Bilkent University
Department of Computer Engineering

CS315 Project Report

Programming Language: El-Drone

Fall 2024-2025

Team Members:

Ahmet Furkan Kızıl 22203112 Section 3

Buğra Çayır 22202461 Section 3

Emirhan Kılıç 22203360 Section 3

Table of Contents

1. BNF of the Language

```
<program> → <stmt_list>
<stmt_list> → <stmt_list> <stmt> | <stmt>

<stmt> → <assign_stmt> | <if_stmt> | <for_stmt> | <while_stmt> | <function_stmt>
      | <return_stmt> | <expr_stmt> | <operation_stmt> | <drone_stmt>
      | <print_stmt> | <comment_stmt> | <multicomment_stmt> | <break_stmt>
      | <import_stmt>

<assign_stmt> → <assign> NEW_LINE | <expr_stmt> NEW_LINE
<assign> → <string_assign> | <num_assign> | <logic_assign>

<string_assign> → IDENTIFIER ASSIGN <string_expr>
<num_assign> → IDENTIFIER ASSIGN <num_expr> | <num_short_assign>
<logic_assign> → IDENTIFIER ASSIGN <logic_expr>

<num_short_assign> → IDENTIFIER ADD_ASSIGN <num_expr>
              | IDENTIFIER SUB_ASSIGN <num_expr>
              | IDENTIFIER MUL_ASSIGN <num_expr>
              | IDENTIFIER DIV_ASSIGN <num_expr>

<if_stmt> → IF LP <logics> RP CLB <stmt_list> RCB
        | IF LP <logics> RP CLB <stmt_list> RCB ELSE CLB <stmt_list> RCB
        | IF LP <logics> RP CLB <stmt_list> RCB <else_if_seq> ELSE CLB <stmt_list> RCB
        | IF LP <logics> RP CLB <stmt_list> RCB <else_if_seq>

<else_if_seq> → <else_if> | <else_if_seq> <else_if>
<else_if> → ELSE_IF LP <logics> RP CLB <stmt_list> RCB

<while_stmt> → WHILE LP <logics> RP CLB <stmt_list> RCB

<for_stmt> → FOR LP <expr> COMMA <logics> COMMA <num_short_assign> RP CLB

<stmt_list> RCB
        | FOR LP <expr> COMMA <logics> COMMA <short_operation> RP CLB <stmt_list> RCB

<function_stmt> → FUNCTION IDENTIFIER LP <argument_list> RP CLB <stmt_list> RCB
            | FUNCTION IDENTIFIER LP RP CLB <stmt_list> RCB

<return_stmt> → RETURN <expr_stmt>

<argument> → STRING | INTEGER | IDENTIFIER
<argument_list> → <argument> | <argument_list> COMMA <argument>

<expr_stmt> → <expr> NEW_LINE
<expr> → <num_expr> | <string_expr> | <logic_expr>
```

```
<num_expr> → <additive_expr>

<additive_expr> → <multiplicative_expr>
            | <additive_expr> PLUS_OP <multiplicative_expr>
            | <additive_expr> MINUS_OP <multiplicative_expr>

<multiplicative_expr> → <primary_expr>
                | <multiplicative_expr> MULT_OP <primary_expr>
                | <multiplicative_expr> DIV_OP <primary_expr>
                | <multiplicative_expr> MOD_OP <primary_expr>

<primary_expr> → INTEGER | IDENTIFIER | LP <additive_expr> RP | <drone_attrb>

<drone_attrb> → GET_HEADING | GET_ALTITUDE | GET_TIME

<string_expr> → STRING
          | FUNCTION IDENTIFIER LP RP
          | FUNCTION IDENTIFIER LP <argument_list> RP
          | <string_expr> PLUS_OP <string_expr>
          | LP <string_expr> RP

<logics> → TRUE | FALSE | IDENTIFIER <comparators> <expr> | NON_OP LP <logics> RP
<logicals> → <logics> | <logicals> OR <logics>  | <logicals> AND <logics>
<logic_expr> → IDENTIFIER ASSIGN <logicals>
<operation_stmt> → <operation> NEW_LINE
<operation> → <expr> <operators> <expr>
<short_operation> → IDENTIFIER INCREMENT | IDENTIFIER DECREMENT
<operators> → PLUS_OP | MINUS_OP | MULT_OP | DIV_OP | MOD_OP | NON_OP
<comparators> → EQUAL | NOT_EQUAL | GT | GTE | ST | STE

<drone_stmt> → <drone_attrb> | <drone_moves> NEW_LINE
<drone_moves> → MOVE_FORWARD | DESCEND | ASCEND | TURN_LEFT | TURN_RIGHT
          | NOZZLE_ON | NOZZLE_OFF | WAIT | STOP
          | CONNECT_DRONE LP STRING COMMA INTEGER RP

<print_stmt> → PRINT LP <printable> RP NEW_LINE

<printable> → <expr> | <printable> COMMA <expr>

<break_stmt> → BREAK NEW_LINE

<import_stmt> → IMPORT IDENTIFIER <import_alias> NEW_LINE
          | FROM IDENTIFIER IMPORT IDENTIFIER <import_alias> NEW_LINE

<import_alias> → AS IDENTIFIER

<comment_stmt> → COMMENT NEW_LINE
<multicomment_stmt> → MULTI_COMMENT NEW_LINE
```

2. BNF Description

1. Program Structure

- <program> → <stmt_list>
    - The program starts with a list of statements that are the fundamentals of the language. This allows the program to consist of multiple statements which are executed sequentially one by one to meet the needs of the user.

2. Statements

- <stmt_list> → <stmt_list> <stmt> | <stmt>
    - A statement list can be either a single statement or a list of multiple statements. This is designed as left-recursive.
- <stmt> → <assign_stmt> | <if_stmt> | <for_stmt> | <while_stmt> | <function_stmt> | <return_stmt> | <expr_stmt> | <operation_stmt> | <drone_stmt> | <print_stmt> | <comment_stmt> | <multicomment_stmt> | <break_stmt> | <import_stmt>
    - A statement can be one of those: loops, conditional if statement, print commands, expressions, drone-specific commands, comments, break, or import statements.

3. Assignments

- <assign_stmt> → <assign> NEW_LINE | <expr_stmt> NEW_LINE
    - Assignment or expression statements end with a newline.
- <assign> → <string_assign> | <num_assign> | <logic_assign>
    - Assignments can involve either strings, numbers or logic assignments.
- <string_assign> → IDENTIFIER ASSIGN <string_expr>
    - String assignments assign a string expression to an identifier.
- <num_assign> → IDENTIFIER ASSIGN <num_expr> | <num_short_assign>
    - Number assignments assign a numeric expression or use shorthand operations like +=, -=, *=, /= as shown in the below.
- <num_short_assign> → IDENTIFIER ADD_ASSIGN <num_expr>

  | IDENTIFIER SUB_ASSIGN <num_expr>

  | IDENTIFIER MUL_ASSIGN <num_expr>

  | IDENTIFIER DIV_ASSIGN <num_expr>

4. Conditional Statements

- <if_stmt> → IF LP <logics> RP CLB <stmt_list> RCB
  | IF LP <logics> RP CLB <stmt_list> RCB ELSE CLB <stmt_list> RCB
  | IF LP <logics> RP CLB <stmt_list> RCB <else_if_seq> ELSE CLB <stmt_list> RCB
  | IF LP <logics> RP CLB <stmt_list> RCB <else_if_seq>
  - The "if" statement allows the user to write conditional logic. It supports "else" and "else if" clauses but they are optional. Also the case when the user ends the conditional statement with "if else" without an "else" is included. The conditions are expressed using logical expressions, and each block of the if, else, and if else statements are enclosed in braces.
- <else_if_seq> → <else_if> | <else_if_seq> <else_if>
- <else_if> → ELSE_IF LP <logic_expr> RP CLB <stmt> CRB

5. Loops

- <while_stmt> → WHILE LP <logic_expr> RP CLB <stmt> CRB
  - The while loop repeats the execution of the statement inside the braces as long as the logical expression is true.
- <for_stmt> → FOR LP <expr> COMMA <logic_expr> COMMA <num_short_assign> CP CLB <stmt> CRB | FOR LP <expr> COMMA <logic_expr> COMMA <short_operation> CP CLB <stmt> CRB
  - The for loop allows for iteration over a range, including an initialization via an expression, a condition of logical expression, an update at every iteration which might be num_short_assign or short_operation. The 3 elements of the for loop are separated by commas.

6. Functions

- <function_stmt> → FUNCTION IDENTIFIER LP <argument_list> RP CLB <stmt> CRB | FUNCTION IDENTIFIER LP RP CLB <stmt> CRB
  - Functions are defined using the FUNCTION keyword, followed by an identifier and a block of statements. Functions may or may not have arguments, so it is optional.
- <argument_list> → <argument> | <argument_list> COMMA <argument>
  - The argument list consists of one or more arguments separated by commas.

- <argument> → STRING | INTEGER | IDENTIFIER
    - Arguments are string, integer or identifiers

7. Expressions

- <expr_stmt> → <expr> NEW_LINE
    - Expression statements must end with a newline.
- <expr> → <num_expr> | <string_expr> | <logic_expr>
    - Expressions can be numeric, string-based or logic.
- <num_expr> → <additive_expr>
- <additive_expr> → <multiplicative_expr>
  | <additive_expr> PLUS_OP <multiplicative_expr>
  | <additive_expr> MINUS_OP <multiplicative_expr>

- <multiplicative_expr> → <primary_expr>
  | <multiplicative_expr> MULT_OP <primary_expr>
  | <multiplicative_expr> DIV_OP <primary_expr>
  | <multiplicative_expr> MOD_OP <primary_expr>
    - Numeric expressions involve combinations of addition, subtraction, multiplication, division, and modulo operations, structured hierarchically. Additive expressions can involve addition and subtraction, while multiplicative expressions handle multiplication, division, and modulo operations. These expressions can be nested to create complex calculations, with operations performed according to their precedence.
- <primary_expr> → INTEGER | IDENTIFIER | LP <additive_expr> RP | <drone_attrb>
- <drone_attrb> → GET_HEADING LP RP | GET_ALTITUDE LP RP | GET_TIME LP RP
    - Drone attributes allow retrieving the drone’s heading, altitude, or the current time.
- <string_expr> → STRING
  | FUNCTION IDENTIFIER LP RP
  | FUNCTION IDENTIFIER LP <argument_list> RP
  | <string_expr> PLUS_OP <string_expr>
  | LP <string_expr> RP

    - String expressions involve string literals, assignments, function calls concatenation via plus operation, or grouping using parentheses.

- <logics> → TRUE | FALSE | IDENTIFIER <comparators> <expr> | NON_OP LP <logics> RP

- <logic_expr> → IDENTIFIER ASSIGN <logics>
    - Logical expressions involve comparing two expressions using comparators or boolean expressions of TRUE and FALSE.. Operations
- <operation_stmt> → <operation> NEW_LINE
    - Operation statements consist of operations followed by a newline.
- <operation> → <expr> <operators> <expr>
    - Operations can involve two expressions combined with operators, or shorthand operations like increment and decrement.
- <short_operation> → IDENTIFIER INCREMENT | IDENTIFIER DECREMENT
    - Shorthand operations for increment and decrement on numeric expressions.
- <operators> → PLUS_OP | MINUS_OP | MULT_OP | DIV_OP | MOD_OP
    - Operators include addition, subtraction, multiplication, division, and modulo.
- <comparators> → EQUAL | NOT_EQUAL | GT | GTE | ST | STE
    - Comparators are used for equality and relational comparisons (==, !=, >, >=, <, <=).

8. Drone Statements

- <drone_stmt> → <drone_attrb> | <drone_moves> NEW_LINE
    - This rule is defined for the commands which are specific to control the drone. A drone statement can either involve retrieving a drone attribute, such as heading or altitude, or issuing a drone movement command.
- <drone_moves> → MOVE_FORWARD LP RP | DESCEND LP RP | ASCEND LP RP | TURN_LEFT LP RP | TURN_RIGHT LP RP | NOZZLE_ON LP RP | NOZZLE_OFF LP RP | WAIT LP RP | STOP LP RP | CONNECT_DRONE LP STRING COMMA NUMBER RP
    - The drone moves include commands to control the drone’s movement and functions. This covers moving ascending, descending, forward, turning left or right, controlling the nozzle for spraying, waiting, stopping, and establishing a

connection to the drone. The CONNECT_DRONE command also accepts a string argument to specify the connection parameters which are IP address and port number as can be seen at the end of the rule.

9. Print Statements

<print_stmt> → PRINT LP <printable> RP NEW_LINE

- The list of printable items is enclosed within parentheses (LP, RP), and the statement ends with a newline. It should start with the PRINT keyword.

<printable> → <expr> | <printable> COMMA <expr>

- Either a single expression or a list of expressions separated by commas can be printed. Expressions are either number or string; and they need to be given as input sequentially to avoid type mismatches.

10. Break Statements

<break_stmt> → BREAK NEW_LINE

- The break statement exit loops like for or while loops to immediately stop iteration when certain conditions are met.

11. Import Statements

- <import_stmt> → IMPORT IDENTIFIER <import_alias> NEW_LINE | FROM IDENTIFIER IMPORT IDENTIFIER <import_alias> NEW_LINE
- <import_alias> → AS IDENTIFIER
    - The import statement allows the user to bring in external modules or libraries into the current program. This supports four variations:
        - IMPORT IDENTIFIER: Imports a module by its identifier.
        - IMPORT IDENTIFIER AS IDENTIFIER: Imports a module and gives it an alias for use in the program.
        - FROM IDENTIFIER IMPORT IDENTIFIER: Imports a specific function or variable from a module.

        - FROM IDENTIFIER IMPORT IDENTIFIER AS IDENTIFIER: Imports a specific function or variable from a module and assigns it an alias.

12. Comments Statement

- <comment_stmt> → COMMENT NEW_LINE
    - Comments start with a # symbol and continue to the end of the line. This allows users to include annotations that do not affect program execution.

3. Variables and Terminals

3.1 Variables

letter: Matches any uppercase or lowercase letter (A-Z, a-z).

symbols: Matches underscore (_), typically used in identifiers.

zero: Matches the digit zero (0).

nonZeroDigit: Matches any digit from 1 to 9.

digit: Matches a single digit (0-9), including zero and non-zero digits.

sign: Matches a positive (+) or negative (-) sign.

integer: Matches an integer number, possibly with a sign.

float: Matches a floating-point number, possibly with a sign.

number: Matches either an integer or floating-point number.

alphanumeric: Matches a sequence of letters and digits, used in identifier matching.

identifier: Matches a valid identifier, which can consist of letters, digits, and symbols.

comment: Matches a comment starting with "#" and continuing to the end of the line.

stringSingleQuote: Matches a string enclosed in single quotes.

stringDoubleQuote: Matches a string enclosed in double quotes.

string: Matches either single-quoted or double-quoted strings

3.2 Terminals

ASSIGN: Represents the assignment operator =.

EQUAL: Represents the equality comparison operator ==.

NOT_EQUAL: Represents the not-equal comparison operator !=.

GT: Represents the greater-than operator >.

GTE: Represents the greater-than-or-equal operator >=.

ST: Represents the less-than operator <.

STE: Represents the less-than-or-equal operator <=.

PLUS_OP: Represents the addition operator +.

MINUS_OP: Represents the subtraction operator -.

MULT_OP: Represents the multiplication operator *.

DIV_OP: Represents the division operator /.

MOD_OP: Represents the modulo operator %.

NON_OP: Represents the logical not operator !.

ADD_ASSIGN: Represents the addition assignment operator +=.

SUB_ASSIGN: Represents the subtraction assignment operator -=.

MUL_ASSIGN: Represents the multiplication assignment operator *=.

DIV_ASSIGN: Represents the division assignment operator /=.

LP: Represents the left parenthesis (.

RP: Represents the right parenthesis).

CLB: Represents the left curly brace {.

RCB: Represents the right curly brace }.

COMMA: Represents the comma ,.

COMMENT: Represents a comment token #.

MULTI_COMMENT: Represents multiple comments in between 2 ##'s

NEW_LINE: Represents a new line in the input \n.

4. Non-Trivial Tokens

4.1 Comments

To make a line comment in this language, there should be "#" in the beginning of the line. Until a new line character appears, everything counts in the comment block. The comment feature is crucial to make a programming language readable because anybody who reads the code can directly understand what it does thanks to the comment lines that explains the code in human-understandable language. This makes it also writable because the log-related code lines used while debugging can be left as comment lines which eliminates the burden of constantly writing and deleting the same code.

4.2 Identifiers

identifier: Matches a valid identifier, consisting of letters, digits, symbols including underscores and hyphens, and alphanumeric characters.

4.3 Literals

string: Matches single-quoted or double-quoted strings.
INTEGER: Represents an integer value token.
FLOAT: Represents a floating-point value token.
IDENTIFIER: Represents an identifier token.
TRUE: Boolean literal representing a true value.
FALSE: Boolean literal representing a false value.

4.4 Reserved Words

For readability, writability, and reliability criteria, the reserved words are chosen as simple and understandable as possible.
getHEADING: Retrieves the current heading (direction) of the drone.
getALTITUDE: Retrieves the current altitude of the drone.
getTIME: Retrieves the current time in epoch.
moveFORWARD: Moves the drone forward.
DESCEND: Lowers the altitude of the drone.
ASCEND: Increases the altitude of the drone.
turnLEFT: Turns the drone to the left.
turnRIGHT: Turns the drone to the right.
WAIT: Pauses the drone's actions for a specified duration.
STOP: Stops the drone's movement or operations.
nozzleON: Activates the drone's nozzle (e.g., for spraying or releasing materials).
nozzleOFF: Deactivates the drone's nozzle.
connectDRONE: Establishes a connection to the drone.
function: Declares a new function.

print: Outputs information to the console or display.

input: Receives input from the user.

return: Returns a value from a function.

break: Exits a loop prematurely.

import: Imports a module or library.

as: Renames an imported module for local use.

from: Specifies the module from which to import functions or variables.

if: Begins a conditional statement.

"else if": Specifies an additional condition after an if statement.

else: Specifies a block of code to run if no preceding conditions are met.

while: Starts a loop that runs as long as a condition is true.

for: Begins a loop that iterates over a range or collection.

and: Logical operator used to combine two conditions.

or: Logical operator used to check if at least one of multiple conditions is true.

TRUE: Boolean literal representing a true value.

FALSE: Boolean literal representing a false value

5. Evaluation of the Language

5.1 Readability

Since users of the language may not be professional programmers, the language should be easy to learn and comprehend. For this reason, the reserved words are designed to be intuitive and closely resemble the common English words. The commands like moveFORWARD, turnLEFT are self-explanatory so that the drone user can achieve his needs accordingly. Minimal number of symbols are used while designing the language. The use of such operators like "&" are avoided and instead direct English words like "and" are used. Moreover, using the # symbol for comments ensures that the explanations can be added directly into the code without interfering with the logic, which increases readability.

5.2 Writability

Built-in primitive functions for drone control are included in the language so that the programmers can easily express their ideas in the language. Using "print" and "input" for basic input/output operations allows users to easily interact with the language, which makes it easy to use for non-expert programmers too. Additionally, arithmetic and boolean expressions are added to the language to increase the flexibility when writing complex conditions, loops, or calculations. The language allows identifiers that include letters, digits, underscores, and hyphens, which makes it easier to name variables meaningfully and descriptively. Lastly, the standard constructs like if-else blocks, "while" and "or" loops make the language familiar to users who already know about other programming languages.

5.3 Reliability

The language is designed to prevent errors and ensure that programs behave as expected. The literals like INTEGER, or BOOLEAN are strictly defined to enforce type safety via reducing the potential errors arising from mismatches during operations. Standard comparison operators are included to make a clear distinction between other operators just like in the example of assignment (=) and equality (==) operators. Functions like getHEADING return predefined results which ensures that users can rely on these functions to behave consistently. This is crucial for controlling real-world systems. Since the users can easily add comments, the debugging and maintenance processes make it easier to understand the intent behind specific pieces of code when revisiting the project.

5.4 Conflicts:

In the grammar of El-Drone, the shift/reduce conflicts occurred because the parser was unsure about how to handle operator precedence operators. Specifically, it cannot decide whether to read the next token (shift) or combine parts of the expression (reduce) when it encounters operators like addition, subtraction, multiplication, division, and modulo. This operator precedence conflict occurred in the grammar during the design of the BNF of programming language, the numerical expressions are written with the focus of ensuring maintainability with the other expressions. In an ideal programming language, operator precedence is detailly maintained to operate addition, subtraction, multiplication, division, and modulo with their respective priorities.