Loops (Planned):** Future updates will include loop constructs like for and while to enhance control flow capabilities.

Another crucial aspect of BoinkLang is its **custom error-handling system**. Unlike generic error messages, BoinkLang provides clear, detailed error descriptions, helping users quickly identify and resolve issues in their code. This enhances the debugging process, making it more intuitive and user-friendly.

With these features, BoinkLang serves as a great learning tool for understanding interpreter design. However, developing these functionalities came with its own set of challenges, which I'll discuss next.

Challenges Faced Slide

Building BoinkLang was both an exciting and challenging experience. While implementing its core functionalities, we encountered several technical hurdles that required careful consideration and problem-solving.

Some of the major challenges we faced included:"

1. **Parsing Complexity:** One of the biggest challenges was handling **operator precedence and associativity** correctly. Ensuring that expressions were evaluated in the right order while keeping the parser simple required extensive testing and refinement. Balancing accuracy with simplicity was a key hurdle in this process.
2. **Memory and Variable Management:** Managing memory and variable bindings was another challenge, especially when dealing with **nested function calls and variable scopes**. Ensuring that variables were referenced correctly across different scopes while maintaining efficient memory usage required careful planning and optimization.
3. **Error Handling and Debugging:** Creating a robust error-handling system that provided **clear and informative error messages** was essential. Unlike traditional languages that often return cryptic errors, we wanted BoinkLang to guide users through debugging in a more intuitive way.

Overcoming these challenges allowed us to refine BoinkLang into a more stable and user-friendly interpreter. Now, let's look at what we have achieved so far and what's next for BoinkLang.

Results & Future Work Slide

Results: We have successfully achieved our core goals. We've developed a lightweight interpreter that demonstrates key concepts of language processing, such as tokenization, parsing, and evaluation. Our implementation, written in Go, has proven to be efficient for handling basic syntax and parsing tasks. The interpreter's design allows for easy extensions, meaning that the language can grow to include additional features while maintaining its simplicity.

Future Work: Moving forward, we have exciting plans to further enhance BoinkLang. Our key priorities include expanding the language's functionality with more complex data structures and improving its error handling mechanisms for a smoother user experience. We also aim to optimize the performance of the interpreter to handle more demanding use cases. Additionally, we plan to introduce features such as loops, switch cases, and increment/decrement operators, adding more flexibility and control to the language.

Conclusion Slide

Concluding this presentation, we can say that we have successfully achieved our core goals. The development of a lightweight interpreter in Go has proven to be efficient, handling basic syntax, parsing, and evaluation tasks. With this strong foundation, BoinkLang is well-positioned for future growth and enhancements.

Looking ahead, we plan to introduce features such as loops, switch cases, and increment/decrement operators to provide users with more flexibility and control in their programming. Additionally, optimizing performance and expanding the language’s functionality will remain key priorities.

In conclusion, with significant progress already made, we are excited about the future of BoinkLang. We look forward to refining and expanding the language, ensuring it remains a useful tool for those exploring programming language design.

Thank You for Your Precious Time and Attention