Step3.1 – Formal Grammar

Creating a complete programming language with a formal grammar, lexer, parser, semantic analysis, and code generation is a significant task that requires substantial effort and expertise. As an AI language model, I can provide you with an outline and general concepts for each component. However, implementing a fully functional language involves a substantial amount of work and code that goes beyond the scope of a single response. Here's an overview of each component:

1. Formal Grammar: Formal grammar defines the syntax and structure of the language using Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF). It specifies the rules for how different language constructs can be combined to form valid programs. Here's a simplified example of a formal grammar for "Pear":

program   ::= (statement | function_definition | class_definition)*

statement   ::= assignment_statement | if_statement | while_statement | for_statement | ...

assignment_statement ::= "var" identifier ":" type "=" expression

if_statement   ::= "if" expression ":" block ("elif" expression ":" block)* ("else" ":" block)?

while_statement ::= "while" expression ":" block

for_statement ::= "for" identifier "in" expression ":" block

...

expression   ::= logical_expression

logical_expression ::= comparison_expression (("and" | "or") comparison_expression)*

comparison_expression ::= additive_expression (("==" | "!=" | ">" | ">=" | "<" | "<=") additive_expression)*

additive_expression ::= multiplicative_expression (("+" | "-") multiplicative_expression)*

...

type   ::= "int" | "float" | "str" | "bool" | identifier

block   ::= INDENT statement+ DEDENT

2. Lexer: The lexer is responsible for tokenizing the source code into a sequence of tokens based on the formal grammar rules. Each token represents a fundamental element of the language, such as keywords, identifiers, operators, and literals. Here's an example of some tokens for "Pear":

INT_LITERAL   ::= [0-9]+

FLOAT_LITERAL ::= [0-9]+ "." [0-9]+

STRING_LITERAL ::= "\"" .*? "\""

IDENTIFIER   ::= [a-zA-Z_][a-zA-Z0-9_]*

...

3. Parser: The parser takes the stream of tokens generated by the lexer and builds an abstract syntax tree (AST) based on the formal grammar rules. The AST represents the hierarchical structure of the program. For instance, the "if" statement in the source code would be represented in the AST as a node with children representing the condition and the blocks for the "if," "elif," and "else" clauses.

4. Semantic Analysis: Semantic analysis involves checking the meaning and validity of the program based on its context. It includes type checking, scope analysis, and resolving variable references. For example, in "Pear," the semantic analyser ensures that variables are declared before they are used and checks for type compatibility in expressions.

5. Code Generation: Code generation converts the AST into machine code or intermediate representation (IR) that can be executed by the target platform. This step involves transforming the high-level language constructs into low-level operations.

Please note that building a complete language requires a deep understanding of compiler theory and language design. It's a complex and time-consuming task that often involves multiple iterations and refinements.

To start implementing these components, you can use tools like ANTLR, Bison, or PLY (Python Lex-Yacc) to generate the lexer and parser based on the formal grammar. Additionally, we'll need to create the necessary data structures and algorithms for the semantic analysis and code generation stages.