Purpose and Functionality

The code implements a lexer and parser for a simple intermediate language (IL) designed to support arithmetic expressions and assignment operations. Using **Flex** for tokenizing input and **Bison** for parsing and syntax analysis, the program efficiently handles variable assignments, arithmetic computations, and basic optimizations while maintaining a symbol table to track variables and their values.

Components and Features

1. Lexer (Flex Component)

- **Objective**: To tokenize the input stream into recognizable components (tokens) such as variables, constants, operators, and delimiters for the parser.
- **Key Features**:
  - **Identifiers**: Recognized using the pattern `[a-zA-Z_][a-zA-Z0-9_]*`.
  - **Numeric Constants**: Matches sequences of digits `[0-9]+`.
  - **Operators**: Detects `=`, `+`, `-`, `*`, `/`, and `^` for assignments and arithmetic expressions.
  - **Delimiters**: Recognizes `;` to indicate the end of a statement.
  - **Line Number Tracking**: Tracks and increments line numbers for debugging and error reporting.
  - **Whitespace and Errors**: Ignores unrecognized characters and whitespace.

---

2. Parser (Bison Component)

- **Objective**: To parse tokens from the lexer, evaluate expressions, execute assignments, and optimize the intermediate code through simplification and constant folding.
- **Key Features**:
  - **Symbol Table Management**:
    - Utilizes an `unordered_map<string, item>` to store and manage identifiers and constants.
    - Each `item` comprises:
      - `is_constant`: Boolean flag indicating if the variable holds a constant value.
      - `value`: Stores the numeric value or ASCII representation.
    - Handles uninitialized variables by assigning their ASCII value.
  - **Expression Evaluation**:
    - **Addition and Subtraction**:
      - Simplifies expressions like `x + 0` or `x - 0`.
      - Performs constant folding for purely numeric operations.
    - **Multiplication and Division**:
      - Optimizes cases such as `x * 1`, `x * 0`, and `1 * x`.
      - Handles division carefully, ensuring no division by zero.
    - **Exponentiation**:
      - Simplifies common scenarios, e.g., `x^1` or `x^2`.
      - Computes numeric results using `pow()` for constant expressions.
  - **Assignments**:
    - Variables are assigned computed values or identifiers are stored as is.

- Supports propagation of constants and identifiers to streamline code.

Implementation Details

1. **Lexer-Parser Interaction**:
    - The lexer identifies tokens and passes them to the parser.
    - `yylval` is used to transfer token-specific values (e.g., numeric values or variable names).
2. **Symbol Table Updates**:
    - Each assignment updates the symbol table with the computed result or a reference to another variable.
    - Default values (ASCII representations) are assigned to undefined variables.
3. **Optimization Techniques**:
    - **Constant Folding**: Computes operations on numeric constants at compile time (e.g., `x = 3 + 5` becomes `x = 8`).
    - **Simplification**: Eliminates redundant expressions (e.g., `x + 0`, `x * 1`) and substitutes equivalent operations (e.g., `y^2` to `y * y`).
4. **Error Handling**:
    - Detects and reports division by zero.
    - Handles unexpected tokens gracefully, continuing to process valid input.

Conclusion

The implementation efficiently processes and optimizes intermediate code by combining robust lexing, parsing, and expression simplification techniques. Its support for constant folding, algebraic simplifications, and symbol table management provides a strong foundation for further enhancements to the intermediate language. Prioritizing the evaluation of `x` as the last operator highlights the emphasis on reducing unnecessary calculations and improving the robustness of arithmetic optimizations.