CS-315

PROJECT 1

Team 16

Language Name → Aether

23/02/2025

Name - id - section of group members:

Berin Su İyici - 22102342 - 001

Moin Khan - 22101287 - 001

Dilara Kıymaz - 22003308 - 001

1. BNF Description

1.1. Non-Trivial Tokens

<START> ::= start

<L_Bracket> ::= {
<R_Bracket> ::= }

<S_L_Bracket> ::= [
<S_R_Bracket> ::=]

<LP> ::= (
<RP> ::=)
<SLC>::= //
<MLCS>::= /*
<MLCE>::= */

<END_LINE> ::= ;

<ASSIGN_OP> ::= =

<LOW_LETTER> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<UP_LETTER> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<NL> ::= \n
<TAB> ::= \t
<SPACE> ::=

<DIGIT> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<PLUS_SIGN> ::= +
<MINUS_SIGN> ::= -
<DIV_SIGN> ::= /
<MULT_SIGN> ::= *
<MOD_SIGN> ::= %
<POW_SIGN> ::= ^

<IF> ::= if
<ELSE_IF> ::= elseif
<ELSE> ::= else

<OR > ::= ||
<AND > ::= &&
<NOT_EQ > ::= !=
<EQ > ::= ==
<LESS > ::= <
<GREATER> ::= >
<GREATER_EQ> ::= >=
<LESS_EQ> ::= <=
<NOT> ::= !

<EMPTY> ::=

<FUNCTION> ::= function

<COMMA> ::= ,

<INCREMENT> ::= ++
<DECREMENT> ::= --

<QUOTE_S> ::= “
<QUOTE_E> ::= ”

<SYMBOLS> ::= ! | ‘ | ^ | + | & | { | } | [|] | ? | \ | , | - | _ | | # | * | \n | _

<DECIMAL>::= .

<FOR> ::= for
<FOR_SEPARATOR> ::= ::
<WHILE> ::= while

<RETURN> ::= return

<BOOL> ::= 1 | 0

<PRINT> ::= print

<INPUT> ::= input
<REAL>::= real
<BOOL_TYPE>::= bool
<VAR_TYPE> ::= real | bool | list
<LIST>::= list
<BOOL> ::= 001 | 000
<END> ::= end

1.2. Primitive Types

<LETTER> ::= <LOW_LETTER> | <UP_LETTER>

<IDENTIFIER> ::= <LETTER> <IDENTIFIER_TAIL>
<IDENTIFIER_TAIL> ::= <LETTER> <IDENTIFIER_TAIL> | <DIGIT> <IDENTIFIER_TAIL> | <EMPTY>
<DIGITS>::= <DIGIT> | <DIGITS><DIGIT>
<CHAR> ::= <SYMBOLS> | <LETTER> | <DIGITS> | <DECIMAL>

1.3. Program Definition

<program> ::= <START><block><END_LINE><END>

<block> ::= <L_Bracket> <statement_list> <R_Bracket>

<statement_list> ::= <statement>

| <statement> <statement_list>

<statement> ::= <assignment_statement>

| <if_statement>

| <while_statement>

| <for_statement>

| <function_definition>

| <function_call_statement>

| <print_statement>

| <input_statement>

| <collection_declaration>

| <comment>

| <return_statement>

| <increment_decrement_statement>

<variable_declaration> ::= <real_declaration>

| <bool_declaration>

| <list_declaration>

<real_declaration> ::= <REAL> <IDENTIFIER> <END_LINE>

| <REAL> <IDENTIFIER> <ASSIGN_OP> <arithmetic_expression> <END_LINE>

<bool_declaration> ::= <BOOL_TYPE> <IDENTIFIER> <END_LINE>

| <BOOL_TYPE> <IDENTIFIER> <ASSIGN_OP> <bool_expression> <END_LINE>

<list_declaration> ::= <LIST> <IDENTIFIER> <END_LINE>

| <LIST> <IDENTIFIER> <ASSIGN_OP> <list_expression> <END_LINE>

<list_expression> ::= <S_L_Bracket> <real_list> <S_R_Bracket> | <IDENTIFIER>

<list_indexing> ::= <IDENTIFIER> <S_L_Bracket> <real_index> <S_R_Bracket>

<real_index> ::= <DIGITS>

| <IDENTIFIER>

<real_list> ::= <real_number>

| <real_number> <COMMA> <real_list>

<assignment_statement> ::= <real_assignment>

| <bool_assignment>

| <list_assignment>

| <list_indexing> <ASSIGN_OP> <arithmetic_expression> <END_LINE>

<real_assignment> ::= <IDENTIFIER> <ASSIGN_OP> <arithmetic_expression> <END_LINE>

<bool_assignment> ::= <IDENTIFIER> <ASSIGN_OP> <bool_expression> <END_LINE>

<list_assignment> ::= <IDENTIFIER> <ASSIGN_OP> <list_expression> <END_LINE>

<if_statement> ::= <if_without_else>

| <if_with_else>

<if_without_else> ::= <IF> <LP> <bool_expression> <RP><block>

<if_with_else> ::= <IF> <LP> <bool_expression> <RP> <block> <else_part>

<else_part> ::= <ELSE_IF> <LP> <bool_expression> <RP><block> <else_part>

| <ELSE> <block>

<while_statement> ::= <WHILE> <LP> <bool_expression> <RP> <block>

<for_statement> ::= <FOR> <LP> <optional_init_for> <FOR_SEPARATOR> <bool_expression> <FOR_SEPARATOR> <opt_increment_for> <RP> <block>

<optional_init_for> ::= <init_for>

| <EMPTY>

<init_for> ::= <IDENTIFIER> <ASSIGN_OP> <arithmetic_expression> | <REAL> <IDENTIFIER> <ASSIGN_OP> <arithmetic_expression>

<optional_increment_for> ::= <increment_for>

| <EMPTY>

<increment_for> ::= <IDENTIFIER> <ASSIGN_OP> <arithmetic_expression>

<function_definition> ::= <FUNCTION> <IDENTIFIER> <LP> <op_parameter_list> <RP> <function_block>

<function_block> ::= <L_Bracket> <statement_list> <op_return_statement> <R_Bracket>

<op_parameter_list> ::= <parameter_list>

| <EMPTY>

<parameter_list> ::= <parameter>

| <parameter> <COMMA> <parameter_list>

<parameter> ::= <VAR_TYPE> <IDENTIFIER>

<op_return_statement> ::= <return_statement> | <EMPTY>

<return_statement> ::= <RETURN> <expression> <END_LINE>

| <RETURN> <END_LINE>

<function_call_statement> ::= <IDENTIFIER> <LP> <op_argument_list> <RP> <END_LINE>

<op_argument_list> ::= <argument_list> | <EMPTY>

<argument_list> ::= <expression> | <expression> <COMMA> <argument_list>.

<comment> ::= <single_line_comment>

| <multi_line_comment>

<single_line_comment> ::= <SLC> <comment_content>

<multi_line_comment> ::= <MLCS> <comment_content> <MLCE>

<comment_content> ::= <CHAR> <comment_content>

| <CHAR>

| <EMPTY>

<print_statement> ::= <PRINT> <LP><printable_list><RP><END_LINE>

<printable_list> ::= <printable_item>

| <printable_item> <COMMA> <printable_list>

<printable_item> ::= <QUOTE_S><string_literal><QUOTE_E>

| <IDENTIFIER>

| <expression>

<string_literal> ::= <CHAR> <string_literal>

| <EMPTY>

<input_stmt> ::= <INPUT> <LP> <IDENTIFIER> <RP> <END_LINE>

<increment_decrement_statement>::= <IDENTIFIER> <INCREMENT> <END_LINE>

| <IDENTIFIER> <DECREMENT> <END_LINE>

<real_number> ::= <optional_sign> <DIGITS> | <optional_sign> <DIGITS> <DECIMAL> <DIGITS>

<optional_sign>::= <EMPTY> | -

<expression> ::= <arithmetic_expression> | <bool_expression>

<bool_expression> ::= <bool_term>

| <bool_expression> <OR> <bool_term>

<bool_term> ::= <bool_factor>

| <bool_term> <AND> <bool_factor>

<bool_factor> ::= <NOT> <bool_factor>

| <relational_expression>

| <LP> <bool_expression> <RP>

| <BOOL>

<relational_expression> ::= <arithmetic_expression> <relational_ops> <arithmetic_expression>

<relational_ops> ::= <LESS>

| <LESS_EQ>

| <GREATER>

| <GREATER_EQ>

| <EQ>

| <NOT_EQ>

<arithmetic_expression> ::= <term>

| <arithmetic_expression> <additive_ops> <term>

<additive_ops> ::= <PLUS_SIGN> | <MINUS_SIGN>

<term> ::= <factor>

| <term> <mult_ops> <factor>

<mult_ops> ::= <MULT_SIGN>

| <DIV_SIGN>

| <MOD_SIGN>

<factor> ::= <primary>

| <primary> <POW_SIGN> <factor>

<primary> ::= <IDENTIFIER>

| <real_number>

| <function_call_statement>

| <LP> <arithmetic_expression> <RP>

| <IDENTIFIER> <INCREMENT>

| <IDENTIFIER> <DECREMENT>

2. Reserved Words

- **start** – Marks the beginning of a program; it is the entry point for execution.
- **end** – Marks the end of the program.
- **if** – Used to initiate a conditional statement, allowing branching based on boolean expressions.
- **elseif** – Specifies an additional condition if the initial if condition is false.
- **else** – Specifies an alternative block of code that executes when the if condition is false.
- **for** – Defines a for loop, used for iterating over a range or a collection.
- **while** – Initiates a while loop, which continues execution as long as a condition remains true.
- **return** – Indicates returning a value from a function or exits a function without returning a value.
- **001** – Represents the boolean value true; used in boolean expressions.
- **000** – Represents the boolean value false; used in boolean expressions.
- **print** – Represents a built-in function for outputting data to the console.
- **input** – Represents a built-in function for receiving user input.
- **bool** – Specifies that a variable is of boolean type (1 or 0).
- **real** – Defines real numbers, the only numeric type in the language.
- **list** – Used to declare a collection of real numbers.
- **function** – Defines a function that can take parameters and return a value.

Special Operators & Symbols

- **=** – Assignment operator.

- **+, -, *, /, %, ^** – Arithmetic operators.
- **<, <=, >, >=, ==, !=** – Relational operators.
- **||, &&, !** – Logical operators.
- **::** – Used as a separator in for loops.

3. Descriptions

3.1 Program Structure

- <START> start: Marks the beginning of a program.
- <END> end: Marks the termination of the program.

3.2 Brackets and Parentheses

- <L_Bracket> {: Opens a block of code.
- <R_Bracket> }: Closes a block of code.
- <S_L_Bracket> [: Used for list (array) declarations.
- <S_R_Bracket>]: Closes a list (array) declaration.
- <LP> (: Opens an expression grouping or function parameter list.
- <RP>): Closes an expression grouping or function parameter list.

3.3 Statements and Operators

- <END_LINE> ;: Terminates a statement.
- <ASSIGN_OP> =: Assigns values to variables.

3.4 Characters and Identifiers

- <LOW_LETTER>: Represents lowercase letters a-z.
- <UP_LETTER>: Represents uppercase letters A-Z.
- <LETTER>: Represents either <LOW_LETTER> or <UP_LETTER>.
- <DIGIT>: Represents digits 0-9.
- <IDENTIFIER>: A variable name, starting with a letter and followed by letters or digits.
- <IDENTIFIER_TAIL>: Allows additional letters or digits after the first character in an identifier.
- <DIGITS>: A sequence of one or more digits.

3.5 Whitespace and Formatting

- <SPACE>: Represents a space character.
- <NL> \n: Represents a newline.
- <TAB> \t: Represents a tab.

3.6 Arithmetic Operators

- <PLUS_SIGN> +: Addition.
- <MINUS_SIGN> -: Subtraction.
- <MULT_SIGN> *: Multiplication.
- <DIV_SIGN> /: Division.
- <MOD_SIGN> %: Modulus.
- <POW_SIGN> ^: Exponentiation.

3.7 Comparison and Logical Operators

- <LESS> <: Less than.
- <GREATER> >: Greater than.
- <LESS_EQ> <=: Less than or equal to.
- <GREATER_EQ> >=: Greater than or equal to.
- <EQ> ==: Equality check.
- <NOT_EQ> !=: Inequality check.
- <AND> &&: Logical AND.
- <OR> ||: Logical OR.
- <NOT> !: Logical NOT.

3.8 Control Flow Keywords

- <IF> if: Begins an if conditional block.
- <ELSE_IF> elseif: Specifies an alternative condition.
- <ELSE> else: Specifies the default block when all other conditions fail.
- <FOR> for: Declares a loop with initialization, condition, and increment.
- <FOR_SEPARATOR> ::: Separates expressions in a for loop.
- <WHILE> while: Declares a loop that runs while a condition is true.

3.9 Function and Variable Handling

- <FUNCTION> function: Declares a function.

- <RETURN> return: Specifies the return value of a function.
- <VAR_TYPE>: Represents available primitive types (real, bool, list).
- <REAL> real: Represents the real number data type.
- <BOOL_TYPE> bool: Represents the boolean data type.
- <LIST> list: Represents a list data type.
- <BOOL>: Represents boolean values (1 for true, 0 for false`).

3.10 Increment and Decrement Operators

- <INCREMENT> ++: Increases a variable's value by one.
- <DECREMENT> --: Decreases a variable's value by one.

3.11 I/O Operations

- <PRINT> print: Prints output to the console.
- <INPUT> input: Reads input from the user.

3.12 String Handling

- <QUOTE_S> “: Opens a string literal.
- <QUOTE_E> ”: Closes a string literal.
- <string_literal>: Represents a sequence of characters enclosed in quotes.

3.13 Comments

- <SLC> //: Denotes the start of a single-line comment.
- <MLCS> /*: Marks the beginning of a multi-line comment.
- <MLCE> */: Marks the end of a multi-line comment.
- <comment_content>: Represents the content inside comments, which can contain characters, symbols, and whitespace.

3.14 Numeric and Boolean Representation

- <BOOL>: Represents boolean values (1 for true, 0 for false).
- <real_number>: Represents floating-point numbers.
- <DECIMAL> .: Used to define real numbers.

- <optional_sign>: Allows numbers to be positive or negative (- for negative, empty for positive).

3.15 Expressions

- <expression>: Can be an arithmetic or boolean expression.
- <arithmetic_expression>: Represents numerical calculations.
- <bool_expression>: Represents boolean logic.
- <relational_expression>: Compares values.

3.16 Mathematical Structure

- <additive_ops>: Includes <PLUS_SIGN> + and <MINUS_SIGN> -.
- <mult_ops>: Includes <MULT_SIGN> *, <DIV_SIGN> /, <MOD_SIGN> %.
- <factor>: Can be an identifier, real number, function call, or a parenthesized expression.
- <primary>: Represents a variable, a number, a function call, or increment/decrement operations.

3.17 Function Definition and Calls

- <function_definition>: Defines a function with parameters.
- <function_call_statement>: Calls a function.
- <parameter_list>: List of parameters inside a function.
- <op_argument_list>: Optional arguments for a function call.

3.18 List Handling

- <list_expression>: Defines a list using square brackets [ ] with real numbers separated by commas.
- <list_indexing>: Accesses list elements using an index.
- <real_index>: Represents either a number or a variable used as an index.

3.19 Symbols and Special Characters

- <SYMBOLS>: Includes various special characters:
    - !, ', ^, +, &, {, }, [,], ?, \, ,, -, _, #, *, \n, _.

4 Design Choice Explanations

Aether is designed to prioritize readability, reliability, and structured simplicity while ensuring that all variables store only real values. This design decision stems from a strict project requirement that prohibits non-numeric data storage, leading us to enforce a strongly typed system where all variables—whether representing numbers, booleans, or lists—are ultimately treated as real values. **Booleans are strictly constrained to 001 (true) and 000 (false), ensuring explicit distinction from numeric values. This avoids potential confusion and floating-point precision issues that could arise if arbitrary real values were used in logical expressions.**

To maintain clarity and prevent unintended side effects, Aether enforces strict syntax rules, including explicit block delimiters `{}` and mandatory semicolons `;`. **Unlike some indentation-based languages, indentation in Aether is purely for readability and does not affect execution.** While Aether does not support a dedicated string data type, **string literals can still be passed directly to the `print()` function, allowing for essential output functionality without violating the numeric-only storage constraint.**

The language's structured control flow, explicit typing, and minimalistic yet expressive constructs aim to reduce cognitive load, prevent logical ambiguities, and create a reliable programming environment. **While this approach limits some common programming patterns, such as text manipulation and heterogeneous data structures (lists in Aether are restricted to real numbers), it ensures that numerical computations and logical operations remain deterministic and easy to reason about.** By balancing these trade-offs, Aether provides a clear, structured, and maintainable programming environment that aligns with the project's strict data representation requirements.

4.1 Non-trivial Tokens

- **Comments:** Aether supports both single-line (//) and multi-line (/* ... */) comments. This enhances readability by allowing programmers to document their code effectively, making it easier to understand and maintain. Multi-line comments are particularly useful for providing explanations for complex logic without cluttering the main code structure.
- **Identifiers:** Identifiers begin with a letter (uppercase or lowercase) followed by any combination of letters and digits. Special symbols are not allowed to ensure simplicity and avoid ambiguity in parsing. By enforcing this structure, Aether ensures better readability, as variable and function names remain clear and distinct from operators and keywords.
- **Literals:**
    - **Real Numbers:** Numbers are represented using the real type, allowing decimal points for floating-point representation. The explicit use of real

ensures that developers do not have to infer the numeric type, thereby reducing errors and improving clarity.
    - **Boolean Values:** Booleans are represented using 001 (true) and 000 (false) instead of 1 and 0 to avoid confusion with numeric values. This prevents accidental misuse where a numeric zero might be incorrectly interpreted as false and enhances code reliability by making boolean expressions explicit.
- **Reserved Words:** Keywords such as if, for, while, return, function, and end are reserved for language constructs. This enforces consistency and prevents variable names from overlapping with essential control structures, improving code predictability.

4.2 Primitive Types

Primitive types were kept minimal to simplify implementation while covering common computational needs. This simplification helps reduce the learning curve for new users and makes the language more intuitive and less costly in this sense.

Aether supports three primitive data types:

- **real**: Represents floating-point numbers, allowing decimal arithmetic. The explicit declaration of numerical types avoids confusion between integer and floating-point operations, ensuring predictable behavior.
- **bool**: Represents boolean values (001 for true, 000 for false). By using distinct representations instead of common 1 and 0, Aether eliminates ambiguity between boolean and numeric values, enhancing reliability.
- **list**: A collection of real numbers stored in a sequence. Lists are mutable, which provides the user with more control and efficiency for programs that need real-time updates to the elements. Mutable lists allow in-place modifications, enabling efficient updates, dynamic resizing, and flexible data management, but they require careful handling to ensure code safety and predictable behavior.

4.3 Program Definition

An Aether program follows a structured block format:

- **start** and **end** clearly define program boundaries. This ensures that code has a well-defined beginning and end, making it easier to read and debug.
- **Curly braces {} encapsulate blocks**, similar to C-style languages. This provides a visual cue for grouping statements together, improving code organization and comprehension.
- **Semicolon ; is used to terminate statements**, enforcing explicit syntax that avoids confusion in multi-line expressions.

5 Lex Specification

The provided Lex specification file is written with testing purposes in mind, as also specified in the project requirements. The unconventional usage of print statements, along with the approach of recognizing only the start and end characters of comment sections while ignoring their content, will be revised later in the second part of the project.

```
%option main

%%

"start"           { printf("START\n"); }
"end"             { printf("END\n"); }
"if"              { printf("IF\t"); }
"elseif"          { printf("ELSE_IF\t"); }
"else"            { printf("ELSE\t"); }
"for"             { printf("FOR\t"); }
"while"           { printf("WHILE\t"); }
"return"          { printf("RETURN\t"); }
"print"           { printf("PRINT\t"); }
"input"           { printf("INPUT\t"); }
"real"            { printf("REAL\t"); }
"bool"            { printf("BOOL_TYPE\t"); }
"list"            { printf("LIST\t"); }
"function"        { printf("FUNCTION\t"); }
"void"            { printf("VOID\t"); }

"000"|"001"           { printf("BOOL_LITERAL\t"); }


"="               { printf("ASSIGN_OP\t"); }
"+"               { printf("PLUS_SIGN\t"); }
"-"               { printf("MINUS_SIGN\t"); }
"*"               { printf("MULT_SIGN\t"); }
"/"               { printf("DIV_SIGN\t"); }
"%"               { printf("MOD_SIGN\t"); }
"^"               { printf("POW_SIGN\t"); }

"<"               { printf("LESS\t"); }
"<="              { printf("LESS_EQ\t"); }
">"               { printf("GREATER\t"); }
">="              { printf("GREATER_EQ\t"); }
"=="              { printf("EQ\t"); }
"!="              { printf("NOT_EQ\t"); }

"||"              { printf("OR\t"); }
"&&"              { printf("AND\t"); }
```

```
"!"                 { printf("NOT\t"); }

"{"                 { printf("L_Bracket\n"); }
"}"                 { printf("R_Bracket\n"); }
"["                 { printf("S_L_Bracket\t"); }
"]"                 { printf("S_R_Bracket\t"); }
"("                 { printf("LP\t"); }
")"                 { printf("RP\t"); }
";"                 { printf("END_LINE\n"); }
","                 { printf("COMMA\t"); }

"++"                { printf("INCREMENT\t"); }
"--"                { printf("DECREMENT\t"); }

"::"                { printf("FOR_SEPARATOR\t"); }

"//".*              { printf("SLC\n"); }  // Single-line comment
"/*"([^*]|\*+[^*/])*\*+"/" { printf("MLCS MLCE\n"); }  //
Multi-line comment

[0-9]+(\.[0-9]+)? { printf("REAL_NUMBER\t"); }
[a-zA-Z_][a-zA-Z0-9_]* { printf("IDENTIFIER\t"); }
\".*\"              { printf("STRING_LITERAL\t"); }
[ \t\n\r]+          { /* Ignore whitespace */ }

.                   { printf("UNKNOWN\n"); }

%%
```

6 Example Programs

6.1 Program 1

```
start{
     real x;
     real y;
     real z;

     print("Enter x:");
     input(x);

     print("Enter y:");
     input(y);

     print("Enter z:");
     input(z);
```

```
	while( (x == 0) || (y == 0) || (z == 0) ){
		print("Please enter only non-zero values.");
		input(x);
		input(y);
		input(z);
	};

	print(x * y * z);
};
end
```

6.2 Program 2

```
start{
	function foo(real p, real q){
		print("Function name: foo");
		print("Parameter p = ", p);
		print("Parameter q = ", q);

	if( p > q ){
		return p;
	};
	return q;
	};

	list list_1 = [3.15, 7, 0.03, -1.7];
	list list_2 = [9, -1.2, 1.2];

	real a;
	real b;
	real c;
	real i;
	real j;

	for(i = 0 :: i < 4 :: i = i + 1){
		a = list_1[i];

		for(j = 0 :: j < 3 :: j = j + 1){
			b = list_2[j];
			c = foo(a, b);
			print("a=", a, ", b=", b, ", c=", c);
		};
	};
};
```

```
end
```

6.3 Program 3

```
start {

     /*
     * Aether Language Test Program
     * ----------------------------
     * This program demonstrates the core features of Aether,
including:
     * - Variable declarations (real, bool, list)
     * - Arithmetic operations (addition, multiplication,
division, exponentiation)
     * - Boolean logic with 001 (true) and 000 (false)
     * - Conditional statements (if-else)
     * - Looping structures (for, while)
     * - Function definition and calling
     * - List manipulation and indexing
     * - Printing both values and string literals
     *
     * Aether enforces strict real-number typing,
     * so booleans are represented as 001 (true) and 000 (false).
     * Numbers can be written as whole numbers (e.g., 1) or with
floating points (e.g., 1.5).
     */

    // Declare and initialize real variables with both whole and
floating-point numbers
    real x = 5;
    real y = 10.25;
    real z = 3.5;

    // Perform arithmetic operations
    real sum = x + y + z;
    real product = x * y * z;
    real quotient = y / z;
    real power = x ^ 2;

    // Declare boolean values using 001 (true) and 000 (false)
    bool isGreater = 001;  // True
    bool isSmaller = 000;  // False

    // Print computed values
    print("Sum:", sum);
```

```
    print("Product:", product);
    print("Quotient:", quotient);
    print("Power of x^2:", power);

    // Conditional check using boolean values
    if (x < y) {
        print("x is less than y");
    } else {
        print("x is greater than or equal to y");
    }

    // Declare and initialize a list with both whole and
floating-point numbers
    list numbers = [1, 2.5, 3, 4.75, 5];

    // Loop through the list using a for loop
    for (real i = 0 :: i < 5 :: i = i + 1) {
        print("List item:", numbers[i]);
    }

    // Define a function to add two real numbers
    function add(real a, real b) {
        return a + b;
    }

    // Call the function and store the result
    real result = add(x, y);
    print("Function result (x + y):", result);

    // Demonstrate a while loop
    real counter = 0;
    while (counter < 3) {
        print("Counter value:", counter);
        counter++;
    }

}; end
```