CS 315 – Programming Languages

PROJECT 2 – Group 11

REALANG

Group Members:

Yunus Günay [22203758] [Section 1]

Daib Malik [22201013] [Section 1]

Burak Kağan Gür [22203365] [Section 1]

BNF of Realang

1. Program:

```
<program> ::= START <statement_list> END
<statement_list> ::= <statement> | <statement_list> <statement>
```

2. Statements:

```
<statement> ::= <declaration> | <assignment> | <conditional>
              | <loop_stmt> | <io_stmt> | <function_def> | <return_stmt>
              | <list_declaration> | <list_func> | <comment>
```

2.a) Declarations & Assignments

```
<declaration> ::= REAL <assignment>
                | REAL <identifier_list> SC
<identifier_list> ::= IDENTIFIER| <identifier_list> COMMA IDENTIFIER
<assignment> ::= IDENTIFIER ASSIGN <real_expr> SC
```

2.b) Conditional Statements

```
<conditional> ::= <matched_if> | <unmatched_if>
<matched_if> ::= IF LP <logical_expr> RP <block> ELSE <matched_if>
               | IF LP <logical_expr> RP <block> ELSE <block>
<unmatched_if> ::= IF LP <logical_expr> RP <block>
                 | IF LP <logical_expr> RP <block> ELSE <unmatched_if>
```

2.c) Blocks

```
<block> ::= LBRACE <statement_list> RBRACE
```

2.d) Loop Statements

```
<loop_stmt> ::= WHILE LP <logical_expr> RP <block>
            | FOR LP <for_dec> SC <logical_expr> SC <for_assign> LP <block>
<for_dec> ::= REAL <for_assign> | <for_assign>
<for_assign> ::= IDENTIFIER ASSIGN <real_expr>
```

2.e) I/O Statements

```
<io_stmt> ::= <input_stmt> | <output_stmt>
<input_stmt> ::= IDENTIFIER ASSIGN INPUT LP RP SC
```

```
<output_stmt> ::= PRINT LP <print_args> RP SC
<print_args> ::= STRING | <real_expr> | <print_args> COMMA STRING
               | <print_args> COMMA <real_expr>
```

2.f) Function Definitions and Return Statement

```
<function_def> ::= FUNCTION IDENTIFIER LP <parameter_list_opt> RP <block>
<parameter_list_opt> ::= <parameter_list> | ε
<parameter_list> ::= <parameter> | <parameter_list> COMMA <parameter>
<parameter> ::= REAL IFENTIFIER | REAL REF IDENTIFIER
<return_stmt> ::= RETURN <real_expr> SC
```

2.g) Data Structure to Store Real Numbers and Its Operations

```
<list_declaration> ::= LIST IDENTIFIER ASSIGN LBRACE <real_list_opt> RBRACE
SC
<real_list_opt> ::= <real_list> | ε
<real_list> ::= <real_expr> | <real_list> COMMA <real_expr>
<list_func> ::= IDENTIFIER ADD_FUNC LP <real_expr> RP SC
              | IDENTIFIER REMOVE_FUNC LP <real_expr> RP SC
```

2.h) Comments

```
<comment> ::= COMMENT
```

3. Expressions:

3.a) Logical Expressions

```
<logical_expr> ::= <logical_expr> OR <logical_term> | <logical_term>
<logical_term> ::= <logical_term> AND <logical_factor> | <logical_factor>
<logical_factor> ::=  NOT <logical_factor>
                    | LP <logical_expr> RP
                    | <real_expr> <comparison_op> <real_expr>

<comparison_op> ::= EQUALS | NOT_EQUALS | LESS_EQUAL
                  | GREATER_EQUAL | LESS_THAN | GREATER_THAN
```

3.b) Real Number Expressions

```
<real_expr> ::= <term>
              | <real_expr> ( PLUS | MINUS ) <term>
<term> ::= <factor_term>
```

```
        | <term> ( MUL | DIV | MOD ) <factor_term>
<factor_term> ::= <factor>
                | <factor> POW <factor_term>
<factor> ::= <primary>
           | <unary_op> <primary>
<unary_op> ::= PLUS | MINUS
<primary> ::= REAL_NUM
            | INT
            | IDENTIFIER
            | LP <real_expr> RP
            | <function_call>
            | <list_expr>
```

4. Function Calls and Data Structure Expressions:

```
<function_call> ::= IDENTIFIER LP <argument_list_opt> RP
<argument_list_opt> ::= <argument_list> | ε
<argument_list> ::= <argument> | <argument_list> COMMA <argument>
<argument> ::= <real_expr> | REF IDENTIFIER

<list_expr> ::= IDENTIFIER GET_FUNC LP <index_term> RP
              | IDENTIFIER SIZE_FINC LP RP

<index_term> ::= <index_factor> | <index_term> ( PLUS | MINUS )
<index_factor>
<index_factor> ::= <index_primary>
                 | <index_factor> ( MUL | DIV | MOD ) <index_primary>
<index_primary> ::= INT | IDENTIFIER | LP <index_term> RP
```

Language Constructs: Usage, Meaning, and Conventions

1. Program Structure and Statement List

<program>: A Realang program must begin with the keyword “start” and end with “end,” marking where the program’s instructions start and finish. This boundary makes the language easy to parse and signals the scope of a program.

<statement_list>: A <statement_list> can either be a single statement or a list of them. This design ensures that Realang supports multiple executable statements in sequence. It enforces an orderly flow of execution, improving readability by making the code structure self-evident.

2. Statements

<statement>: Building block representing any instruction in Realang (declaration, assignment, conditional, loop, I/O operation, function definition, return statement, data structure declaration/operation, or comment). Grouping all these constructs under <statement> ensures a uniform structure throughout the language, improving consistency and readability.

2.a) Declarations & Assignments (Limited to Real Numbers)

<declaration> & <assignment>: Realang restricts declarations and assignments to real expressions, ensuring variables always hold numeric values. The production eliminates the need for explicit type declarations but type “real,” simplifying both syntax and usage. Thus, every variable is implicitly real in Realang language.

2.b) Conditionals

<conditional>: Conditionals manage decision-making. It is split into <matched_if> and <unmatched_if>, addressing the classic dangling-else issue.

<matched_if>: Represents a complete if-else structure. It can chain by itself for “else if” scenarios. This chaining approach removes any ambiguity about which if an else matches, promoting reliability in complex decision-making.

<unmatched_if>: Handles incomplete if statements (those without an else or those nested further). By distinguishing these from matched statements, the language ensures conditionals are parsed unambiguously.

2.c) Blocks

<block>: A block encloses one or more statements within curly braces {…}. This not only indicates scope clearly but also fosters readibility and reliability, since the start and end of grouped statements are explicit. Additionally, they are used to encapsulate the body of conditionals, loops, and functions.

2.d) Loops

<loop_stmt>: Realang supports while and for loops. The while loop repeatedly executes a <block> as long as a logical condition <logicial_expr> remains true. The for loop syntax includes an initialization, condition, and update statement, making it particularly convenient for numeric iterations.

<for_dec> & <for_assign>: Within a for loop, <for_dec> declares and <for_assign> initializes a loop variable, such as, "for(real i = 0; <logical_expr>; i = i + 1)" or "for(i = A.getRealNum(2); <logical_expr>; i = i + foo(5))"

2.e) Inputs and Outputs

<io_stmt>: Collects input and output operations under one umbrella for clarity.

<input_stmt>: Assigns the result of an input() call to an variable, such as, "x=input();" Its syntax indicates that the value read from input is stored in the given identifier.

<output_stmt>: Uses print(…) to output one or more <print_args> (strings or numeric expressions), optionally separated by commas. This design makes it easy to produce formatted output, such as, "print("Sum of x+y is ", (x+y), ".");"

<print_args>: Defines exactly how strings and real expressions may appear in a print statement. The comma-separated approach is flexible yet remains consistent, improving readability and maintainability of code that prints multiple values. In Realang, string literals are only allowed here.

2.f) Function Definitions and Return

<function_def>: Specifies how functions are declared. Functions start with the keyword "function," then an identifier (the function's name), an optional parameter list enclosed in parantheses, and end in a <block>.

<parameter_list_opt> & <parameter_list> & <parameter>: A function may have zero or more parameters with type of "real." Using a comma-separated list of identifiers

ensures clarity in how arguments are named and passed. Formal parameters indicated with “real &” allows pass-by-reference.

<return_stmt>: Ensures all functions return a numeric result via “return” <real_expr> “;”, reflecting the numeric theme of Realang. This unifies the language model since everything ultimately revolves around real expressions.

2.g) Data Structure to Store Real Numbers and Its Operations

<list_dec>: Declares a new data structure using the syntax: "list" <identifier> "=" "{" <real_list_opt> "}" ";". This allows for the creation of an empty list or a list initialized with one or more real expressions, such as, “list A = {1, 2, 3};” or “list B = {x+y, 45, foo(5)};”

<real_list_opt> & <real_list>: Handle optional or multiple real expressions separated by commas, letting lists be empty or populated. This approach allows easy definition of numeric data collections.

<list_func>: Realang provides built-in methods .add(<real_expr>) and .remove(<real_expr>) for operations on lists (adding or removing elements). It improves writability by making list operations straightforward and reliable in behavior. These operations can be used as “A.add(3.14);” or “B.remove(2);”

2.h) Comments

<comment>: The grammar allows both single-line comments and multi-line comments. Single-line comments begin with // and continue to the newline, while the multi-line comments are enclosed within /* and */. Explanations via comments greatly improves readability, maintainability, and overall reliability of code.

3. Logical Expressions

<logical_expr>: This nonterminal defines boolean logic but constructed from real numbers. It uses logical OR operator ||. It can be a single <logical_term> or a combination of <logical_expr> with ||. The aim is to maintain operation presedence.

<logical_term>: Used to combine logical factors with the logical AND operator &&. By defining it separately, the grammar of the Realang assigns higher presedence to logical AND (&&) over logical OR (||).

<logical_factor>: This is the basic unit of logical expressions. It may be logical NOT (!), parenthesized subexpressions, or comparisons between two real expressions.

<comparison_op>: Specifies the relational operators. Used to compare two real expressions, these yield a logical result.

4. Real Number Expressions

<real_expr>: Defines arithmetic expressions that evaluate to real numbers. It is built from <term> and allows for addition (+) and subtraction (-) operations. The recursive definition ensures that arithmetic expressions follow the standard order of arithmetic hierarchy.

<term>: Component of <real_expr> that handles multiplication (*) , division (/) , and modulo (%). It is defined recursively to ensure that these operations take precedence over addition and subtraction.

<factor_term>: Defines exponentiation (^) within real number expressions. This operation is right associative, meaning an expression like a^b^c is interpreted as a^(b^c).

<factor>: Represents the smallest unit of real number expressions. It may consist of a <primary> optionally preceded by an unary operator (+ or -). Realang language allows only one unary operator per <factor> to prevent ambiguous expressions like --x.

<primary>: Basic component of the real number expressions. It can be a literal real number, an identifier, a parenthesized real expression, and a function call. It can also be some built-in list expressions resulting in real numbers, such as, “A.getElement(2)” or “A.getSize()” All these forms yield a real number, which fits the language’s design that restricts assignments and expressions to real numbers.

5. Function Calls and Built-In Data Structure Methods

<function_call>: Represents the any invocation of a function, returns a real number. Realang disallows standalone function call statements because every function returns a numeric value and is thus expected to use in an expression context.

<argument_list_opt> & <argument_list> & <argument>: Allows zero or more real expressions in the argument list. “&” symbol enables pass-by-reference function calls.

<list_expr>: Allows operations that retrieve data from a list. It has two forms: one to get a real number at a given index with .getRealNum(<index_expr>), and one to get the size of the list with .getSize(). These calls return real numbers, making them valid <primary> components in further arithmetic expressions.

<index_expr> & <index_term> & <index_factor>: Define arithmetic operations for list indexing. <index_term> allows addition (+) and subtraction (-), ensuring they have lower precedence, while <index_factor> handles multiplication (*), division (/), and modulo (%), which are evaluated first.

<index_primary>: Represents the smallest unit of an index expression. It can be an integer (INT), a variable (IDENTIFIER), or a parenthesized expression ((i + 2) * 3).

6. Lexical Definitions

REAL_NUM: Defines the format for numeric literals. A real number can either consist of a sequence of digits followed by an optional fractional part (1, 2., 1.2) or start with a dot followed by digits (.1). This accomodates both standard and fractional notation.

INT: Represents one or more digits.

IDENTIFIER: Begins with either a letter or an underscore, followed by letters, digits, or underscores.

STRING: Defines string literals, which are enclosed in double quotes. In Realang, strings are only allowed to be used in the output/print statements.

Precedences and Associativity

1. Expressions within parantheses have the highest precedence, ensuring that any expression enclosed is evaluated as a unit.

2. Unary plus (+), unary minus (-), and logical NOT (!) are applied directly to a single operand. These operators bind tightly to their operands.

3. Exponentiation (^) operator is computed next (right-associative). Then, multiplication (*), division (/), and modulo (%) operators are evaluated, and they are left-associative.

4. Addition (+) and subtraction (-) are also left-associative. They are evaluated after multiplication, division, modulo, and exponentiation.

5. Comparison operators are used to compare real expressions. When real expressions are used within logical expressions, they yield a boolean context value (i.e., zero is interpreted as false).

6. Logical AND (&&) is evaluated after comparisons. It has higher precedence than logical OR (||), ensuring that multiple conditions joined by "&&" are grouped correctly.

7. Logical OR (||) has the lowest precedence among the operators. It is evaluated after all the above operations, allowing it to combine complete boolean expressions.

Conflicts

*** All parsing conflicts have been successfully resolved, ensuring that the grammar is unambiguous and follows the correct precedence and associativity rules.

Nontrivial Tokens in Realang

1. Comments

Definition: Realang supports two comment styles: single-line (//...) and mutli-line (/*...*/). Single-line comments run until the end of the current line, while multi-line comments span everything between /* and */.

Usage:

```
// This is a single-line comment

/* This is a

    multi-line comment */
```

Motivation: Comments allow programmers to annotate their code, improving clarity and making it easier for others to understand the logic.

Constraints: Multi-line comments cannot be nested, otherwise, parsing complexity would rise.

Relation to Language Criteria:

-Readability: Well-placed comments increase code clarity, helping future maintainers grasp the logic.

-Writability: Familiar C-like comment syntax is straightforward.

-Reliability: Properly delimited comments avoid unexpected parsing issues, it is clear that comment text is non-executable.

2. Identifiers

Definition: An identifier in Realang must begin with a letter (a-z, A-Z) or underscore(_), followed by zero or more letters, digits (0-9), or underscores. Identifiers cannot be reserved words.

Usage:

```
myVariable = 42;

_temp1 = myVariable + 5;
```

Motivation: Distinguishes user-defined names (variables, functions) from numeric literals and reserved words.

Constraints: Cannot start with a digit or be a reserved word, preventing ambiguity.

Relation to Language Criteria:

-Readability: Clear, consistent naming rules help readers to quickly identify variable and function names.

-Writability: Flexible use of letters, digits, underscore lets programmers create meaningful names.

-Reliability: Restricting identifiers from starting with digits or matching reserved words minimizes potential parsing conflicts.

3. Literals

3.a) Real Number Literals

Definition: Real numbers can be written as a sequence of digits (optionally followed by a fractional part) or a dot followed by digits. Examples: 42, 3.14, 2., .5,

Usage:

```
x = 2.;
z = 3.14;
```

Motivation: Realang focuses on real (numeric) computations. Allowing deciaml and leading-dot forms provides flexibility.

Constraints: Must not conflict with identifiers; numbers cannot begin with a letter. The grammar must unambiguously parse forms like 2., .2, 3.14 etc.

Relation to Language Criteria:

-Readability: Familiar decimal notation is instantly recognizable, reducing confusion.

-Writability: One can easily write expressions like .5 or 2., matrching typical math usage.

-Reliability: Strict grammar for numeric literals prevents parsing ambiguity between numbers and other tokens.

3.b) String Literals

Definition: Strings are enclosed in double quotes "…". They can contain letters, digits, symbols or be empty as well.

Usage:

```
print("Hello World!");
print("Value of x is", x);
```

Motivation: Provides a straightforward way to embed text for output operations. Double-quote delimiters are standard approach, making them easy to parse.

Constraints: Strings are restricted to certain contexts, such as output statements, aligning with Realang's numeric focus. Must not contain unescaped internal double quotes unless the language or tooling has special rules.

Relation to Language Criteria:

-Readability: Clear delimiters (double quoting) for string literals separate them from numeric expressions.

-Writability: Quoted strings are a common convention, making it easy to incorporate text.

-Reliability: Strict quoting rules prevent text from being misread as code.

4. Operators

Definition: Realang supports arithmetic operators (+, -, *, /, %, ^), comparison operators (==, !=, <, <=, >, >=), logical operators (&&, ||, !), and assignment (=). Additionally, certain list-related operations like .add(), .remove(), .getRealNum(), and. getSize() act similarly to method calls.

Usage:

```
x = 3 ^ 2;

if( x > 5 && x < 10) {

        print(x, " is between 5 and 10.");

}

myList.add(x);
```

Motivation: Operators must cover typical arithmetic, logical, and assignments while maintaining unambiguous precedence. Built-in list methods align with typical object/method call patterns (like A.add(3.14);).

Constraints: Must define clear precedence rules, e.g. multiplication/divison/modulo > addition/subtraction. Must limit assignment (=) to real expressions only.

Relation to Language Criteria:

-Readability: Familiar operator symbols are used to quickly parse expressions.

-Writability: Standard precedence and associativity arithmetic rules reduce confusion.

-Reliability: Consistent operator behavior prevents subtle bugs and clarifies code intent.

5. Reserved Words

Definition: Reserved words are keywords like "start, end, if, else, while, for, function, list, return, print, input, etc." They cannot be used as identifiers.

Usage:

```
start
    list myNumbers = {1, 2.5, .3};
    if (myNumbers.getSize() > 2) {
        print("List has more than two elements.");
    }
end
```

Motivation: Marking certain words as reserved ensures they cannot be used as identifiers. Makes Realang's control structures (if, else) and built-in features (print, list) immediately recognizable to both the parser and the user.

Constraints: Reserved words must be distinct. Must not overlap with potential identifiers.

Relation to Language Criteria:

-Readability: Reserved words give immediate clues about code structure. For instance, "for" means a loop and "if" means a conditional statement.

-Writability: Resuing common terms from popular languages reduces the learning curve.

-Reliability: Preventing these keywords from being redefined as variables or functions avoids parsing errors.

Evaluation of the Language

Readability

Realang's syntax was kept minimal and straightforward, making it easy to read and understand. The language uses **curly braces** {} to delimit blocks, and **explicit start and end tokens** to define program boundaries, ensuring a well-structured and easily recognizable format. The use of **a small set of well-known keywords** (e.g., if, else, while, for, function) makes the language familiar.

To further improve readability, Realang supports **both single-line (// ...) and multi-line (/* ... */)** comments, allowing one to document their code effectively. The language's **numeric-only variable** restriction eliminates the need for type declarations, making the code visually cleaner and avoiding confusion related to variable types. Additionally, the clear separation of statements and well-defined nontrivial tokens (real numbers, strings, identifiers) reduce parsing errors.

Writability

By restricting variables **only to real numbers**, Realang doesn't require the need for explicit type declarations except the keyword **"real,"** simplifying the programming model. The language consists of familiar constructs for both control flow (if, while, for) and I/O operations (input, print).

Built-in **list operations** (.add(...), .remove(...), .getRealNum(...), .getSize()) follow an intuitive **method-call style**, making them easy to write and remember. The decision to keep function calls solely within real number expressions ensures consistency with Realang's numeric focus.

Reliability

Realang is designed to ensure predictable and consistent behavior. It enforces **matched/unmatched if-statements**, removing the classic "dangling-else" problem, which cause confusion. In addition, **operator precedence** is well-defined, ensuring that arithmetic and logical operations evaluate correctly without ambiguity.

Strict **real numbers-only assignments** prevent implicit type conversions, eliminating unexpected type errors. Moreover, **reserved words cannot be used as identifiers**, avoiding conflicts with core language constructs. The **lexical definitons** for numbers, identifiers, and keywords reduce the likelihood of parser errors. These design choices together make Realang more stable, reliable, and easier to implement.