Bilkent University

CS-315

Fall 2022/2023

Project-1

06/10/2022

Member Name	Student ID	Section
Halil Alperen Gözeten	21902464	01
Kaan Tek	21901946	01
Korhan Kemal Kaya	21903357	01

CONTENTS

BNF Description of KRACK Language

1. Program

<program> ::= <begin><stmt_list><end>

<stmt_list> ::= <stmt>

| <stmt><stmt_list>

<stmt> ::= <assign_stmt><semicolon>

| <loop_stmt>

| <declaration_stmt><semicolon>

| <update_stmt><semicolon>

| <conditional_stmt>

| <return_stmt><semicolon>

| <comment>

| <quit_stmt><semicolon>

| <function><semicolon>

2. Statements

<assign_stmt> ::= <variable><assigner><expression>

| <variable><assigner><variable>

| <variable><assigner><literal>

| <variable><assigner><function>

<loop_stmt> ::= <while_stmt> | <for_stmt>

<conditional_stmt> ::= <if><open_p><expression><close_p>

<open_b><stmt_list><close_b>

| <if><open_p><expression><close_p>

<open_b><stmt_list><close_b><else><open_b><stmt_list><close_b
>

<declaration_stmt> ::= <variable_declaration>

| <function_definition>

| <object_declaration>

<return_stmt> ::= <return><variables>

| <return><literal>

| <return><expression>

| <return>

<update_stmt> ::= <increment_stmt> | <decrement_stmt>

<increment_stmt> ::= <increment_op><variable>

| <variable><increment_op>

<decrement_stmt> ::= <decrement_op><variable>

| <variable><decrement_op>

<comment> ::= <comment_op><characters><comment_op>

<quit_stmt> ::= <quit>

3. IoT Objects, Statements, and Symbols

<object_declaration> ::= <iot_object_type> <variable><open_p>

<close_p>

| <iot_object_type>

<variable><open_p><argument_list><close_p>

<object_function_call> ::= <variable><dot><iot_function><open_p>

<close_p>

| <variable><dot><iot_function><open_p><argument_list>

<close_p>

| <variable><dot><function_name><open_p><close_p>

| <variable><dot><function_name><open_p><argument_list>

<close_p>

<iot_object_type> ::= <url> | <connection> | <sensor> | <switch> |

<timer>

<iot_function> ::= <get_data> | <get_temperature> | <get_humidity>

| <get_air_pressure> | <get_air_quality> | <get_light>

| <get_sound_level> | <change_url> | <send_data> | <turn_on>

| <turn_off> | <get_timestamp>

<url> ::= URL

<connection> ::= Connection

<sensor> ::= Sensor

<switch> ::= Switch

<timer> ::= Timer

<get_data> :: _get_data

<get_temperature> :: _get_temperature

<get_humidity> ::= _get_humidity

<get_air_pressure> ::= _get_air_pressure

<get_air_quality> ::= _get_air_quality

```
<get_light> ::= _get_light
<get_sound_level> ::= _get_sound_level
<change_url> ::= _change_url
<send_data> ::= _send_data
<turn_on> ::= _turn_on
<turn_off> ::= _turn_off
<get_timestamp> ::= _get_timestamp
```

4. Types and Literals

```
<type_id> ::= int|float|string|char|bool
<literal> ::= <int>|<float>|<string>|<char>|<bool>
<int> ::= [+-]?[0-9]+
<float> ::= [+-]?[0-9]*(\.)[+-]?[0-9]+
<string> ::= <d_quote>[^\"]*<d_quote>
<char> ::= <s_quote>[^\']<s_quote>
<bool> ::= true | false
```

5. Variables and Identifiers

```
<variables> ::= <variable> | <constant_variable>
<variable> ::= <letter> | <variable><digit>
      | <variable><letter>
      | <variable><underscore>
<constant_variable> ::= <upper_case_letter>
      | <constant_variable><upper_case_letter>
      | <constant_variable><underscore>
<variable_declaration> ::= <type_id><variables><assigner>
     <expression>
<lower_case_letter> ::= [a-z]
<upper_case_letter> ::= [A-Z]
<letter> ::= <lower_case_letter> | <upper_case_letter>
<digit> ::= [0-9]
```

6. Expressions

```
<expression> ::= <logic_expression> | <arithmetic_expression>
```

<arithmetic_expression> ::=
<arithmetic_expression><arith_low_precedence_op><term>
| <term>
<term> ::= <term><arith_high_precedence_op><factor>| <factor>
<arith_high_precedence_op> ::= <multiplication_op> | <division_op>
| <modulus_op>
<arith_low_precedence_op> ::= <addition_op> | <subtraction_op>
<factor> ::= <open_p><arithmetic_expression><close_p>
| <variables>
| <integer>
| <float>
| <function>
<logic_expression> ::=
<open_p><logic_expression><logic_op><logic_expression>
<close_p>
| <open_p><comparison_expression><close_p>
| <variables>
| <literal>
| <function>
| <not_op><logic_expression>
<comparison_expression> ::= <variables><comparison_op><variables>
|<variables><comparison_op><literals>
|<literals><comparison_op><literals>
|<literals><comparison_op><variables>

7. Loop Statements

<for_stmt> ::= <for><open_p><for_initial><semicolon>
<expression><semicolon><assign_stmt><close_p><open_b>
<stmt_list><close_b>
<for_initial> ::= <assign_stmt> | <variable_declaration> |
<variable>
<while_stmt> ::= <while><open_p><expression><close_p>
<open_b><stmt_list><close_b>

8. Functions

<function> ::= <function_call> | <primitive_function_call>
| <object_function_call>
<function_call> ::= <function_name><open_p><close_p>
| <function_name><open_p><argument_list><close_p>
<primitive_function_call> ::= <input_function> | <output_function>
<function_definition> ::= <def><function_name><open_p><close_p>
<open_b><stmt_list><close_b>
| <def><function_name><open_p><parameter_list><close_p>
<open_b><stmt_list><close_b>
<function_name> ::= <underscore><letter>
| <function_name><letter>
| <function_name><digit>
| <function_name><underscore>
<argument_list> ::= <variables>
| <variables><comma><argument_list>
<parameter_list> ::= <type_id><variables>
| <type_id><variables><comma><parameter_list>
<input_function> ::= <in><open_p><close_p>
| <in><open_p><string><close_p>
<output_function> ::= <out><open_p><output_list><close_p>
<output_list> ::= <variables> | <literal>
| <variables><comma><output_list>
| <literal><comma><output_list>

9. Symbols and Operators

<open_p> ::= (
<close_p> ::=)
<open_b> ::= {
<close_b> ::= }
<s_quote> ::= ‘
<d_quote> ::= “
<logic_op> ::= && | ||
<comparison_op> ::= < | > | <= | >= | == | !=
<comma> ::= ,

<comment_op> ::= #

<assigner> ::= =

<semicolon> ::= ;

<return> ::= return

<quit> ::= quit

<def> ::= def

<for> ::= for

<while> ::= while

<if> ::= if

<else> ::= else

<begin> ::= begin

<end> ::= end

<in> ::= in

<out> ::= out

<addition_op> ::= +

<subtraction_op> ::= -

<division_op> ::= /

<multiplication_op> ::= *

<modulus_op> ::= %

<increment_op> ::= ++

<decrement_op> ::= --

<underscore> ::= _

<dot> ::= .

Description of KRACK Language Components

1. Program

- **<program>** ::= <begin><stmt_list><end>

Any program written in the KRACK language consists of a statement list which should be written between the `begin` and `end` terminals.

- **<stmt_list>** ::= <stmt>
  | <stmt><stmt_list>

A statement list may consist of a single statement or a list of statements.

- **<stmt>** ::= <assign_stmt><semicolon>
  | <loop_stmt>
  | <declaration_stmt><semicolon>
  | <conditional_stmt>
  | <return_stmt><semicolon>
  | <comment>
  | <quit_stmt><semicolon>
  | <function><semicolon>

A statement can be of various types. A combination of those leads to a statement list and, thus, a program. Those statements have different capabilities and differ from each other. Each of them will be explained in detail in the next section.

2. Statements

- **<assign_stmt>** ::= <variable><assigner><expression>
  | <variable><assigner><variable>
  | <variable><assigner><literal>
  | <variable><assigner><function>

Assign statements are used to assign a value to a variable (which can't be a constant), and the assigned value can be of different types: expression, variable, literal, or a result of a function.

- **<loop_stmt>** ::= <while_stmt> | <for_stmt>

There are two types of loop statements in KRACK language, while statement and loop statement. These statements are explained more in detail in their corresponding section, however, they are pretty similar to the traditional while and for statements that are used in most common languages.

- **<conditional_stmt>** ::= <if><open_p><expression><close_p>
  <open_b><stmt_list><close_b>
  | <if><open_p><expression><close_p>
  <open_b><stmt_list><close_b><else><open_b><stmt_list>
  <close_b>

Conditional statements are used to decide whether an action should be taken or not. A conditional statement starts with the `if` keyword and can optionally be followed by an `else` block. All expressions defined in the KRACK language have a corresponding boolean value (explained in detail in section five); hence any expression can be used as a condition of the conditional statements. We kept the general structure similar to the traditional if-else statement of many common languages in order to increase the writability. Also, the use of curly braces is required to write complete if or if-else blocks. This usage of braces prevents ambiguity while increasing the readability of the language.

- **<declaration_stmt>** ::= <variable_declaration>
  | <function_definition>
  | <object_declaration>

Declarations in our language can be of two types: variable declaration and function declaration. When declaring a variable, the initialization should also be performed at the same time to increase readability and reliability.

- **<return_stmt>** ::= <return><variables>
  | <return><literal>
  | <return><expression>

| <return>

Return statements are also similar to the traditional return logic in most programming languages; they are used to exit a function or return a value as well. A variable, literal, or expression can be returned after the specified return keyword.

- **<update_stmt>** ::= <increment_stmt> | <decrement_stmt>

The update statements are an easy way for the users to change the value of variables. It can be of two types, increment or decrement, each of which changes the value of the variable by 1.

3. Iot Objects, Statements, and Symbols

- **<object_declaration>** ::= <iot_object_type> <variable><open_p> <close_p>
  | <iot_object_type> <variable><open_p><argument_list> <close_p>

The objects required for the IoT operations can be created by passing an optional number (can be zero) of variables to the constructor. In providing the necessary functionalities for IoT operations, we had to choose between providing through functions or through objects. We've chosen object-oriented programming over functional as both of them bring a reliability deficiency even though the object-oriented way is more user-friendly and readable. The same reliability problem for both methods is that we needed passing variables to the constructors of the objects or to the functions to specify their connection with real hardware components, yet variables can be of any type because of our BNF description. One way to overcome this problem would be to force a different naming convention for IoT object variables, however, we chose not to implement this as it would hinder the writability of the language as we already have other naming conventions for functions and constant variables.

- **<object_function_call>** ::=
  <variable><dot><iot_function><open_p><close_p>
  | <variable><dot><iot_function><open_p><argument_list> <close_p>
  | <variable><dot><function_name><open_p><close_p>

| <variable><dot><function_name><open_p><argument_list>
<close_p>

The objects we created work through functions as part of object-oriented design. One reliability issue with our BNF approach for this architecture is that any method call is allowed with any IoT variable, which wouldn't be the case in real life. However, as we mentioned, we needed to force extra requirements in the naming of the IoT variables to overcome this issue, which makes our language less writable, so we chose to leave this as the programmer's responsibility.

- **<iot_object_type>** ::= <url> | <connection> | <sensor>
  | <switch> | <timer>

These are the provided built-in object types for the KRACK language, which we believe are enough for a programmer to write any kind of functionality by interacting with these objects.

- **<iot_function>** ::= <get_data> | <get_temperature>
  | <get_humidity> | <get_air_pressure> |
  <get_air_quality> | <get_light> | <get_sound_level> |
  <change_url>
  | <send_data> | <turn_on> | <turn_off> | <get_timestamp>
- **<url>** ::= URL

This is the URL object that is used in every Connection object to provide the connection for the device.

- **<connection>** ::= Connection

This is the Connection object that is used to provide the connection for the device together with the URL object. It is used to send and receive data.

- **<sensor>** ::= Sensor

This is the Sensor object used to get various types of data, such as temperature, humidity, light, sound level and etc., from actual hardware by invoking methods on it. It's created by supplying the related hardware number on the constructor.

- **<switch>** ::= Switch

This is the Switch object that is physically wired with some actuators. The status of various switches might control the status of the actuators. This object is created by supplying the related physical switch number on its constructor.

- **<timer>** ::= Timer

This is the Timer object which the user will interact with to get the current timestamp value.

- **<get_data>** ::= _get_data

This is a provided function that belongs to the Sensor and Connection objects to get data. The Connection object will use it to fetch data, and the Sensor object will return relevant information about the data it measures, which can be considered as the general case of functions like `_get_temperature`, `_get_humidity` and etc.

- **<get_temperature>** ::= _get_temperature

This is a provided function that belongs to the Sensor object to get the current temperature value.

- **<get_humidity>** ::= _get_humidity

This is a provided function that belongs to the Sensor object to get the humidity level.

- **<get_air_pressure>** ::= _get_air_pressure

This is a provided function that belongs to the Sensor object to get the air pressure.

- **<get_air_quality>** ::= _get_air_quality

This is a provided function that belongs to the Sensor object to get the air quality.

- **<get_light>** ::= _get_light

This is a provided function that belongs to the Sensor object to get the amount of light.

- **<get_sound_level>** ::= _get_sound_level

This is a provided function that belongs to the Sensor object to get the level of sound given a frequency.

- **<change_url>** ::= _change_url

This is a provided function that belongs to the Connection object to change the connected URL address.

- **<send_data>** ::= _send_data

This is a provided function that belongs to the Connection object to send data through this function to the connected URL address.

- **<turn_on>** ::= _turn_on

This is a provided function that belongs to the Switch object to turn it on.

- **<turn_off>** ::= _turn_off

This is a provided function that belongs to the Switch object to turn it off.

- **<get_timestamp>** ::= _get_timestamp

This is a provided function that belongs to the Timer object to get the current timestamp.

4. Types and Literals

- **<type_id>** ::= int|float|string|char|booL

The supported types in the KRACK language include int, float, string, char, and bool. These types are similar to those in other common languages. These types are used in variable declarations and parameter lists.

- **<literal>** ::= <int>|<float>|<string>|<char>|<bool>

Literals are used to express the fixed values in the code.

- **<int>** ::= [+-]?[0-9]+

<int> variable is used to represent integers, which can have an optional preceding sign, to denote positivity or negativity.

- **<float>** ::= [+-]?[0-9]*(\.)[+-]?[0-9]+

<float> variable is used to represent floating point numbers, which can have an optional preceding sign to denote positivity or negativity. A preceding dot should be included before the fraction part.

- **<string>** ::= <d_quote>[^\"]*<d_quote>

A string in KRACK language should be written inside double quotation marks, as it is in most common languages. A string can contain any character except a “.

- **<char>** ::= <s_quote>[^\']<s_quote>

A char is used to represent a single character, which can be any character except ‘.

- **<bool>** ::= true | false

A bool is represented by either a `true` or `false` keyword.

5. Variables and Identifiers

- **<variables>** ::= <variable> | <constant_variable>

Variables can be of 2 types. `<variable>` is a type whose value can be changed after the declaration, whereas a `<constant_variable>` is the type of variable whose value cannot be changed after the declaration, namely a constant one.

- **<variable>** ::= <letter> | <variable><digit>
  | <variable><letter>
  | <variable><underscore>

A `<variable>` can consist of a combination of letters, digits, or the underscore character.

- **<constant_variable>** ::= <upper_case_letter>
  | <constant_variable><upper_case_letter>
  | <constant_variable><underscore>

A `<constant_variable>` is similar to a regular variable, except that it should contain only capital letters or the underscore character.

- **<variable_declaration>** ::= <type_id><variables><assigner>
  <expression>

In order to declare a variable, its type and its name should be specified, followed by an assigner terminal which is a =, and then the value should be provided, which can be a result of an expression. The whole variable declaration should be performed on a single line in order to increase readability and reliability.

- **<lower_case_letter>** ::= [a-z]
- **<upper_case_letter>** ::= [A-Z]
- **<letter>** ::= <lower_case_letter> | <upper_case_letter>

Letters consist of the English alphabet letter, which can be lower or upper case.

- **<digit>** ::= [0-9]

Digits are any number that is between 0 and 9.

6. Expressions

- **<expression>** ::= <logic_expression>| <arithmetic_expression>

An expression is either a logic expression or an arithmetic expression. They are combinations of variables, literals, operators, and functions that are used for computation to produce a value, which is either a boolean, integer or float type. In KRACK language, they can be one of two types `logic_expression` and `arithmetic_expression`.

- **<arithmetic_expression>** ::=
  <arithmetic_expression><arith_low_precedence_op><term>
  | <term>
- **<term>** ::= <term><arith_high_precedence_op><factor>| <factor>
- **<factor>** ::= <open_p><arithmetic_expression><close_p>
  | <variables>
  | <integer>
  | <float>
  | <function>

Arithmetic expressions are performed in a similar way to other programming languages; however, in KRACK language, new variables named term and factor are defined to indicate operator precedence and prevent ambiguity. Multiplication, division, and modulo are equal in precedence, which is higher than the precedence of addition or subtraction, which also have the same precedence. They return either a float or an integer.

- **<logic_expression>** ::=
  <open_p><logic_expression><logic_op><logic_expression>
  <close_p>
  | <open_p><comparison_expression><close_p>
  | <variables>
  | <literals>

| <function>
| <not_op><logic_expression>

Logic expressions are combinations of logical operators (and=&& & or = ||), variables, literals, and functions, which results in a boolean that has a value of false or true. Also, any literal can be regarded as a boolean in KRACK in the following manner:

If a string is empty, its boolean value is false. Else, true.

If the int value is 0, its boolean value is false. Else, true.

If the float value is 0, its boolean value is false. Else, true.

If char is the null character (ASCII numeric value is 0), its boolean value is false. Else, true.

With the use of parentheses, any number of logical expressions can be combined, and this approach increases readability and reliability. Moreover, we allowed any number of not operations, "!" inside a logic expression.

- **<comparison_expression>** ::=
  <variables><comparison_op><variables>
  | <variables><comparison_op><literals>
  | <literals><comparison_op><literals>
  | <literals><comparison_op><variables>

Comparison expressions are used to compare variables and literals, which return a boolean value as the result.

7. Loop Statements

- **<for_stmt>** ::= <for><open_p><for_initial><semicolon>
  <expression><semicolon><assign_stmt><close_p><open_b>
  <stmt_list><close_b>
- **<for_initial>** ::= <assign_stmt> | <variable_declaration> |
  <variable>

For statement in the KRACK language is similar to the for implementation in other common languages. <for_initial> variable is used as the initializer in the for loop, which can be an assignment, a variable declaration, or a pre-defined variable. The body of the loop, which is a statement list, will be in between curly

braces, and it will execute until the <expression> in for statement has a boolean value of false.

- **<while_stmt>** ::= <while><open_p><expression><close_p>
  <open_b><stmt_list><close_b>

While statement is similar as the body consists of a statement list that is closed in between curly braces, and it executes until the <expression> in while statement is not satisfied.

8. Functions

- **<function>** ::= <function_call> | <primitive_function_call>
  | <object_function_call>

There are three types of functions in KRACK language. <function_call> is used to call functions that are defined by the user in the program, and <primitive_function_call> is used to call predefined functions in the KRACK language that exist in the standard library itself. Finally, <object_function_call> is used to describe the functions that can be invoked on IoT-related objects such as Sensor and Connection.

- **<function_call>** ::= <function_name><open_p><close_p>
  | <function_name><open_p><argument_list><close_p>

In order to call the user-defined functions; function name followed by a pair of parentheses should be supplied. If the function takes arguments, an <argument_list> is also supplied in between the parentheses.

- **<primitive_function_call>** ::= <input_function>
  | <output_function>

Primitive functions can be two types, an input function used for getting an input from the user, and an output function used for printing to the screen.

- **<function_definition>** ::=
  <def><function_name><open_p><close_p><open_b><stmt_list>
  <close_b>
  | <def><function_name><open_p><parameter_list><close_p>

<open_b><stmt_list><close_b>

Function definitions start with the <def> keyword, followed by the function name and a pair of parentheses, which optionally includes a parameter list. Then comes the function body, between a pair of curly braces, which consists of a statement list.

- **<function_name>** ::= <underscore><letter>
  | <function_name><letter>
  | <function_name><digit>
  | <function_name><underscore>

By convention we forced the function name to start with an underscore, then followed by a number of letter, digit, or the underscore character. This is a feature that helps the users easily distinguish between variable and function names.

- **<argument_list>** ::= <variables>
  | <variables><comma><argument_list>

Argument list consists of one or more variables, separated by commas.

- **<parameter_list>** ::= <type_id><variables>
  | <type_id><variables><comma><parameter_list>

Similar to an argument list, parameter list consists of one or more variables which are separated by commas, but types of the variables should also be supplied

- **<input_function>** ::= <in><open_p><close_p>
  | <in><open_p><string><close_p>

This primitive function is used get input from the user. It is denoted by the in keyword, followed by a pair of parentheses. The programmer can also include a message between the parentheses that is to be displayed to the user when they see the prompt.

- **<output_function>** ::= <out><open_p><output_list><close_p>

This primitive function is used to print on the screen. It is denoted with the out keyword, followed by an output_list between parentheses.

- **<output_list>** ::= <variables> | <literal>

```
| <variables><comma><output_list>
| <literal><comma><output_list>
```

Output list can consist of one or more variables and literals, separated by commas.

9. Symbols and Operators

- **<open_p>** ::= (
- **<close_p>** ::=)

Used in if statements, loops, arithmetic expressions, function definitions, and function calls.

- **<open_b>** ::= {
- **<close_b>** ::= }

Used in function bodies and bodies of loops, if and else statements.

- **<s_quote>** ::= ‘

Used to denote chars.

- **<d_quote>** ::= “

Used to denote strings.

- **<logic_op>** ::= && | ||

Used in logical expressions, denotes AND and OR, respectively.

- **<comparison_op>** ::= < | > | <= | >= | == | !=

Used for comparing variables and literals.

- **<comma>** ::= ,

Used to separate variables in argument and parameter list, and literals too in output list

- **<comment_op>** ::= #

Used to denote starting and ending point of a comment.

- **<assigner>** ::= =

Used in assignment statement and variable declarations.

- **<semicolon>** ::= ;

Used to denote the end of a statement and used in for statements.

- **<return>** ::= return

A reserved word used in return statements to return a variable, a literal, an expression, or simply to terminate the function.

- **<quit>** ::= quit

A reserved word used in quit statements to exit a loop.

- **<def>** ::= def

A reserved word used in function definition.

- **<for>** ::= for

A reserved word used to denote a for loop.

- **<while>** ::= while

A reserved word used to denote a while loop.

- **<if>** ::= if

A reserved word used to denote an if block in conditional statement.

- **<else>** ::= else

A reserved word used to denote an else block in conditional statement.

- **<begin>** ::= begin

A reserved word used to denote the starting point of the program.

- **<end>** ::= end

A reserved word used to denote the ending point of the program.

- **<in>** ::= in

A reserved word used to denote the primitive input function.

- **<out>** ::= out

A reserved word used to denote the primitive output function.

- **<addition_op>** ::= +

Used to denote the plus sign in arithmetic expressions.

- **<subtraction_op>** ::= -

Used to denote the minus sign in arithmetic expressions.

- **<division_op>** ::= /

Used to denote the division sign in arithmetic expressions.

- **<multiplication_op>** ::= *

Used to denote the multiplication sign in arithmetic expressions.

- **<modulus_op>** ::= %

Used to denote the modulo sign in arithmetic expressions.

- **<increment_op>** ::= ++

Used in update statement to increment the value by one.

- **<decrement_op>** ::= —

Used in update statement to decrement the value by one.

- **<underscore>** ::= _

Used in function definitions before the function name.

- **<dot>** ::= .

Used to invoke functions on the IoT objects, comes after the variable name and before the function name.

Description of KRACK Language Non-Trivial Tokens

NEWLINE: This is a token only for demonstrating purposes for the lexical analyzing process to emphasize that a newline character "\n" exists. This token will not be used by the syntactic analyzer, and it's also not used in the BNF form.

OPENP: This is a token reserved for "(" character. It's used in expressions, functions, conditionals, and loops in order to avoid ambiguity together with a ")" character.

CLOSEP: The main purpose of this token is to demonstrate an expression or operation starting with the **OPENP** token has reached its end. It is ")", and it helps us to avoid ambiguity and provides reliability and neatness together with the **OPENP** token.

OPENB: This token is used to emphasize that a statement list can start in it, and it also defines a scope. As demonstrated by "{", it is used before loop, function, and conditional bodies.

CLOSEB: When used after the **OPENB** token, this token emphasizes that the statement list has reached to end. Indicating that scope starting with **OPENB** token has finished, the "}" character also increases reliability and readability.

ASSIGNER: This token is reserved for assignment operation. It is an operator that assigns the left-hand side's value to the right-hand side of the assignment operation.

DOT: This token is reserved for invoking functions on IoT objects, it should be preceeded by a **VARIABLE** token that is the name of the object and followed by a IoT function name token. The usage of this symbol increases the reliability and readability of our language as it's used only for IoT object related actions.

COMMA: This token is reserved for listing and ordering the parameters and outputs in function declarations and arguments in function calls. It's usage increases reliability and readability.

SEMICOLON: It is a token dedicated to demonstrating a statement has reached to end. Also, it is used in the for loop after an expression. It's usage increases readability.

INCREMENT_OP: This token is used to identify the increment operator “++”. The increment operator, which increments the variable's value by one and assigns it to itself, increases the writability.

DECREMENT_OP: This token is used to identify the decrement operator “--”. The decrement operator, which decreases the variable's value by one and assigns it to itself, increases the writability.

EQUAL_OP: This token is used to identify the equality operator “==”. The equality operator is dedicated to check whether two sides of a comparison expression are equal. It returns true if the condition is met.

NOT_EQUAL_OP: This token is used to identify inequality operator “!=”. The inequality operator is dedicated to check whether two sides of an comparison expression is inequal. It returns true if the condition is met.

ADDITION_OP: This token is used to identify addition operator “+”.

SUBTRACTION_OP: This token is used to identify subtraction operator “-”.

MULTIPLICATION_OP: This token is used to identify multiplication operator “*”.

DIVISION_OP: This token is used to identify the division operator “/”.

MODULUS_OP: This token is used to identify the modulus operator “%”.

AND_OP: This token is used to identify and operator “&&”.

OR_OP: This token is used to identify or operator “||”.

LT_OP: This token is used to identify less than operator “<”.

GT_OP: This token is used to identify greater than operator “>”.

LT_EQ_OP: This token is used to identify less than or equal to operator “<=”.

GT_EQ_OP: This token is used to identify greater than or equal to operator “>=”

BEGIN: This token identifies the reserved word begin. Any program in KRACK must start with begin reserved word, which is a feature that increases the reliability of the language.

END: This token identifies the reserved word end. Any program in KRACK must end with end reserved word, which increases the reliability of the language.

INT_TYPE: This token is reserved for variable type int. It is used while defining a variable to indicate its type is integer.

FLOAT_TYPE: This token is reserved for variable type float. It is used while defining a variable to indicate its type is float.

STRING_TYPE: This token is reserved for variable type string. It is used while defining a variable to indicate its type is string.
CHAR_TYPE: This token is reserved for variable type char. It is used while defining a variable to indicate its type is character.
BOOL_TYPE: This token is reserved for variable type bool. It is used while defining a variable to indicate its type is bool.
TIMER: This token is reserved to describe a Timer object and used to indicate the type in the object declaration statement.
SWITCH: This token is reserved to describe a Switch object and used to indicate the type in the object declaration statement.
SENSOR: This token is reserved to describe a Sensor object and used to indicate the type in the object declaration statement.
CONNECTION: This token is reserved to describe a Connection object and used to indicate the type in the object declaration statement.
URL: This token is reserved to describe a URL object and used to indicate the type in the object declaration statement.
QUIT: This is a token for describing early exits (break) from iterations like “for” and “while” loops. This provides efficiency and easiness in the implementation and is a common feature of many known languages.
FOR: This is a token reserved for describing “for” loops in our language, which provides support for iterated actions, just as many known languages like do.
WHILE: This is a token reserved for the declaration of “while” loops in our language, which is another way of supporting iterated actions of users. It’s commonly used by all well-known languages (Java, C++, Phyton and etc.) as it provides easiness.
DEF: This is a token reserved for the declaration of function definitions in our language. This token must be followed by the tokens related to the function name, arguments, and function body.
RETURN: This is a token for describing the exiting from a function, possibly followed by other tokens describing the return value. It’s again a common practice used by most programming languages as it’s an easiness for users.

IF: This token is for describing the start of the conditional statements, and it might be possibly followed later by an **ELSE** token. It's an easiness for the users of the language.

ELSE: This token is to describe the second part of a conditional statement; it must be preceded with an **IF** token for it to be meaningful. It's an easiness for the users of the language.

GET_DATA: This token is used to describe the name of the `_get_data` function, which is invoked on either Connection or Sensor objects to get relevant data. This token should be preceded by **DOT** and **VARIABLE** tokens.

GET_TEMPERATURE: This token is used to describe the name of the `_get_temperature` function, which is invoked on a Sensor object to get the current temperature value. This token should be preceded by **DOT** and **VARIABLE** tokens.

GET_HUMIDITY: This token is used to describe the name of the `_get_humidity` function, which is invoked on a Sensor object to get the humidity value. This token should be preceded by **DOT** and **VARIABLE** tokens.

GET_AIR_PRESSURE: This token is used to describe the name of the `_get_air_pressure` function, which is invoked on a Sensor object to get the air pressure value. This token should be preceded by **DOT** and **VARIABLE** tokens.

GET_AIR_QUALITY: This token is used to describe the name of the `_get_air_quality` function, which is invoked on a Sensor object to get the air quality value. This token should be preceded by **DOT** and **VARIABLE** tokens.

GET_LIGHT: This token is used to describe the name of the `_get_light` function, which is invoked on a Sensor object to get the light value. This token should be preceded by **DOT** and **VARIABLE** tokens.

GET_SOUND_LEVEL: This token is used to describe the name of the `_get_sound_level` function, which is invoked on a Sensor object to get the sound level value. This token should be preceded by **DOT** and **VARIABLE** tokens.

CHANGE_URL: This token is used to describe the name of the `_change_url` function, which is invoked on a Connection to change the URL attribute of the object. This token should be preceded by **DOT** and **VARIABLE** tokens.

SEND_DATA: This token is used to describe the name of the `_send_data` function, which is invoked on a Connection object to send relevant data. This token should be preceded by **DOT** and **VARIABLE** tokens.

TURN_ON: This token is used to describe the name of the `_turn_on` function, which is invoked on a Switch object to turn it on. This token should be preceded by **DOT** and **VARIABLE** tokens.

TURN_OFF: This token is used to describe the name of the `_turn_off` function, which is invoked on a Switch object to turn it off. This token should be preceded by **DOT** and **VARIABLE** tokens.

GET_TIMESTAMP: This token is used to describe the name of the `_get_timestamp` function, which is invoked on a Timer object to get the number of seconds elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970. This token should be preceded by **DOT** and **VARIABLE** tokens.

IN: This is a token reserved for describing the built-in primitive function **in()** to take inputs from the user.

OUT: This is a token reserved for describing the built-in primitive function **out()** to take inputs from the user, and it provides convenience.

INTEGER: This token is reserved for integer literals.

FLOAT: This token is reserved for float literals.

CHAR: This token is reserved for character literals.

STRING: This token is reserved for string literals.

BOOL: This token is reserved for boolean literals.

COMMENT: This token is reserved for single and multi-line comments. Comments must start and end with a hashtag. This attribute should increase both the readability and reliability of the language.

CONSTANT_VARIABLE: This token is used to identify constant variables. As a convention, we forced all constant variables to consist of only capital letters and the underscore character, which increases the reliability and readability as it provides an easy distinction between constants and non-constants.

VARIABLE: This token is used to identify the non-constant variables that start with a letter followed by alphanumeric characters. This token contains at least one non-uppercase letter character as it's non-constant.

`FUNCTION_NAME:` This token is used to identify function names. As a convention, all function names must start with an underscore and letter followed by optional alphanumeric characters. This attribute increases reliability and readability as it helps to distinguish between variable and function names.

Evaluation of KRACK Language Design Criteria

1. Readability

The syntax of the KRACK language is generally similar to the syntax of most of the common well-known languages, such as Java and C-level languages. A developer who is new to this language can easily grasp the structure of conditional statements, loops, and functions; which makes this language more readable. In addition to those, CRACK language requires function names to start with an underscore. This also increases the readability because someone can easily differentiate the function name from a variable name when they see an underscore. Also, curly braces are required in **`if-else`** statements, functions, and loops. This also increases the overall readability of the language. Semicolons are also required at the end of a statement, which again makes it easy to read and separate one statement from another. However, this feature decreases writability, as the programmer is forced to put a semicolon every time after a statement. We also forced the constant variables to consist of only capital letters and the underscore character, which makes it easier to differentiate them from the non-constant ones by the users, thus, increasing the readability. Also, some reserved words such as **`def`**, **`begin`**, and **`end`** increases the readability, as these words make it easy for the reader to recognize when a function definition starts or where the program starts and ends, respectively.

2. Writability

KRACK language is designed especially for IoT edge devices in contrast to other programming languages. Many functions, operations, and objects that an IoT node may require are built into the language itself. Thus, KRACK has a strong writability attribute compared to other languages when anyone wants to develop an IoT node. Moreover, KRACK supports, similar to advanced programming languages, functions, conditional statements, loop statements, inputting, and outputting. This prevents a programmer from writing the same code many times,

hence, increasing the writability. Also, we chose to implement IoT functionalities through object-oriented design, which we believe increased the writability of our language as it is a natural way for the programmer to interact with the physical world. Furthermore, KRACK supports different operators for flexible coding. Examples of this situation are increment (++) and decrement (--) operators, which boost writability.

In contrast to those features increasing the writability of the language, there are some trade-offs leading to a decrease in writability. Semicolons after statements, parentheses before and after expressions, and curly braces to emphasize scopes increases both the readability and reliability of the code, whereas they reduce the writability by adding restrictions. Additionally, since function and constant names have certain restrictions and conventions increasing readability, they lead to a decrease in both orthogonality and writability of the language. To limit the effects of this situation, KRACK does not force naming conventions for IoT variables, objects, and functions but uses the same convention for regular variables. This leads to a decrease in the reliability of the language since IoT functions are allowed to be called on any variable in our grammar, but we chose to leave avoiding this situation to the responsibility of the programmer while not permitting a decrease in the writability.

3. Reliability

A feature of the KRACK language is that variables must have a value assigned to them in their declaration. This feature provides users with a reliable implementation process as it avoids the usage of an uninitialized variable. Moreover, the user also needs to supply a type for the variable in a declaration statement which might be helpful in tracking semantic errors originating from type issues if a compiler is implemented for this language in the future. Another approach that we took to increase the reliability of the KRACK language was the use of the **`def`** keyword before the function definitions. Together with this, we kept the use of the underscore symbol _ before the function names as a convention. We chose to have these two requirements as they provide the users more secure implementation process, even though it's a compromise to the writability of the language. One thing common for the conditional and iterative

statements was the compulsory use of the curly braces (`{` and `}`) before and after the list of statements inside these constructs. This helps to define a clear scope for the variables inside these blocks (if a compiler is designed) and provides secure implementation by helping detect errors like a variable is accessed outside its scope. Similar to this, we've chosen the **#** as the comment starting symbol and enforced another **#** symbol to close the comment. This is a slight decrease in the writability of our language, but we kept this convention as it might be particularly helpful in avoiding conflicts between code and comments. Besides that, the use of reserved words like **`begin`** and **`end`** ensures a clear distinction for the starting and beginning of the program code and helps to omit any text outside them.