1. To handle variable shadowing across nested blocks, the interpreter must maintain scopes. Each block introduces a new scope, and variables declared in an inner block can shadow variables with the same name in an outer block. When resolving a variable, the interpreter should first check the innermost (local) scope and then move outward to the enclosing (parent) scopes.
2. This interpreter supports the following types of expressions:
   1. Integer Literals:
   2. Direct integer values, such as 10, 42, or -5.
   3. Variable References:
   4. Variables that have been previously defined in the environment.
   5. Binary Operations:
   6. Arithmetic operations involving two operands:
   7. Subtraction (-): Subtracts the second value from the first.
   8. Multiplication (\*): Multiplies two values.
   9. Division (/): Divides the first value by the second.
   10. Nested Expressions:
   11. Supports nested binary operations.
   12. Print operation
3. To add support for if statements and while loops in the interpreter, you need to extend the eval\_expr method and add new methods to handle these constructs. Additionally, you need to modify the parser to generate appropriate AST nodes for if and while statements.
   1. Steps to Add Support for if Statements and while Loops:
      1. Extend the AST:
   2. Add new classes for If and While nodes in ast\_nodes.py.
   3. Update the Parser:
      1. Extend the parser to recognize if and while statements and generate the corresponding AST nodes.
   4. Extend the Interpreter:
      1. Add methods to evaluate If and While nodes.
4. Changing the interpreter from dynamically typed to statically typed would significantly affect its design. Here are the key changes and considerations:
   1. Type Checking:
   2. Dynamic Typing: Types are determined at runtime, and type errors are only caught when the program is executed.
   3. Static Typing: Types are determined at compile-time (or during parsing), and type errors must be caught before execution.
   4. Impact:
      1. The interpreter would need to perform type checking during parsing or in a separate type-checking phase before execution.
      2. Each variable declaration would need to specify its type explicitly (e.g., int x = 10;), and the interpreter would need to ensure that assignments and expressions conform to the declared types.
5. Recursive descent parsing is a great choice for simple grammars and when customizability is important, but it may not be suitable for more complex or performance-critical applications.