CS 315

Project 1

Maryam Shahid - 21801344 - Section 1

Hammad Khan Musakhel - 21801175 - Section 3

Şükrü Can Erçoban - 21601437- Section 1

Language Name: Drone-X

Drone-X has been constructed using a set of guidelines that add to its readability, writability and reliability. The scope of the project is to construct a programming language for drones. The general components of **Drone-X** use conventional declarations such as semicolon to end a statement, logical expressions for loops and condition statements, and brackets to mark the start and end of loops and conditions; ultimately making it more writable, readable, reliable, and easy to understand. Since the handling of drones does not require multiple data types, they are not declared or used in variables. To handle precedence, the lowest declared rules have higher precedence. The language supports loops and conditional statements (also nested conditional statements). It also has built-in primitive functions, adding to its reliability, which the user can use to retrieve data regarding the drone and pass commands to excite actions such as take picture, turn on/off video, read temperature, read altitude, read time stamp, etc. The drone can be connected to the base computer through wifi. It does so by reading the computer's IP address and establishing a connection. The BNF and its description are given below.

BNF

1. Program Definition

<program> ::= <main>

<main> ::= <LP> <RP> <LB> <statements> <RB>

<statements> ::= <stmt>
| <statements> <stmt>

<stmt> ::= <loops>
| <cond_stmt>
| <comments>
| <expr><end_stmt>
| <connection_stmt> <end_stmt>
| <function_def><end_stmt>
| <function_dec_call><end_stmt>
| <return_st><end_stmt>
| <assign_st>< end_stmt>
| <print_st><end_stmt>

<comments> ::= # <stmt>
| #* <statements> ##

<end_stmt> ::= ";"

2. Loops

<loops> ::= <while_loop> | <for_loop>

<while_loop> ::= while <LP> (<logic_expr> | <funct_call>) <RP> <LB> <statements> <RB>

<for_loop> ::= for <LP> <for_expr><end_stmt> <RP> <LB> <statements> <RB>

3. Conditional Statements

<conditional_stmt> ::= <if_stmt>

<if_stmt> ::= if <LP> (<logic_expr> | <funct_call>) <RP>
| <matched_if>
| <unmatched_if>

<matched_if> ::= if <LP> <logic_expr> <RP> <matched_if> else <matched_if>
| if <LP> <logic_expr> <RP> <matched_if> elseif <matched_if>
| <LB><statements><RB>

<unmatched_if> ::= if <LP> <logic_expr> <RP> <matched_if> else <unmatched_if>
| if <LP> <logic_expr> <RP> <matched_if> elseif <matched_if>
else <unmatched_if>

4. Functions Definitions and Calls

<parameters> ::= <multi_parameters> , <var>
| <multi_parameters> , <const>
| <var>
| <const>
| <IP>

<IP> ::= <string> <dot> <string> <dot> <string> <dot> <string>

<receive_IP> ::= <var> <receive> <IP>

<function_def> ::= function <var> <LP> <parameters> <RP> <LB> <statements> <RB>

<funct_call> ::= <var> <LP> <parameters> <RP>

<connection_st> ::= <receive_IP>
| <var> <comparison_op> <IP> connect
| <var> <comparison_op> <IP> disconnect

<return_st> ::= return <var>
| return <expr>
| return <constant>

<functions> ::= readAltitude | readTemperature | readAcceleration | turnOnCamera
| turnOffCamera | takePicture | readTimestamp | connectToBase

5. Expressions

<assign_st> ::= <var> = <var>
| <var> = <expr>
|<var> = <logic_expr>
| <var> = <const>
| <var> = <boolean>
| <var> = <funct_call>
| <var> = <input>
| <var> = <functions>

<expr> ::= ::= <operand> <op> <LP> <expr_base> <RP>
| <operand> <op> <operand>
| <LP> <expr_base> <RP>
| < expr> <op> <operand> <op> <operand>
| < expr> <op> <LP> <expr_base> <RP>

<logic_expr> ::= < operand> <comparison_op> <operand>
| <LP> <logic_expr> <RP>
| <logic_expr> <boolean_op> <operand> <comparison_op> <operand>
| <logic_expr> <boolean_op> <LP> <operand> <comparison_op> <operand>
<RP>

<for_expr> ::= <var> <assignment_op> <integer> to <var> <end_stmt>
| <var> <assignment_op> <var> to <integer> <end_stmt>
| <var> <assignment_op> <var> to <var> <end_stmt>

<operand> ::= <var> | <const> | <funct_call> |<prim_functions>

<arithmetic_op> ::= + | - | * | / | %

<comparison_op>::= < | > | <= | >= | == | !=;

<boolean_op>::= && | || | !;

<op>::= <Boolean_op> | <arithmetic_op>

6. Input and Output Statements

<output> ::= print (<operand>)
| print (<expr>)
| print (<logic_expr>)
| print (<var>)
| print(<prim_functions>)
| print(<function_call>)

<input> ::= input()
| input (<parameters>)
| input (<multi_parameters>)

7. Variable Identifiers

<assignment_op> ::= “=”

<LB> ::= “{“

<RB> ::= “}”

<LP> ::= “(“

<RP> ::= “)”

<dot> ::= “.”

<newline> ::= \n

<var> ::= <char> | <char> <string>

<const> ::= ‘<string>’ | <integer>

<string> ::= <string> <char> | <string> <digit> | <char> | <digit> | < >

<integer> ::= <integer> <digit> | <digit>

<boolean> ::= true | false

<true> ::= true | 1

<false> ::= false | 0

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

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

Description

<program> ::= <main>

<main> ::= <LP> <statements> <RP>

The program starts with the declaration of main. It consists of statements which forms the body of the program to be written.

<statements> ::= <stmt>
| <stmt> <statements>

Statements can have multiple stmt attributes, allowing writability for the program and extending the scope of the program.

<stmt> ::= <loops>
| <cond_stmt>
| <comments>
| <expr><end_stmt>
| <connection_stmt> <end_stmt>
| <function_def>
| <function_call>
| <return_st> <end_stmt>
| <assign_st>< end_stmt>

Stmt includes loops, conditional statements, comments, expressions, function calls, function definitions and return statements. In essence, the majority of the statements/lines written in the program would be stmt.

<comments> ::= # <stmt>
| #* <statements> ##

Single line comment can initialized with '#' and multiple line comments can be initialized with '#*' and ended by '##'.

<end_stmt> ::= ";"

To keep track of the ending of an expression, we assign a semicolon to mark the end, making the language more readable. It is easy to know where the statement ends.

<expr> ::= ::= <operand> <op> <LP> <expr_base> <RP>
| <operand> <op> <operand>
| <LP> <expr> <RP>
| < expr> <op> <operand> <op> <operand>
| < expr> <op> <LP> <expr_base> <RP>

The expression statement deals with expressions such as variables being multiplied with a constant and there's an addition operator there as well, for example: var1 + B * 5 / 12. The Arithmetic operator precedence is enforced in the language. The expression could take any form for e.g. **<var> <arithmetic_op> <var>, or <var> <arithmetic_op> <const>, <expr><arithmetic_op> (<var><arithmetic_op><constant>), or much more to this.**

<logic_expr> ::= < operand> <comparison_op> <operand>
| <LP> <logic_expr> <RP>
| <logic_expr> <boolean_op> <operand> <comparison_op> <operand>
| <logic_expr> <boolean_op> <LP> <operand> <comparison_op> <operand>
<RP>

A logic expression can contain relational operators and logical operators and variables to relate with each other. The comparison operators that are being used in our language:

>: greater than
<: less than
>=: greater than or equals to
<=: less than or equals to
==: equals
!=: not equals

The expression can be in any form, such as (z > 2 and (x ==1 or y < 4)). This has been primarily done to provide a more subtle form of logical expressions for the conditional statements. This adds to the writability of the programming language as now one can have more than one relational operator and condition for a while/if statement to execute. The logical operators in our language are going to be "and" and "or" and "not." They are described as '&&', "!" and '||.' These are general components shared by many programming languages hence making the program simper to understand, learn, and write.

<for_expr> ::= <var> <assignment_op> <integer> to <var> <end_stmt>
| <var> <assignment_op> <var> to <integer> <end_stmt>
| <var> <assignment_op> <var> to <var> <end_stmt>

For expression allows the user to understand the condition for the loop to proceed and terminate. The for expression forms the part between the parents in the for loop. It has almost every

required condition/expression for the form loop in this programing language; for example: x = 1 to 9; x = a to b.

<loops> ::= <while_loop>
| <for_loop>

There are only two loops in this language, while and for

<while_loop> ::= while <LP> <logic_expr> <RP> <LB> <statements> <RB>

The while loop contains a logical expression and statements

<for_loop> ::= for <LP> <for_expr><end_smt> <RP> <LB> <statements> <RB>

The for loop contains a for expression and statements, a generic way to go with this extensively used loop; we have kept it generic to keep the user feel more easy to program this loop - adding to writability.

<conditional_stmt> ::= <if_stmt>

<if_stmt> ::= if <LP> <logic_expr> <RP>
| <matched_if>
| <unmatched_if>

This is used to define an if statement with a logical expression. The matched_if and unmatched_if are used to solve the dangling else problem in case of nested ifs.

<matched_if> ::= if <LB> <logic_expr> <RB> <matched_if> else <matched_if>
| if <LB> <logic_expr> <RB> <matched_if> elseif <matched_if>
| <statements>

This is also used to solve the problem of the dangling else, it is used to define a nested if which has not been matched with its else statement. The else if should not have a space in between.

<unmatched_if> ::= if <LB> <logic_expr> <RB> <matched_if> else <unmatched_if>
| if <LB> <logic_expr> <RB> <matched_if> elseif <matched_if>
else <unmatched_if>

This is used in solving the problem of the dangling else. It is used to define a nested if which has been matched with its else statement. This makes the language more writable as more and more nested loops can be used for better functions and catering to the needs of the user.

<parameters> ::= <multi_parameters> , <var>

| <multi_parameters> , <const>
| <var>
| <const>
| <IP>

The definition for parameters is to supply constants, variables, and IP into a function call. This increases the RELIABILITY of the language as now we won't be encountering parameters such as (, var), (, constant), etc.

<IP> ::= <string> <dot> <string> <dot> <string> <dot> <string>

This is the format of an IP address. It consists of a 32-bit number which is written as four numbers separated by periods. A specific definition of the IP makes it more readable and understandable, hence easy to learn and use. The user would know what the IP would be for.

<receive_IP> ::= <var> <receive> <IP>

This is the parameter of the function that connects the drone to a base computer. It receives the IP address of the base computer and assigns it to a separate variable

<function_def> ::= function <var> <LP> <parameters> <RP> <LB> <statements> <RB>
| function<var><LP> <multi_parameters><RP><LB><statements><RB>

The function definition names the function in a string and adds parameters in the parenthesis for the function call. It is followed by statements required.

<funct_call> ::= <var> <LP> <parameters> <RP>
| <var> <LP> <multi_parameters> <RP>

Calling the function with its name, and parameters in parenthesis followed by an end statement in the end, that is a semi colon.

<assign_st> ::= <var> = <var>
| <var> = <expr>
|<var> = <logic_expr>
| <var> = <const>
| <var> = <boolean>
| <var> = <funct_call>
| <var> = <input>
| <var> = <functions>

We have defined the assignment of an expression to a variable, assignment of a functional call to a variable, and the assignment of input to a variable, assignment of the boolean to the variable, assignment of the primitive functions to the variable, assignment of a constant to a variable in the assign statement. This fits easier in the definition of stmt and makes it more writable and reliable for the user adding to the diversity of the language's assigning options.

<connection_st> ::= <receive_IP>
| <var> <comparison_op> <IP> connect
| <var> <comparison_op> <IP> disconnect

Connection statement receives the IP address of the base computer and connects the computer to the drone using “connect”. It can be disconnected using “disconnect.”

<return_st> ::= return <var>
| return <expr>
| return <constant>

The return statement to return an expression, a constant, or a variable to a function call.

<functions> ::= readAltitude | readTemperature | readAcceleration | turnOnCamera | turnOffCamera | takePicture | readTimestamp | connectToBase

This is a terminal statement consisting of reserved words for functions. This is the data/command which we will get/give from our input pins. This data/command given to us is executed directly by calling these built-in functions

<input> ::= input()
| input (<parameters>)
| input (<multi_parameters>)

Input statements should get input from a user. The value is stored and can be used for further instructions. This adds to the writability of the language as the user can provide inputs as well to be stored, and much more.

<output> ::= print (<operand>)
| print (<expr>)
| print (<logic_expr>)
| print (<var>)
| print(<prim_functions>)
| print(<function_call>)

Forms the integral part of the language; showing the user values from different functions and allowing the user to also make use of this feature to edit the console the way he/she wants. This is a prime feature which uplifts the reliability and readability of the language. This also increases the reliability as the user can see results on the console for the prime functions, user declared functions, and also of variables.

<assignment_op> ::= “=”

In this language, ‘=‘ will be used as the assignment operator, that is assigning the contents of RHS to the variable on LHS of the operator.

<var> ::= <string>

The name of the variable is always a string

<const> ::= "<string>" | <integer>

In this language, constants can contain either strings or integers.

<arithmetic_op> ::=

+ : Addition

- : Subtraction

***: Multiplication**

/: Division

****: Power**

%: Modulus

The arithmetic operators used in our language. The precedence is enforced in the language.

<string> ::= <string> <char> | <string> <digit> | <char> | <digit>

The string can contain multiple integers and characters, or only digit or chars.

<integer> ::= <integer> <digit> | <digit>

An integer contains multiple digits

<boolean> ::= true | false

Boolean is either true or false in this language

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

Both the uppercase and lowercase alphabets are taken in as characters.

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

Digits range from 0 - 9

Handling Precedence and Ambiguity:

Precedence of arithmetic operators is enforced by in the yacc file; that it is not apparent in the BNF structure but is rather declared in the language's yacc file. Kindly refer to the lex and yacc file for the better understanding of the implementation of the program language's definitions and operations.
It won't be harmful to mention that we collectively worked to make the BNF more explicit in terms of operator precedence, however the facing of conflicts made it hard for us to proceed with that. We tried tring another form of expr and logic_expr but we received many conflicts and hence we had to change. We branched the expr into Operand Arithmetic Operators with defined precedence of arithmetic operators and a different branch of operandLogical where we defined the precedence of logical operators, that is ! highest followed by && and then ||. We grasped the idea that lower the rule, higher the precedence with ( ) having the highest above all. However, we tried our best yet faced many conflicts and hence had to use our previous definitions. An example of our revised expr and logic_expr that we chose to not use:

<expr> ::= <operandArithOperators> | <operandLogical>

<operandArithOperators>::= <term>
| <operandArithOperators> + <term>
| <operandArithOperators> - <term>

<term>::= <fact>
| <term> * <fact>
| <term> / <fact>

<fact>::= <operand>
| <LP> <operandArithOperators><RP>

<operandLogical>::= <terms>
| <operandLogical> || <terms>

<terms>::= <facts>
| <terms> && <facts>
| <terms> ! <facts>

<facts>::= <operand>
| <LP><operandLogical><RP>

<logic_expr> ::= <comparisonExpression> | <comparisonLogical>

<comparisonExpression>::= <operand> > <operand>
| <operand> < <operand>
| <operand> >= <operand>
| <operand> <= <operand>
| <operand> == <operand>

| <operand> != <operand> |

<comparisonLogical>::= <sigma>
| <comparisonLogical> || <sigma>

<sigma>::= <authentic>
| <sigma> && <authentic>
| <sigma> ! <authentic>

<authentic>::= <comparisonExpression>
|<LP><comparisonLogical><RP>