You're making great progress! Now that you've laid out a solid foundation for your grammar and addressed the main syntax and logical constructs, the next stages in the process of building a programming language typically involve the following:

1. **Lexical Analysis (Lexer/Scanner)**
   - **Purpose**: The lexer's role is to break the input code into tokens, which are the smallest units of meaning in the language (e.g., keywords, identifiers, literals, operators, etc.).
   - **What to do**:
     - Write a **lexer** (typically using a tool like **Flex** or **ANTLR**). This lexer should match the tokens in your grammar (e.g., `var`, `const`, operators like `+`, `&&`, `if`, etc.).
     - The lexer will output a stream of tokens that the parser can use. Tokens represent different components like keywords, variable names, operators, and literals.
     - Handle whitespace and comments by ignoring them during tokenization.

   - **Next step**: After implementing the lexer, test it by feeding it various source code examples to make sure it's correctly identifying tokens.

2. **Parsing (Syntax Analysis)**
   - **Purpose**: The parser checks if the sequence of tokens from the lexer matches the rules of your grammar.
   - **What to do**:
     - You've already written the **grammar** (in Bison, ANTLR, or another parser generator), which is excellent. The next step is to integrate this with your lexer to generate a **syntax tree** or **abstract syntax tree (AST)**.
     - The parser will build this tree to represent the structure of the input program, checking that all syntax rules are followed.

   - **Next step**: Use the lexer and parser together to build a working **syntax tree** or **abstract syntax tree** (AST) from valid programs. This is a critical structure for further processing.

3. **Semantic Analysis**
   - **Purpose**: After parsing, the program should undergo **semantic analysis** to ensure the code is logically correct. This involves checking things like type consistency, scope, and variable declarations.
   - **What to do**:
     - **Symbol Table**: Build a **symbol table** to track variables, functions, and classes, along with their types and scopes.
     - **Type Checking**: Check that the types of operands in expressions match the expected types (e.g., ensuring you can't add a string to an integer, or call a function with the wrong number or types of arguments).
     - **Scope Checking**: Ensure that variables and functions are defined in the correct scope (e.g., checking if you use a variable before declaring it).

   - **Next step**: Start implementing **semantic checks** by performing type inference and scope resolution. This will catch bugs related to uninitialized variables, function calls with incorrect arguments, and other logic errors.

4. **Intermediate Representation (IR)**
   - **Purpose**: Translate the AST into a simpler form, often called **intermediate representation (IR)**, which is easier to manipulate and optimize before translating to the final output (e.g., machine code, bytecode, etc.).

  - **What to do**:
   - Convert the **AST** into an **intermediate representation** that will be used for optimization or code generation. For example, you can convert your AST into a set of instructions or a simpler tree format that is easier to manipulate.

  - **Next step**: This might be a more advanced step depending on how deep you want to go, but converting to an **IR** is a great intermediate step before final code generation.

5. **Optimization (Optional but Recommended)**
  - **Purpose**: Optimize the intermediate representation (IR) to make the final code faster or smaller.
  - **What to do**:
   - Implement basic optimizations like removing unused variables, constant folding (evaluating constant expressions at compile-time), or optimizing loops.
   - This step isn’t mandatory for simple interpreters or compilers, but it improves performance.

  - **Next step**: You can implement a simple optimization phase that improves the generated code (e.g., removing redundant calculations or variables).

6. **Code Generation**
  - **Purpose**: Translate the intermediate representation (IR) or AST into executable code, bytecode, or another lower-level format.
  - **What to do**:
   - Convert the **IR** or **AST** into target code, like **machine code** (for a compiler), **bytecode** (for a virtual machine), or **another language** (for transpilers).
   - Implement a **code generator** that emits the target code for your chosen platform (e.g., if you are writing a compiler, this could be x86 assembly or a virtual machine code).

  - **Next step**: Implement a basic **code generator** that outputs simple instructions or code. You might want to start by generating a higher-level language (e.g., C or Java) or bytecode first.

7. **Runtime System (If Needed)**
  - **Purpose**: Implement the runtime environment for your language (if you are targeting a virtual machine or bytecode).
  - **What to do**:
   - If you're building an **interpreter** or **virtual machine**, you need to create a runtime environment. This includes managing memory, executing bytecode, handling variables, function calls, etc.
   - For compiled languages, this step is not always needed, but for bytecode or interpreted languages, a runtime environment will be necessary.

  - **Next step**: Focus on building the **runtime** environment or an interpreter if you're working with an interpreted language or bytecode.

8. **Error Handling & Debugging Support**
  - **Purpose**: Provide helpful error messages when users write invalid code.
  - **What to do**:
   - Implement good **error reporting** during parsing, semantic analysis, and runtime. Your language should provide descriptive error messages that point to the problem in the code and explain what’s wrong.
   - Add **debugging support**, like stack traces or symbolic error reports.

   - **Next step**: Implement meaningful error messages during the parsing and execution phases, and if needed, set up **debugging tools** for your language.

9. **Testing & Refining the Language**
   - **Purpose**: Test the language with various code samples to ensure it behaves as expected.
   - **What to do**:
     - Continuously test the language by writing a large set of test cases that cover all features of the language.
     - Refine your design and implementation based on feedback and issues that arise during testing.

   - **Next step**: Start writing **unit tests** and sample programs in your language to ensure correctness. You can use **unit testing frameworks** or simple test programs to verify the expected output.

10. **Documentation and Language Features**
   - **Purpose**: Provide good documentation for your language and its features.
   - **What to do**:
     - Write a **user manual** or **language specification**. Describe the syntax, semantics, and features of the language.
     - Document how to use the compiler or interpreter, how to write programs in the language, and how to run/debug them.

   - **Next step**: Focus on writing clear documentation and providing examples of how to use the language effectively.

Summary of Next Steps:
1. **Implement the lexer** and test it to make sure it correctly tokenizes the input.
2. **Build the parser** using the grammar and generate the AST.
3. **Perform semantic analysis** to check for type errors, undeclared variables, etc.
4. Optionally, **implement an intermediate representation (IR)** for further manipulation.
5. **Generate target code** (e.g., machine code, bytecode).
6. **Build a runtime system** if your language is interpreted or has bytecode.
7. **Add error handling** and meaningful error messages.
8. **Test your language** and refine it based on the feedback from testing.
9. Write **documentation** to describe how to use the language and its features.

These steps will get you closer to a working interpreter or compiler. Each stage builds on the previous one, so try to tackle them incrementally. Would you like more details on any of these stages or help with a specific part of the process?