CS 315 Project 2 Report

Fall 2021

GEKRONE

Name,Surname	ID	Section
Gizem Bal	21601886	03
Khashayar Amini	21903613	03
Ezgi Lena Sönmez	21703799	03

1. GEKRONE

1.1. Program

<program> ::= <block_statements> | <statements>

1.2. Statements

```
<statements> ::= <statement> <statements>
               |  <statement>
<statement> ::= <if_statements>
              |  <assignment_statement>
              |  <loops>
              |  <function_call>
              |  <function_declaration>
              |  <comment>
```

<comment> ::= COMMENT

<assignment_statement> ::= IDENTIFIER ASSIGNMENT_OP
<additive_expressions> SEMI_COLON
| IDENTIFIER ASSIGNMENT_OP <conditional_expression>
SEMI_COLON
| IDENTIFIER ASSIGNMENT_OP <function_call>

<block_statements> ::= LBRACE RBRACE | LBRACE<statements>RBRACE |
LBRACE<block_statements>RBRACE

1.3. Variable Identifiers

<constant> ::= <numeric_constant> | BOOLEAN_CONSTANT

<numeric_constant> ::= INT | FLOAT

```
<identifier_list> ::= IDENTIFIER
                   |  <identifier_list> COMMA IDENTIFIER
```

1.4. Expressions (arithmetic, relational, boolean, their combination)

```
<additive_expressions>  ::= <additive_expressions> PLUS
<multiplicative_expressions>

                        |    <additive_expressions> MINUS
<multiplicative_expressions>

                        |    <multiplicative_expressions>

<multiplicative_expressions> ::= <multiplicative_expressions> MULTIPLY <term>

                             | <multiplicative_expressions> DIVIDE <term>

                             | <multiplicative_expressions> REMINDER <term>

                             | <term>

<term> ::= IDENTIFIER | <constant>

<equality_expression> ::= <additive_expressions> EQUALITY_CHECK
<additive_expressions>

                      | <additive_expressions> NOT_EQUAL
              <additive_expressions>

<relational_expression> ::= <additive_expressions> SMALLER
<additive_expressions>

                        | <additive_expressions> GREATER <additive_expressions>

                        | <additive_expressions> SMALLER_OR_EQUAL
              <additive_expressions>

                        | <additive_expressions> GREATER_OR_EQUAL
              <additive_expressions>

<conditional_expression> ::= <equality_expression>

                           | <relational_expression>

                           | <conditional_expression> AND <equality_expression>

                           | <conditional_expression> AND
<relational_expression>
```

| <conditional_expression> OR <equality_expression>

| <conditional_expression> OR <relational_expression>

<movement_expression> ::= AUTO COMMA <term>
| MANUAL COMMA <term> COMMA <term>

1.5. Loops

<loops> ::= WHILE LP <conditional_expression> RP <block_statements>

| DO <block_statements> WHILE LP <conditional_expression> RP SEMI_COLON

| FOR LP <term> RP <block_statements>

1.6. Conditional statements

<if_statement> ::= IF LP <conditional_expression> RP <block_statements>

<if_else_block> ::= ELSEIF LP <conditional_expression> RP <block_statements>

| <if_else_block> ELSEIF LP <conditional_expression> RP <block_statements>

<if_statements> ::= <if_statement>

| <if_statement> <if_else_block>

| <if_statement> <if_else_block> ELSE <block_statements>

| <if_statement> ELSE <block_statements>

1.7. Function definitions and function calls

<function_declaration> ::= GEK IDENTIFIER LP <identifier_list> RP <block_statements>

```
                                 | GEK IDENTIFIER LP RP <block_statements>

<function_call> ::= IDENTIFIER LP RP SEMI_COLON
                  | IDENTIFIER LP <identifier_list> RP SEMI_COLON
                  | <primitive_functions> SEMI_COLON
```

1.8. Primitive functions

```
<primitive_functions> ::= READ_ALTITUDE LP RP
                        | READ_TEMPERATURE LP RP
                        | READ_DIRECTION LP RP
                        | READ_TANK LP RP
                        | READ_LATITUDE LP RP
                        | READ_LONGITUDE LP RP
                        | READ_BATTERY LP RP
                        | CONNECT_TO_DRONE LP RP
                        | NOZZLE LP <term> RP
                        | VERTICAL_MOVEMENT LP <movement_expression> RP
                        | HORIZONTAL_MOVEMENT LP <movement_expression>
RP
                        | TURN LP <movement_expression> RP
                        | SET_HFS  LP<term> RP
                        | SET_HBS LP <term> RP
                        | SET_VUS LP <term> RP
                        | SET_VDS LP <term> RP
                        | SET_TS LP <term> RP
```

2. Description of language constructs

2.1. Program

<program> ::= <block_statements> | <statements>

Program is either block_statement or a list of statements.

2.2. Statements

<statements> ::= <statement> <statements>
| <statement>

This is the recursive construct of a list of statements.

<statement> ::= <if_statements>
| <assignment_statement>
| <loops>
| <function_call>
| <function_declaration>
| <comment>

A statement can be either an if statement, assignment statement, a loop, a function call, function declaration or a comment.

<assignment_statement> ::= IDENTIFIER ASSIGNMENT_OP <additive_expressions> SEMI_COLON
| IDENTIFIER ASSIGNMENT_OP <conditional_expression> SEMI_COLON
| IDENTIFIER ASSIGNMENT_OP <function_call>

This is the description of an assignment statement. The r value can either be an expression or a function call. In case of a function call, the value returned by the function will be used. There is no need to define a type in this language. Because all variables are either integer or float, the compiler will decide which type the variable will be.

<block_statements> ::= LBRACE RBRACE | LBRACE<statements>RBRACE | LBRACE<block_statements>RBRACE

This is the description of a block statement.

2.3. Comment

<comment> ::= COMMENT

Users can write comments. comments will be written between /## and #/ .

2.4. Variable Identifiers

<constant> ::= <numeric_constant> | BOOLEAN_CONSTANT

In our language, a constant will be either a numeric constant or a boolean.

<numeric_constant> ::= INT | FLOAT

A numeric constant is either an integer or a floating point number.

<identifier_list> ::= IDENTIFIER
| <identifier_list> COMMA IDENTIFIER

Identifier list is a list of identifiers separated by comma.

2.5. Expressions (arithmetic, relational, boolean, their combination)

<additive_expressions> ::= <additive_expressions> PLUS <multiplicative_expressions>

| <additive_expressions> MINUS <multiplicative_expressions>

| <multiplicative_expressions>

Additive expression is addition or subtraction of multiplicative expressions or a multiplicative expression on its own.

<multiplicative_expressions> ::= <multiplicative_expressions> MULTIPLY <term>

| <multiplicative_expressions> DIVIDE <term>

| <multiplicative_expressions> REMINDER <term>

| <term>

Multiplicative expression is multiplication, division or reminder of terms or a term on its own.

<term> ::= IDENTIFIER | <constant>

Term is either a variable identifier or a constant.

<equality_expression> ::= <additive_expressions> EQUALITY_CHECK <additive_expressions>

| <additive_expressions> NOT_EQUAL <additive_expressions>

Equality expression will check whether two additive expressions are equal or not.

<relational_expression> ::= <additive_expressions> SMALLER <additive_expressions>

| <additive_expressions> GREATER <additive_expressions>

| <additive_expressions> SMALLER_OR_EQUAL <additive_expressions>

| <additive_expressions> GREATER_OR_EQUAL <additive_expressions>

Relational expression checks the relation between two additive expressions.

<conditional_expression> ::= <equality_expression>

| <relational_expression>

| <conditional_expression> AND <equality_expression>

| <conditional_expression> AND <relational_expression>

| <conditional_expression> OR <equality_expression>

| <conditional_expression> OR <relational_expression>

Conditional expression is a combination of different boolean type expressions which will be used for writing if and while expressions.

<movement_expression> ::= AUTO COMMA <term>
| MANUAL COMMA <term> COMMA <term>

Movement expression will express how a movement is going to take place. Movement can be either manual or automatic, chosen by movement expression.

2.6. Loops

<loops> ::= WHILE LP <conditional_expression> RP <block_statements>

| DO <block_statements> WHILE LP <conditional_expression> RP SEMI_COLON

| FOR LP <term> RP <block_statements>

There are three types of loops in GEKRONE which are while loop ,do while loop , and for loop.

2.7. Conditional statements

<if_statement> ::= IF LP <conditional_expression> RP <block_statements>

If statement without any else or elseif statements.

<if_else_block> ::= ELSEIF LP <conditional_expression> RP <block_statements>

| <if_else_block> ELSEIF LP <conditional_expression> RP <block_statements>

One or more elseif statements.

<if_statements> ::= <if_statement>

| <if_statement> <if_else_block>

| <if_statement> <if_else_block> ELSE <block_statements>

| <if_statement> ELSE <block_statements>

If statements can either be a single if, one if followed by one or more elseifs and one optional else statement at the end.

2.8. Function definitions and function calls

<function_declaration> ::= GEK IDENTIFIER LP <identifier_list> RP <block_statements>
| GEK IDENTIFIER LP RP <block_statements>

This is how a function can be declared in our language. By using the keyword gek, the user can enter the name of the function, its parameters and what the function will do. a function can exist both with or without any arguments.

<function_call> ::= IDENTIFIER LP RP SEMI_COLON
| IDENTIFIER LP <identifier_list> RP SEMI_COLON
| <primitive_functions> SEMI_COLON

A function can be called by writing the function name followed by parentheses. If the function has parameters, those parameters will be written inside parentheses.

3. Primitive Functions

<primitive_functions> ::= READ_ALTITUDE LP RP
| READ_TEMPERATURE LP RP
| READ_DIRECTION LP RP
| READ_TANK LP RP
| READ_LATITUDE LP RP
| READ_LONGITUDE LP RP
| READ_BATTERY LP RP
| CONNECT_TO_DRONE LP RP
| NOZZLE LP <term> RP
| VERTICAL_MOVEMENT LP <movement_expression> RP
| HORIZONTAL_MOVEMENT LP <movement_expression> RP
| TURN LP <movement_expression> RP
| SET_HFS LP<term> RP
| SET_HBS LP <term> RP
| SET_VUS LP <term> RP
| SET_VDS LP <term> RP
| SET_TS LP <term> RP

read_altitude(): This function will get the altitude of the drone and return it.
read_tempreture(): This function will get the temperature of the liquid inside tanks and return it.
read_direction(): This function will get the direction of the drone and return a number between 0 and 359.
read_tank(): This function will return a number between 0 and 100, showing how much liquid is remaining inside the tank.
read_latitude (): This function will return the latitude of the drone based on its current location.
read_longitude (): This function will return the longitude of the drone based on its current location.
read_battery(): This function will return a number between 0 and 100 showing the status of the drone's battery.
connect_to_drone (): This function will be the main way in which the drone will be communicating with the control center.
nozzle (<term>): This function will turn the nozzle off or on based on the parameter.
vertical_movement (<movement_expression>): This function will control the vertical movement of the drone.
horizontal_movement (<movement_expression>): This function will control the horizontal movement of the drone in the direction in which the drone is facing.
turn (<movement_expression>): This function will control the direction in which the drone is facing
set_HFS (<term>): This function will set the horizontal forward speed equal to the parameter inserted by user
set_HBS (<term>): This function will set the horizontal backward speed equal to the parameter inserted by user
set_VUS (<term>): This function will set the vertical upward speed equal to the parameter inserted by user
set_VDS (<term>): This function will set the vertical downward speed equal to the parameter inserted by user
set_TS (<term>): This function will set the turn speed equal to the parameter inserted by user

4. Terminals

SEMI_COLON is used at the end of statements. COMMA is used to separate parameters from each other. LBRACE RBRACE are used to embrace blocks of statements. LP and RP are used in function calls and declarations. For precedence in BNF, in our relation we have written expressions inside each other. This means, when the user writes a function declaration or an assignment statement, first the program goes to additive expressions. Inside the additive expressions (PLUS, MINUS or direct conduct to multiplicative expressions) BNF rule, we wrote the multiplication expressions (MULTIPLY, DIVIDE, and REMINDER) BNF rule. In this way, the leaves of the parse tree become our multiplicative expressions to do multiplicative expressions as precedence. Therefore, we do not use parentheses for the precedence. In addition, with the writing the BNF in left recursive, our program becomes left most precedence.

5. Descriptions of nontrivial tokens

Because GEKRONE is designed to be used by farmers and people who have no previous experience with programming languages, we decided to keep the language as simple as possible. In this way we allow the users to learn the language faster and use it more easily and efficiently without needing help from a professional.

5.1. Variable Types

One way of doing so was not including things which are not needed, by this i mean, there only exists numerical variables. As the program is only used to control a drone and its functions, there is no need to have variables like string. By only having integer and float, we make it easier and more practical for the user.

5.2. Variable Declaration

While defining a variable, this language does not put a rule about writing the type of variable. Either the identifier the user defines is an integer or a float, he/she enters the name (identifier) of the variable and the value which the program takes as a number then the program completes the variable declaration.

Since GEKRONE has two types in total, it does not need such a declaration while being designed.

5.3. For Loop

Another decision we made was making functions and loops simpler to use.
Our for loop is defined as simply as possible. All the user needs to do is write the number of times they want the loop to execute and it will work.

5.4. Function Declaration

In order to declare a new function, the user needs to use the keyword “gek”. This will make it easier for the user to learn and declare new functions. While declaring functions, there is no need to define its return type because functions will not return any value.

5.5. Movement Functions

Our language provides users with needed movement functions. These functions such as horizontal movement and vertical movement can be used in two different ways.

5.6. Auto Movement

First, AUTO which is the easiest method to move the drone. By using AUTO mode, the user only needs to define the amount of distance they want the drone to move. For example, HORIZONTAL_MOVEMENT (AUTO, 5) means “move forward for 5 meters”, or HORIZONTAL_MOVEMENT (AUTO, -3) means “move backward for 3 meters”. In AUTO mode, the speed of the drone will be a constant speed defined by the user using the set speed functions.

5.7. Manual Movement

The second way in which the user can move the drone is using MANUALmovement. In MANUAL movement, the user will define movement using speed and time. For example, HORIZONTAL_MOVEMENT (MANUAL, 5, 3) means “move forward at the speed of 5 m/s for 3 seconds”. This will give the user more flexibility while still being really simple to use.

6. Reserved Words

IF, ELSEIF, ELSE, FOR, WHILE, DO, HFS, HBS, VUS, VDS, TS, GEK and all primitive functions.

7. Evaluation of GEKRONE

Readability

In terms of readability, GEKRONE is a simple language to read and understand even by those who do not have experience. Primitive functions are easy to understand just by reading their name and parameters. Loops are also simple to read because they are defined in the simplest and most intuitive way. variables not having a type will not be a readability problem because there are only numeric type variables and confusing them will not pose any problem.

Writability

In terms of writability, GEKRONE is again a simple language to write. Everything has been designed in the simplest and most intuitive way. The idea behind this approach was to allow farmers and people without previous programming experience to write and modify their own drone programs. Because of this approach, it may be hard for some experienced people to write a program because they need to simplify their idea in order to write it in GEKRONE.

Reliability

By being an easy to read and write language, it is easier for the user to find and fix problems in the program in case they exist.