CS-315

PROJECT 2

Team 16

Language Name → Aether

15/03/2025

Name - id - section of group members:

Berin Su İyici - 22102342 - 001

Moin Khan - 22101287 - 001

Dilara Kıymaz - 22003308 - 001

1. Program BNF Description

Below is the full Backus–Naur Form (BNF) of the language. In this notation, terminal symbols are shown in bold (or as their literal string if they are keywords or punctuation), and nonterminals are written in *italics*. Note that tokens such as START, END, IF, etc., are those returned by your lexer.

<program::=

<START> <block> <END_LINE> <END>

<block> ::=

<L_Bracket> <statement_list> <R_Bracket>

<statement_list> ::=

<EMPTY>

| <statement_list> <statement>

<statement> ::=

<assignment_statement>

| <expression_statement>

| <if_statement>

| <loop_statement>

| <function_definition>

| <print_statement>

| <input_statement>

| <variable_declaration>

| <MLCS>

| <COMMENT>

| <return_statement>

| <increment_decrement_statement>

<lvalue> ::=

<IDENTIFIER>

| <IDENTIFIER> <S_L_Bracket> <expression> <S_R_Bracket>

<assignment_statement> ::= <lvalue> <ASSIGN_OP> <expression> <END_LINE>

<expression_statement> ::= <expression> <END_LINE>

<expression> ::= <logical_or_expr>

<logical_or_expr> ::=

<logical_or_expr> <OR> <logical_and_expr>

| <logical_and_expr>

<logical_and_expr> ::=

<logical_and_expr> <AND> <equality_expr>

| <equality_expr>

<equality_expr> ::=

<equality_expr> <EQ> <relational_expr>

| <equality_expr> <NOT_EQ> <relational_expr>

| <relational_expr>

<relational_expr> ::=

<relational_expr> <LESS> <additive_expr>

| <relational_expr> <LESS_EQ> <additive_expr>

| <relational_expr> <GREATER> <additive_expr>

| <relational_expr> <GREATER_EQ> <additive_expr>

| <additive_expr>

<additive_expr> ::=

<additive_expr> <PLUS_SIGN> <mult_expr>

| <additive_expr> <MINUS_SIGN> <mult_expr>

| <mult_expr>

<mult_expr> ::=

<mult_expr> <MULT_SIGN> <pow_expr>

| <mult_expr> <DIV_SIGN> <pow_expr>

| <mult_expr> <MOD_SIGN> <pow_expr>

| <pow_expr>

<pow_expr> ::=

<primary> <POW_SIGN> <pow_expr>

| <primary>

<primary> ::=

<IDENTIFIER>

| <REAL_NUMBER>

| <BOOL_LITERAL>

| <LP> <expression> <RP>

| <list_literal>

| <NUMBERS>

<list_literal> ::=

<S_L_Bracket> <real_list> <S_R_Bracket>

<real_list> ::=

<number_value>

| <number_value> <COMMA> <real_list>

<number_value> ::=

<REAL_NUMBER>

| <NUMBERS>

<if_statement> ::=

<IF> <LP> <expression> <RP> <block> <END_LINE>

| <IF> <LP> <expression> <RP> <block> <else_part> <END_LINE>

<else_part> ::=

<ELSEIF> <LP> <expression> <RP> <block> <else_part>

| <ELSE> <block>

<loop_statement> ::=

<WHILE> <LP> <expression> <RP> <block> <END_LINE>

|<FOR><LP><optional_init><FOR_SEPARATOR><expression> <FOR_SEPARATOR> <optional_inc> <RP> <block> <END_LINE>

<optional_init> ::=

<EMPTY>

| <init_for>

<init_for> ::=

<IDENTIFIER> <ASSIGN_OP> <expression>

| <REAL> <IDENTIFIER> <ASSIGN_OP> <expression>

optional_inc ::=

<EMPTY>

| <inc_for>

inc_for ::=

<IDENTIFIER> <ASSIGN_OP> <expression>

function_definition ::=

<FUNCTION><IDENTIFIER><LP><opt_param_list><RP><function_block><END_LINE>

< function_block> ::=

<L_Bracket> <statement_list> <R_Bracket>

<opt_param_list ::=

<EMPTY>

| <param_list>

<param_list ::=

<parameter>

| <parameter> <COMMA> <param_list>

<parameter ::=

<var_type> <IDENTIFIER>

<var_type ::=

<BOOL_TYPE>

| <REAL>

| <LIST>

<return_statement ::=

<RETURN > <expression> <END_LINE>

| <RETURN> <END_LINE>

<print_statement ::=

<PRINT> <LP> <print_list> <RP> <END_LINE>

<print_list ::=

<print_item>

| <print_item> <COMMA> < print_list>

<print_item> ::=

<STRING_LITERAL>

| <expression>

<input_statement> ::=

<INPUT> <LP> <IDENTIFIER> <RP> <END_LINE>

<variable_declaration> ::=

<REAL> <IDENTIFIER> <END_LINE>

| <REAL><IDENTIFIER><ASSIGN_OP><expression><END_LINE>

| <BOOL_TYPE><IDENTIFIER> <END_LINE>

|<BOOL_TYPE><IDENTIFIER><ASSIGN_OP><expression><END_LINE

|<LIST><IDENTIFIER><END_LINE>

|<LIST><IDENTIFIER><ASSIGN_OP><list_init><END_LINE>

<list_init> ::=

<list_literal>

| <IDENTIFIER>

<increment_decrement_statement> ::=

<IDENTIFIER> <INCREMENT> <END_LINE>

| <IDENTIFIER><DECREMENT><END_LINE>

2. General Description of the Language Structure

- **Program Structure:**
  Every program must start with the keyword start and end with end. The main block of the program is enclosed within curly braces ({ and }), making it clear where the program's scope begins and ends. A semicolon (represented by the token END_LINE) follows the block and precedes the end keyword.
- **Blocks and Statements:**
  The language uses blocks (delimited by { and }) to group statements. A block may contain a list of statements, which include assignments, expressions, control structures (if, for, while), function definitions, input/output operations, variable declarations, comments, and return statements.
- **Expressions:**
  Expressions form a hierarchical structure, starting with logical operations (OR, AND) and moving down through equality, relational, additive, multiplicative, and power operations. Parentheses can be used to override precedence. The language also supports list literals and both real and integer number types.
- **Control Structures:**
  - **If-Statements:** The language supports if statements with optional elseif and else parts.
  - **Loops:** Two types of loops are supported—while loops and for loops. In for loops, the initialization, condition, and increment parts are clearly separated using the special token :: (denoted by FOR_SEPARATOR).
- **Functions:**
  Functions are declared using the function keyword, followed by an identifier, an optional parameter list (with types real, bool, or list), and a block that defines the function body.
- **I/O Operations:**
  The language includes statements for printing (with support for both string literals and expressions) and for taking input.
- **Comments:**
  Both single-line comments (// …) and multi-line comments (/* … */) are supported.
- **Variable Declarations and Increment/Decrement:**
  Variables can be declared with types real, bool, or list. In addition, the language supports increment (++) and decrement (--) operations as separate statements.

3. Important Nonterminals

The following nonterminals are particularly important for understanding the structure of the language:

- **program:**
  The entry point that defines the overall structure, enforcing that a program begins with start and ends with end.
- **block:**
  Represents a group of statements enclosed in curly braces.
- **statement_list:**
  A potentially empty sequence of statements within a block.
- **statement:**
  The core nonterminal representing any valid statement (assignment, control structure, function definition, etc.).
- **expression:**
  Defines the expression hierarchy used throughout the language, covering logical, relational, arithmetic, and power operations.
- **if_statement & else_part:**
  They define conditional execution with support for multiple elseif clauses and an optional else clause.
- **loop_statement:**
  Encapsulates the syntax for both while and for loops.
- **function_definition:**
  Describes how functions are declared, including the parameter list and the function body.
- **variable_declaration:**
  Specifies how variables are declared and optionally initialized.

These nonterminals form the backbone of the language's grammar, and understanding them is key to parsing and writing programs in this language.

4. How to Parse a Program Written in the Language

A reader should understand that a valid program must adhere to the following overall pattern:

1. **Program Entry:**
   The program starts with the `start` keyword, followed by a block enclosed in `{ ... }`, a semicolon (`;`), and ends with the `end` keyword.
2. **Inside the Block:**
   a. The block contains a list of *statements*.
   b. Each statement ends with a semicolon, except those that themselves are blocks (like the body of an if or a loop
3. **Statements:**
   - Assignment statements follow the pattern:
     identifier = expression;
     Optionally, the left-hand side can be an element of a list (using square brackets).
   - Expressions are built using standard arithmetic and logical operators. Parentheses can be used for grouping.
   - Control structures (if, elseif, else, while, for) clearly define their conditions (within parentheses) and bodies (within curly braces).
   - Function definitions use a similar block structure and include a parameter list with type declarations.
   - I/O statements use the keywords print and input with appropriate delimiters.
   - Variable declarations specify the type (real, bool, list) followed by an identifier, with an optional initialization.

5. Precedence and Ambiguity Resolution Rules

The grammar adopts the following rules to manage operator precedence and resolve ambiguities:

- **Dangling Else Resolution:**
  The nonassociative tokens LOWER_THAN_ELSE and ELSE are used to resolve the dangling else ambiguity so that each else is associated with the closest preceding unmatched if.
- **Operator Precedence (from lowest to highest):**
  - **OR:** Left associative
    (logical_or_expr)
  - **AND:** Left associative
    (logical_and_expr)
  - **Equality operators (EQ, NOT_EQ):** Nonassociative
    (equality_expr)
  - **Relational operators (LESS, LESS_EQ, GREATER, GREATER_EQ):** Nonassociative
    (relational_expr)
  - **Additive operators (PLUS_SIGN, MINUS_SIGN):** Left associative
    (additive_expr)
  - **Multiplicative operators (MULT_SIGN, DIV_SIGN, MOD_SIGN):** Left associative
    (mult_expr)
  - **Power operator (POW_SIGN):** Right associative
    (pow_expr)
  - **NOT operator:** Right associative

These declarations ensure that, for example, multiplication binds tighter than addition and that the power operator is right associative (i.e. a ^ b ^ c is parsed as a ^ (b ^ c)).

6. Description of Nontrivial Tokens and Their Usage

- **Keywords and Types:**
  - **START / END:**
    Mark the beginning and end of a program.
  - **IF / ELSEIF / ELSE:**
    Used to construct conditional statements.
  - **FOR / WHILE:**
    Define loop structures.
  - **RETURN:**
    Used within functions to return a value.
  - **PRINT / INPUT:**
    For outputting data and reading input from the user.
  - **REAL / BOOL_TYPE / LIST:**
    Used in variable declarations and function parameter definitions to denote types.
  - **FUNCTION / VOID:**
    Declare functions; although VOID is present, it isn't used explicitly in the productions shown.
  - **EMPTY**: As a placeholder for empty spaces.
- **Literals and Identifiers:**
  - **IDENTIFIER:**
    Matches variable names and function names.
  - **REAL_NUMBER / NUMBERS:**
    Distinguish between real (floating-point) numbers and integer numbers.
  - **BOOL_LITERAL:**
    Represents boolean values (e.g., "000" or "001", as defined by the lexer).
  - **STRING_LITERAL:**
    Represents strings, typically used in print statements.
- **Operators:**

    - **ASSIGN_OP (=):**
      Used for assignment.
    - **PLUS_SIGN (+), MINUS_SIGN (-), MULT_SIGN (*), DIV_SIGN (/), MOD_SIGN (%), POW_SIGN (^):**
      Standard arithmetic operators with defined associativity.
    - **LESS (<), LESS_EQ (<=), GREATER (>), GREATER_EQ (>=), EQ (==), NOT_EQ (!=):**
      Relational and equality operators.
    - **OR (||), AND (&&), NOT (!):**
      Logical operators.
    - **INCREMENT (++), DECREMENT (--):**
      Special tokens for incrementing or decrementing a variable.
- **Punctuation and Grouping:**
    - **L_Bracket / R_Bracket ({ / }):**
      Define blocks.
    - **S_L_Bracket / S_R_Bracket ([/]):**
      Used for list literals and indexing.
    - **LP / RP ((/)):**
      Group expressions and enclose conditions.
    - **END_LINE (;):**
      Terminates statements.
    - **COMMA (,):**
      Separates parameters, list elements, or items in print statements.
    - **FOR_SEPARATOR (::):**
      Separates the initialization, condition, and increment parts in a for loop.
    - **QUOTE:**
      Although generated by the lexer, it is not used explicitly in the grammar rules; it may be relevant for processing string literals.
- **Comments:**
    - **COMMENT:**
      Single-line comments beginning with //.
    - **MLCS:**
      Multi-line comments, enclosed in /* … */.

7. 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.

7.1 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.

7.2 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.

7.3 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.

8. Summary

In summary, this language is structured around a clear block format with a mandatory start ... end program structure. It supports common programming constructs such as expressions (with a rich operator hierarchy), control flow (if/else, loops), functions with typed parameters, I/O, and variable declarations. Precedence rules and nonassociative declarations help resolve ambiguities (especially the “dangling else” problem), and each nontrivial token (from arithmetic and logical operators to list and comment tokens) is carefully integrated into the grammar.

This report should provide a comprehensive guide for anyone looking to understand or write programs in the language defined.