CS 315 Project 1 Report
Fall 2021
GEKRONE

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

GEKRONE

Program

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

Statements

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

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

<assignment_statement> ::= <variable_identifier> = <additive_expressions>;

<block_statements> ::= { } | {<statements>} | {<block_statements>} | <statement>

Variable Identifiers

<constant> ::= <numeric_constant> | <boolean_constant>

<numeric_constant> ::= <int_constant> | <float_constant>

<int_constant> ::= <int> | <char>

<digit> ::= [0-9]

<int> ::= [+-]?<digit>+

<char> ::= [A-Za-z]

<float_constant> ::= [+-]?<digit>+\.<digit>+

<boolean_constant> ::= 0 | 1

<variable_identifier> ::= <char>+[<char> | <digit>]*

<identifier_list> ::= <variable_identifier>
| <identifier_list>,<variable_identifier>

<string> ::= \“<int_constant>* | *| \n *| \t *\”

| <string>*

Operators

<assignment_operator> ::= =

expressions (arithmetic, relational, boolean, their combination)

<additive_expressions> ::= <additive_expressions> + <multiplicative_expressions>

| <additive_expressions> - <multiplicative_expressions>

| <multiplicative_expressions>

<multiplicative_expressions> ::= <multiplicative_expressions> * <term>

| <multiplicative_expressions> / <term>

| <multiplicative_expressions> % <term>

| <term>

<term> ::= <variable_identifier> | <constant>

<equality_expression> ::= <relational_expression> == <relational_expression>

| <relational_expression> != <relational_expression>

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

| <additive_expressions> > <additive_expressions>

| <additive_expressions> <= <additive_expressions>

| <additive_expressions> >= <additive_expressions>

<conditional_expression> ::= <equality_expression>

| <relational_expression>

| <conditional_expression> && <equality_expression>

| <conditional_expression> && <relational_expression>

| <conditional_expression> || <equality_expression>

| <conditional_expression> || <relational_expression>

<movement_expression> ::= auto , <term>
| manual , <term> , <term>

Loops

<loops> ::= while (<conditional_expression>)<block_statements>

| do<block_statements>while(<conditional_expression>);

| for(<term>)<block_statements>

Conditional statements

<if_statement> ::= if(<conditional_expression>)<statements>

<if_noelse_statement> ::= <if_statement> [elseif (<conditional_expression>)<statements>]*

<if_else_statement> ::= <if_noelse_statement> [else <statements>]?

<if_statements> ::= <if_statements> <if_else_statement>

| <if_else_statement>

Function definitions and function calls

<function_declaration> ::= gek <variable_identifier> (<identifier_list>) <block_statements>
| gek <variable_identifier> ( ) <block_statements>

<function_call> ::= <variable_identifier>()
| <variable_identifier> (<identifier_list>)
| <primitive_functions>

Comments

<comments> ::= /* <string> */

Primitive variables

HFS

HBS

VUS

VDS

TS

Primitive functions

<primitive_functions> ::= read_altitude ()
| read_temperature ()
| read_direction ()
| read_tank ()
| read_latitude ()
| read_longitude ()
| read_battery()
| connect_to_drone ()
| nozzle (<term>)
| vertical_movement (<movement_expression>)
| horizontal_movement (<movement_expression>)
| turn (<movement_expression>)
| set_HFS (<term>)
| set_HBS (<term>)
| set_VUS (<term>)
| set_VDS (<term>)
| set_TS (<term>)

Description of language constructs

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

Program is either block_statement or a list of 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>

A statement can be either a statement, assignment statement, a loop or a function.

<assignment_statement> ::= <variable_identifier> = <additive_expressions>;

This is the description of an assignment statement. The r value can either be a constant or an expression. 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> ::= { } | {<statements>} | {<block_statements>} | <statement>

This is the description of a block statement. A block statement can either be with or without usage of curly braces

<constant> ::= <numeric_constant> | <boolean_constant>

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

<numeric_constant> ::= <int_constant> | <float_constant>

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

<int_constant> ::= <int> | <char>

An integer constant is either an integer or character. In our language we define characters as integer constants. This is more useful for use.

<digit> ::= [0-9]

<int> ::= [+-]?<digit>+

An integer is an optional sign followed by one or more digits.

<char> ::= [A-Za-z]

A character is either uppercase or lowercase character .

<float_constant> ::= [+-]?<digit>+\.<digit>+

A float constant consists of an optional sign followed by one or more digits, followed by a dot, followed by one or more digits.

<boolean_constant> ::= 0 | 1

In our language, we define booleans as zero or one.

<variable_identifier> ::= <char>+[<char> | <digit>]*

A variable identifier is a character followed by one or more characters or digits.

<identifier_list> ::= <variable_identifier>
| <identifier_list>, <variable_identifier>

Identifier list is a list of identifiers separated by comma.

<string> ::= \“<int_constant>* | *| \n *| \t *\”

| <string>*

This is how a string is defined in our language. It can contain characters, digits, newLine, tab and space.

<assignment_operator> ::= =

Expressions (arithmetic, relational, boolean, their combination)

<additive_expressions> ::= <additive_expressions> + <multiplicative_expressions>

| <additive_expressions> - <multiplicative_expressions>

| <multiplicative_expressions>

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

<multiplicative_expressions> ::= <multiplicative_expressions> * <term>

| <multiplicative_expressions> / <term>

| <multiplicative_expressions> % <term>

| <term>

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

<term> ::= <variable_identifier> | <constant>

Term is either a variable identifier or a constant.

<equality_expression> ::= <relational_expression> == <relational_expression>

| <relational_expression> != <relational_expression>

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

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

| <additive_expressions> > <additive_expressions>

| <additive_expressions> <= <additive_expressions>

| <additive_expressions> >= <additive_expressions>

Relational expression checks the relation between two additive expressions.

<conditional_expression> ::= <equality_expression>

| <relational_expression>

| <conditional_expression> && <equality_expression>

| <conditional_expression>&&<relational_expression>

| <conditional_expression> || <equality_expression>

| <conditional_expression> || <relational_expression>

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

<movement_expression> ::= auto , <term>

| manual , <term> , <term>

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

<loops> ::= while (<conditional_expression>)<block_statements>

| do<block_statements>while(<conditional_expression>);

| for(<term>)<block_statements>

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

<if_statement> ::= if(<conditional_expression>)<statements>

If statement without else or elseif statements.

<if_noelse_statement> ::= <if_statement> [elseif (<conditional_expression>)<statements>]*

If statement followed by zero or more elseif statements.

<if_else_statement> ::= <if_noelse_statement> [else <statements>]?

If statement, followed by zero or more elseif statements, followed by an optional else.

<if_statements> ::= <if_statements> <if_else_statement>

| <if_else_statement>

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

Function definitions and function calls

<function_declaration> ::= gek <variable_identifier> (<identifier_list>) <block_statements>

| gek <variable_identifier> ( ) <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> ::= <variable_identifier>()

| <variable_identifier> (<identifier_list>)

| <primitive_functions>

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.

Comments

<comments> ::= /* <string> */

Comments will be written between /* and */. Any string that is between these two symbols will be considered a comment.

Primitive variables

HFS
: horizontal forward speed in meters per second

HBS
: horizontal backward speed in meters per second

VUS
: vertical upward speed in meters per second

VDS
: vertical downward speed in meters per second

TS
: turn speed in degree per second

Primitive functions

<primitive_functions> ::= read_altitude ()
| read_temperature ()
| read_direction ()
| read_tank ()
| read_latitude ()
| read_longitude ()
| read_battery()
| connect_to_drone ()
| nozzle (<term>)
| vertical_movement (<movement_expression>)
| horizontal_movement (<movement_expression>)
| turn (<movement_expression>)
| set_HFS (<term>)
| set_HBS (<term>)
| set_VUS (<term>)
| set_VDS (<term>)
| set_TS (<term>)

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

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.

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 or double. By only having integer and float, we make it easier and more practical for the user.

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.

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. As explained before, we do not have a variable type in our language and the difference between integer and float will be recognised by the compiler itself.

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.

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.

Manual movement:

The second way in which the user can move the drone is using manual movement. 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.

RESERVED WORDS:

if, elseif, else, for, while, do, HFS, HBS, VUS, VDS, TS, gek and all primitive functions.

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.

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.

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. In some cases, the user can encounter problems related to types if they are not careful because we do not have any types for our variables.