New Programming Language

1. Language Design

Scope: Determine the target audience, application domain (e.g., web development, systems programming), and the specific problems the language aims to solve.
Syntax Design: Decide on the syntax of your language. This includes:
Keywords: Define reserved words for control structures (if, else, while, etc.).
Operators: Define how arithmetic, logical, and comparison operators will work.
Data Types: Design the primitive types (integers, floats, strings, etc.) and how they interact.
Complex Data Structures: Design arrays, lists, dictionaries, objects, etc.
Control Structures: Define how loops, conditionals, and functions will be structured.
Design Paradigms: Choose the paradigms your language will support:
Procedural: Emphasizes procedure calls (e.g., C).
Object-Oriented: Supports objects and classes (e.g., Java).
Functional: Treats computation as the evaluation of mathematical functions (e.g., Haskell).
Logic: Based on formal logic (e.g., Prolog).

2. Lexical Analysis

Purpose: Breaks down the source code into tokens, which are the smallest units of meaning (keywords, operators, identifiers, etc.).
Tools:
Flex (Fast Lexical Analyzer Generator): For C++, it generates C++ code that performs lexical analysis based on regular expressions.
PLY (Python Lex-Yacc): For Python, it provides a way to perform lexical analysis with regular expressions in Python.
Implementation:
Define the grammar rules using regular expressions.
Create a lexer that reads the source code and produces tokens.
Handle errors like unrecognized tokens and report them to the user.

3. Parsing

Purpose: Converts the sequence of tokens into a parse tree or abstract syntax tree (AST), representing the syntactic structure of the code.
Tools:
Bison: A parser generator that works with Flex to produce a C++ parser.
PLY (Python Lex-Yacc): Also includes Yacc-like parsing capabilities for Python.
Types of Parsers:
Top-Down Parsing (Recursive Descent, LL Parsers): Simple to implement but can struggle with certain grammar types.
Bottom-Up Parsing (LR, LALR Parsers): More complex but handles a broader range of grammars.
Implementation:
Define grammar rules in Backus-Naur Form (BNF) or Extended BNF (EBNF).
Implement the parser to generate the AST from the token sequence.
Error handling: Implement strategies to recover from syntax errors.

4. Semantic Analysis

Purpose: Ensures that the syntax tree adheres to the language's rules (e.g., type checking, scope resolution).
Steps:
Type Checking: Ensure variables and expressions have the correct types.
Scope Management: Track variable/function declarations and ensure they are used correctly.
Contextual Rules: Implement rules that can't be checked syntactically, like ensuring a function returns a value of the correct type.
Error Handling: Detect and report semantic errors, such as type mismatches and undefined variables.
Implementation:
Traverse the AST to apply semantic rules.
Use symbol tables to track variable and function declarations.

5. Intermediate Representation (IR)

Purpose: An intermediate code that sits between the high-level language and machine code, making optimization easier.
Examples: Three-address code, abstract syntax trees, or bytecode.
Implementation:
Generate IR from the AST.
Perform optimizations like constant folding, dead code elimination, etc.
Target the IR for specific platforms or for a virtual machine.

6. Code Generation

Purpose: Translates the intermediate representation into machine code or bytecode.
Tools:
LLVM: A compiler infrastructure that supports C++ and can generate machine code for various architectures.
Python Bytecode: If targeting a Python VM, generate Python bytecode.
Implementation:
Implement a backend that takes the IR and generates target-specific code.
Handle different instruction sets and calling conventions.

7. Optimization

Purpose: Improve the performance of the generated code.
Techniques:
Peephole Optimization: Simplify short sequences of instructions.
Loop Optimization: Techniques like loop unrolling, invariant code motion.
Inlining: Replace function calls with the function code to reduce call overhead.
Implementation:
Analyze the IR and apply optimization passes.

Ensure that optimizations don't alter the program's behavior.

8. Runtime Environment

Purpose: Manages the execution of programs, handling tasks like memory management, exception handling, and input/output.

Components:

Garbage Collection: Automatically manages memory allocation and deallocation (optional, depending on the language).

Exception Handling: Implements mechanisms for dealing with runtime errors.

Standard Library: Provide common functionality like string manipulation, file I/O, etc.

Implementation:

Design a runtime that integrates with the generated code.

Ensure the runtime is portable and efficient.

9. Testing and Debugging

Purpose: Ensure the language behaves as expected across various scenarios.

Tools:

Unit Testing: Write tests for individual language features.

Fuzzing: Generate random inputs to find edge cases.

Debugging Tools: Implement or use existing debuggers to track down issues in the language.

Implementation:

Develop a comprehensive test suite covering syntax, semantics, and runtime behavior.

Use debugging tools to identify and fix issues during development.

10. Documentation and User Guide

Purpose: Provide users with clear and comprehensive documentation for the language.

Components:

Language Specification: Detailed documentation of the language syntax, semantics, and usage.

Tutorials: Example-driven guides to help users learn the language.

API Documentation: For any libraries or built-in functions provided by the language.

Implementation:

Write detailed documentation covering all aspects of the language.

Ensure the documentation is accessible and easy to understand.

11. Conclusion

This project will involve significant work across these areas, and each step will build upon the last. The choice between C++ and Python depends on your focus—C++ for performance and control, Python for ease of development. This plan provides a strong foundation for developing a new programming language, with each step outlined to guide you through the process.

- **C++**: Often chosen for its performance and control over system resources. It's widely used in compilers and interpreters.
- **Python**: Preferred for rapid prototyping due to its simplicity and readability. Ideal for implementing high-level features and testing ideas quickly.

12. Resources

How to Build a New Programming Language:
https://pgrandinetti.github.io/compilers/page/how-to-build-a-new-programming-language/

Nuts and bolts of Programming Languages:
https://pgrandinetti.github.io/compilers/

I wrote a programming language. Here's how you can, too:
https://www.freecodecamp.org/news/the-programming-language-pipeline-91d3f449c919/