**Ruleset for Successful Compilation**

**1. Lexical Rules (Token Structure)**

The source code must conform to the following token definitions to be lexically valid:

* **Identifiers:**
  + Identifiers must start with a letter (A-Z, a-z) and can be followed by letters, digits (0-9), or underscores (\_).
  + Examples of valid identifiers: x, y, my\_var, result2.
  + Example of invalid identifiers: 2x, @var, -value.
* **Keywords:**
  + The following keywords are reserved and cannot be used as identifiers: return.
  + Example: return is used to return a value from a function.
* **Operators:**
  + Supported operators include:
    - **Arithmetic:** + (addition), - (subtraction), \* (multiplication), / (division).
    - **Assignment:** = (used to assign values to variables).
* **Literals:**
  + **Integers:** Must consist of digits (0-9) without any decimal points or signs.
  + Examples of valid integers: 5, 10, 100.
  + Example of an invalid integer: 5.0, -3 (your compiler does not seem to support floating-point numbers or negative numbers).
* **Punctuation:**
  + **Semicolon (;)**: Each statement must be terminated by a semicolon.
  + **Parentheses ((, ))**: Used for grouping expressions (e.g., z = (x + y)).

**2. Syntactic Rules (Grammatical Structure)**

The code must follow these rules to be syntactically valid:

* **Program Structure:**
  + A valid program consists of a sequence of **statements**, each of which must end with a semicolon (;).
* **Statements:**
  + **Assignment Statement:**
    - Syntax: <identifier> = <expression>;
    - Description: The left-hand side must be a valid identifier, and the right-hand side must be a valid expression.
    - Example: x = 5;
    - Invalid: 5 = x; (The left side cannot be a constant).
  + **Return Statement:**
    - Syntax: return <expression>;
    - Description: The return keyword must be followed by a valid expression and terminated with a semicolon.
    - Example: return x + y;
    - Invalid: return; (A return statement must have an expression).
* **Expressions:**
  + **Arithmetic Expressions:**
    - An expression can involve integers, variables, and arithmetic operators.
    - Syntax: <expression> ::= <term> | <expression> + <term> | <expression> - <term>
    - Example: z = x + (y \* 2);
  + **Parentheses**: Parentheses can be used to group expressions, ensuring correct operator precedence.
    - Example: z = (x + y) \* 2;
* **Operator Precedence:**
  + Multiplication and division have **higher precedence** than addition and subtraction.
  + Parentheses can override precedence.
  + Example: x + y \* 2 is equivalent to x + (y \* 2).

**3. Semantic Rules (Meaning and Context)**

The code must follow these semantic rules to ensure that the meaning of the program is valid:

* **Variable Declaration and Initialization:**
  + Variables must be assigned a value before they are used in an expression.
  + Example:

x = 5; // Valid, x is assigned before use.

y = x + 3; // Valid, x is already assigned.

* + Invalid:

y = x + 3; // Invalid, x has not been assigned a value.

* **Type Rules:**
  + Only **integer** values and variables are supported.
  + Operations involving variables or literals that aren't integers will cause a semantic error.
  + Example: x = 5 + y; is valid if both x and y are integers.
* **Operator Compatibility:**
  + Only arithmetic operations involving integers are allowed. The operands for +, -, \*, and / must both be integers.
  + Example:

z = (x + y) \* 2; // Valid, assuming x and y are integers.

* + Invalid:

z = x / "hello"; // Invalid, cannot divide an integer by a string.

* **Return Statement:**
  + A return statement must return a valid expression.
  + The return value must be a valid integer expression or a previously assigned variable.
  + Example: return z; is valid if z has been assigned a value.

**4. Code Formatting and Structure**

* **Semicolons:** Every statement must end with a semicolon (;).
  + Example: x = 5; is valid, but x = 5 is not.
* **Whitespace:** While spaces and line breaks are ignored, it is good practice to use whitespace to improve code readability.
  + Example:

x = 5;

y = 10;

z = x + y;

**5. Error Handling Rules**

* **Lexical Errors:**
  + Using invalid characters (e.g., @, #, $) in identifiers or expressions will result in lexical errors.
  + Example of an invalid identifier: my$var
* **Syntax Errors:**
  + Syntax errors occur if statements are improperly structured or missing elements, such as semicolons or parentheses.
  + Example of a syntax error: x = (5 + 3; (missing closing parenthesis).
* **Semantic Errors:**
  + A semantic error occurs when a variable is used before being assigned, or when incompatible types are used in expressions.
  + Example of a semantic error: z = a + 5; if a is not declared or assigned before its use.

**6. Compilation Phases Overview**

* **Lexical Analysis:** Ensures that the code consists of valid tokens (identifiers, keywords, literals, operators, etc.).
* **Syntax Analysis:** Ensures that the structure of the code follows the grammatical rules (e.g., valid statements and expressions).
* **Semantic Analysis:** Ensures that variables are properly declared and used, and that expressions make logical sense (e.g., valid types and assignments).
* **Intermediate Code Generation:** Translates the syntactically and semantically valid code into an intermediate form (Three-Address Code, TAC).
* **Code Optimization:** Simplifies the TAC by removing redundant calculations or folding constant expressions.
* **Assembly Code Generation:** Converts the optimized TAC into assembly code instructions.
* **Machine Code Generation:** Translates the assembly code into machine language (binary), making it executable on the target hardware.