# Question # 2

The mini compiler translates a high-level programming language into a lower-level representation by performing multiple phases of analysis and transformation. This documentation outlines two of the core functionalities of the compiler: Parser (Syntax Analysis) and Semantic Analysis.

**Core Functionalities**
1. Parser (Syntax Analysis)
Purpose:
The parser ensures that the source code is syntactically correct. It verifies whether the structure of the code adheres to the grammatical rules of the programming language.

**Functionality:**
The parser receives a stream of tokens from the lexer (lexical analyzer) and constructs a Syntax Tree or Abstract Syntax Tree (AST). This tree represents the hierarchical structure of the program. The parser checks whether the code follows the formal grammar of the language and reports errors for any violations.

**Process:**
Input: Token stream from the lexer.
Output: An Abstract Syntax Tree (AST) that represents the syntactic structure.
Error Detection: If there are any syntax errors (e.g., missing parentheses, misplaced operators), the parser detects and reports them.
Grammar Checking: The parser uses a defined grammar (often in a form like BNF or EBNF) to check whether the code follows the correct language rules.
**Example:**
For the input:

plaintext
Copy code
```
int x = 10 + (5 * 2);
```
The parser might generate an AST like:

yaml
Copy code
```
Assignment
  ├── Type: int
  ├── Variable: x
  └── Expression
      ├── Addition
      ├── Operand: 10
      └── Operand: Multiplication
          ├── Operand: 5
          └── Operand: 2
```

If the input had been malformed, such as int x = 10 + * 5;, the parser would flag it as a syntax error.

**Error Handling:**
Unmatched parentheses: e.g., int x = (10 + 5;
Missing operators: e.g., int x = 10 5;
Incorrect statement structure: e.g., int = 10;
2. Semantic Analysis
Purpose:
The semantic analysis phase ensures that the program has logical correctness in terms of variable declarations, type compatibility, function calls, and other language rules.

**Functionality:**
Semantic analysis operates after the syntax analysis. It validates the meaning and logic of the program. It checks for consistency, such as:

Correct type usage (e.g., a string cannot be assigned to an integer variable).
Scope resolution (e.g., a variable must be declared before use).
Function signature validation (e.g., the number and types of function arguments must match the function declaration).
Variable initialization checks (e.g., variables must be initialized before they are used).
Process:
Input: Abstract Syntax Tree (AST) from the parser.
Output: Annotated AST or error messages for semantic violations.
Error Detection: Detects issues such as type mismatches, uninitialized variables, and improper scope usage.
Symbol Table: A symbol table is typically used to store information about variables, functions, and types. This helps in performing checks like type checking and scope resolution.
Example:
For the input:

plaintext
Copy code
int x = 10;
x = x + "Hello";
The semantic analyzer would flag an error because "Hello" is a string, which cannot be added to an integer variable x. The error message might be:

csharp
Copy code
**Semantic Error:** Type mismatch. Cannot add 'int' and 'string'.
**Error Handling:**
Type Mismatch: Trying to assign a string to an integer variable.
Undeclared Variables: Using a variable without declaring it first.

Function Signature Mismatch: Calling a function with an incorrect number or type of arguments.
Uninitialized Variables: Attempting to use a variable that has not been initialized.
Integration of Parser and Semantic Analysis
These two phases—parsing and semantic analysis—work together to ensure that the code is both syntactically and semantically correct before further stages of compilation like optimization and code generation.

**Parsing:** Ensures the structure of the code is correct.
Semantic Analysis: Ensures the meaning and logical consistency of the code.
Once these phases are completed successfully, the compiler can proceed to the later stages, such as intermediate code generation, optimization, and final code generation.

**Conclusion**
The Parser and Semantic Analysis are critical components of a mini compiler. The parser ensures that the code follows the correct syntax, while the semantic analysis phase ensures that the code makes sense in terms of types, variables, and other logical structures. Together, they help lay the foundation for the subsequent stages of the compilation process.