Key Programming Language

CS315 SECTION 1 TEAM 42 PROJECT 1

İlkim Elif Kervan 22002223
Kemal Kubilay Yılmaz 21901556
Yağız Can Aslan 22001943

Explanation of Language Constructs and Nontrivial Tokens

- **if-else statements:** If-else is a conditional statement whose if block or else block takes action under specific conditions. The condition in this statement can be directly a boolean literal (i.e. true or false), a boolean variable, comparison expressions or combinations of boolean literals, boolean variables and comparison expressions (i.e., a “boolean list” in Key). If the condition in the parentheses is true, the block –statements between the “{“ and “}” characters– right after the terminal “if”, takes action. Otherwise, the block right after the “else” (if “else” exists) takes action. Moreover, if-else statements in Key are defined to be unambiguous selectors with the “matched/unmatched” definitions, where “matched” represents one or more “if-else” pair(s) –same number of “if” and “else” statements–, and “unmatched” represents an unbalanced number of “if” and “else” statements (there are more “if” statements than “else” statements). If-else statement is designed similar to other C-family programming languages, as it is a fundamental statement in Key, which increases readability and writability through similarity.

- **while loops:** While is a loop statement which iterates the block –statements between the “{“ and “}” characters– after itself based on the condition(s) present between the “(“ and “)” characters. The condition(s) in this while loops can be directly a boolean literal (i.e. true or false), a boolean variable, comparison expressions or combinations of boolean literals, boolean variables and comparison expressions (i.e., a “boolean list” in Key). While the condition in between the parentheses is true, the statement between curly braces continues to execute. While is designed similar to other C-family programming languages, as it is a fundamental statement in Key. This increases readability and writability through similarity.

- **do-while loops:** Do-while loop has similar properties with the while loop. However, in the do-while loop there is a “do” keyword followed by a code block –statements between the “{“ and “}” characters– right before the “while” keyword. Regardless of the condition for the while loop, the block after the “do” keyword is executed at least once, and continues to execute while the loop condition is true. Having a do-while loop in addition to a while loop in Key, increases writability of the programs since the do-while loop expands the developers arsenal for implementing their designs into code in Key.

- **for loops:** For is a keyword which is followed by parentheses with specific statements and a code block which includes statements to be executed. Parentheses after the keyword “for” have three statements. The first one may be an assignment or a declaration statement. The second one is a conditional statement, which are exactly the same in the conditions in the if statements. Lastly, the third one may be increment operation, decrement operation, assignment operation or a math statement. The naming and the logic of the for loop are similar to the naming and logic conventions in other C-family programming languages. if the condition (i.e. the second statement in the parentheses) is true, the code block –statements between the “{“ and “}” characters– of the for loop is executed. Then, the third statement between the parentheses is executed. After these steps, the condition is checked again. If it is true, the program continues with the same process as it is described above. When the condition is false, the program exits the loop. Having a for loop in Key increases writability in a considerable way because it avoids the usage of the while statement with some additional initial conditions and statements and expands the developers arsenal for implementing their designs into code.

- **Math statements:** Math statements in Key consist of several arithmetic operations. These operations are addition, subtraction, multiplication, division, modulo and exponentiation (power). Usage of those operations are quite similar with the arithmetic expressions that are used in Mathematics. This increases readability for developers. However, in Key, power operation is represented with symbols “**”. Although this symbol is used in many programming languages, it is not similar to “regular” mathematical expressions. This symbol was chosen since having a superscripted number in a code is not easy to write. So, the power operator reduces readability while increasing the writability (even though it decreases readability, we believe the benefits of increased writability outweighs the shortcomings of decreased readability in this case). The design of the precedence and associativity of the arithmetic operators are similar to normal mathematical expressions. This means multiplication, division, power and modulo operations have the same precedence and their precedences are higher than precedences of addition and subtraction operators, which have the same precedence. In order to correctly define and identify precedences of operators in Key, we grouped operators with the same precedences together and defined separate rules for them. This enables the operators with higher precedences to be lower in the parse tree, essentially making them execute first (with correct precedence). In addition to these, subtraction, division, multiplication, addition, and modulo operators are designed as left associative with left recursive definitions; and the power operator is designed as right associative with a right recursive definition.

- **Boolean list:** Boolean list is used throughout the program to provide conditional expressions. For example, it is used in all of the loops to check whether the loop execution should be continued or the program should break out of the loop. Also, boolean list is used in the conditional part of the if statements. Furthermore, a boolean list can contain a boolean literal, which is true or false, a boolean variable, and results of; a string or number comparison expression, as well as the combinations of these with logic operators and the negation operator. In addition to those, there are no string comparisons with numbers in Key. That restriction increases reliability of the language since it prevents programmers from making unintuitive and undefined comparisons.

- **Primitive functions:** Primitive functions in Key are; getTimestamp(), getTemperature(), getHumidity(), getAirPressure(), getAirQuality(), getLight(), and getSoundLevel(<frequency>). The “get” keyword in the beginning of the primitive functions signify that the “thing” the reader is looking at is an “accessing” function, and the rest of the words provide the context of the accessed data. These functions return the data from the relevant sensors (the getSoundLevel(<frequency>) primitive function accepts a double value as its frequency parameter and returns the data relevant to the given frequency parameter), and then this data could be used in math expressions, assignment expressions and comparison expressions.

- **Variable types:** Variable types in Key are; boolean, int, double, string, char, connection, and timestamp. We chose the names of the variable types with significant inspiration from the C-family programming languages, which increases readability and writability through similarity. The variable type “connection” is associated with the requirement of a URL connection. We decided to create a special variable type for “timestamp” which stores the number of seconds elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970. This variable type is named “timestamp” and not something as “long” as we decided for the purposes of our programming language’s domain (IoT devices), variable name “timestamp” increases readability for developers (even though it decreases writability, we believe the benefits of increased readability outweighs the shortcomings of decreased writability in this case). Also, type checking provides reliability by guiding programmers to use the correct type of variables and parameters.

- **Function calls and declarations:** In Key, function declarations are as follows: function <return_type> <function_name>(<parameter_list>) { <code_block> }. Return type can be any variable type or void. Therefore, not returning anything and just “return;” is accepted by Key. Moreover, the function naming rule is the same as the identifier naming rule. Furthermore, parameter list can have zero or more parameters and while declaring a function, variable types should be given in the parameter list such as “int myVar”. Also, in order to call a function, programmers must write the function name and the necessary parameters without their type. Key also allows programmers to do computations and assignments with function calls. We chose this syntax for function declarations and calls since it is familiar with C-family programming languages (which increases readability and writability through similarity). We added “function” at the beginning of the declaration to emphasize that it is a function declaration statement and to add a unique touch to Key.

- **Increment/Decrement operators:** The increment and decrement operators in Key are defined as “++” and “--” respectively. We selected these symbols from the C-family programming languages, as this would provide similarity to the developers which increases readability and writability through similarity. The increment operator increases the value of a variable by 1 (numerical), while the decrement operator decreases the value of a variable by 1 (numerical). The operators can be used both to the left and to the right of the variables (i.e. ++var, var++, --var, var--). In other words, “++var” and “var++” “do the same thing” in Key (aside from memory management during the operation), similar to C-family programming languages.

- **Comments:** There are 2 available comment types in Key. The first one is the single-line comment which is indicated by “//”. The other comment type is multi-line comment and it’s syntax is as follows: /* comment */. We decided to use these symbols since it is similar to the C-family programming languages and therefore familiar to almost every programmer which increases readability and writability through similarity.

- **Identifiers:** In Key, variable and function names are defined similar to variable and function names in Java as this would provide similarity to the developers which increases readability and writability through similarity. Identifiers in Key start with a letter, and then can be followed by letters, digits, the “_” or “$” symbols.

- **Reserved words:** The reserved words of Key are; main, if, for, while, do, else, function, return, boolean, int, string, char, connection, timestamp, double, true, false, getTimestamp, getTemperature, getHumidity, getAirPressure, getAirQuality, getLight, getSoundLevel, and switch. We selected our reserved words with significant inspiration from the C-family programming languages, as this would provide similarity to the developers which increases readability and writability through similarity. Furthermore, there are 10 switches defined in Key, from switch[0] to switch[9]. To access or modify a switch’s value, these definitions are used like the following “switch[0] = 1” or “int var = switch[0]”. There are no corresponding switches for numbers other than the range [0-9], in Key. Hence, this definition prevents programmers from using non-existing switches, which increases the reliability of Key.

- **Conventions:** In Key, conventions are similar to the C-family programming languages. Both the variable and function name conventions in Key are the “camelCase” form. Sticking to the conventions increases readability of the programs that are written in Key.

Part A - Language Design (BNF Description)

```
<program> ::= main() { <stmt_list_with_if> }
<stmt_list_with_if> ::= <stmt_with_if>
                      | <stmt_with_if> <stmt_list_with_if>

<stmt> ::=  <assignment_expr>
          | <declaration_expr>
          | <loop_expr>
          | <increment_expr>
          | <decrement_expr>
          | <comment_singleline_stmt>
          | <comment_multiline_stmt>
          | <return_stmt>

<stmt_list> ::= <stmt> | <stmt> <stmt_list>
<stmt_with_if> ::= <stmt> | <if_stmt>

<if_stmt> ::= <matched> | <unmatched>

<matched> ::= if ( <boolean_list> ) { <matched> } else { <matched> }
            | <stmt_list> <matched>
            | <stmt_list>

<unmatched> ::=
              if ( <boolean_list> ) { <stmt_list_with_if> }
            | if ( <boolean_list> ) { <matched> } else { <unmatched> }
            | if ( <boolean_list> ) { <unmatched> } else { <unmatched> }
            | if ( <boolean_list> ) { <unmatched> } else { <matched> }
            | <stmt_list> <unmatched>
            | <stmt_list>

<boolean_list> ::= <boolean_expr>
                 | <boolean_expr> <logic_operator> <boolean_list>

<boolean_expr> ::= <negation_operator> <boolean_expr>
                 | <num_compr_expr>
                 | <string_compr_expr>
                 | <boolean_literal>
                 | <boolean_var>

<num_compr_expr> ::=
                   <num_comparable> <comparator_operator> <num_comparable>
```

```
<num_comparable> ::= <int_var>
                    | <char_var>
                    | <num_const>
                    | <double_const>
                    | <math_stmt>
                    | <double_var>

<string_compr_expr> ::= <string_var> <equal_or_not_operator> <string_var>

<logic_operator> ::= <and_operator> | <or_operator>

<comperator_operator> ::= <great_operator>
                    | <less_operator>
                    | <great_or_equal_operator>
                    | <less_or_equal_operator>
                    | <equal_or_not_operator>

<arithmetic_operator> ::= <add_operator>
                    | <subtract_operator>
                    | <mult_operator>
                    | <div_operator>
                    | <modulo_operator>
                    | <power_operator>

<equal_or_not_operator> ::= <equality_operator>
                      | <not_equality_operator>

<negation_operator> ::= !
<and_operator> ::= &&
<or_operator> ::= ||
<great_operator> ::= >
<less_operator> ::= <
<great_or_equal_operator> ::= >=
<less_or_equal_operator> ::= <=
<equality_operator> ::= ==
<not_equality_operator> ::= !=

<declaration_expr> ::= <var_type> <var_id>;
                      | <var_type> <assignment_expr>

<var_type> ::= <int_type>
              | <char_type>
              | <string_type>
              | <connection_type>
              | <boolean_type>
              | <timestamp_type>
              | <double_type>
```

```
<boolean_type> ::= boolean
<int_type> ::= int
<string_type> ::= string
<connection_type> ::= connection
<timestamp_type> ::= timestamp
<double_type> ::= double
<char_type> ::= char

<num_const> ::= <digit>
              | <digit> <num_const>

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

<int_var> ::= <var_id> | <fcn_call>
<char_var> ::= <var_id> | <fcn_call>
<string_var> ::= <var_id> | <fcn_call>
<connection_var> ::= <var_id> | <fcn_call>
<timestamp_var> ::= <var_id> | <fcn_call>
<boolean_var> ::= <var_id> | <fcn_call>
<double_var> ::= <var_id> | <fcn_call>
<boolean_literal> ::= true | false

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

<str_stmt> = “<acceptable_str>”
             | “<acceptable_str>” + <str_stmt>
             | <str_stmt> + “<acceptable_str>”
             | <str_stmt> + <var_id>
             | <var_id> + <str_stmt>
             | <var_id>

<acceptable_str> = <all_chars>
                 | <all_chars> <acceptable_str>
                 | <empty>

<all_chars> ::= <alphanumeric_or_symbol>
              | ! | \n | ( | ) | \’ | + | - | / | * | < | > | { | } | ^
              | # | \t | & | % | \\ | ? |  /* space */ | . | [ | ] | ~
              | : | ; | = | @
```

```
<alphanumeric_or_symbol> ::= <letter>
                           | <digit>
                           | _
                           | $

<assignment_expr> ::= <assignment_expr_no_sc>;
<assignment_expr_no_sc> ::= <var_id> = <assignment_operand>

<assignment_operand> ::= <math_stmt>
                       | <str_stmt>
                       | <var_id>
                       | <boolean_literal>
                       | <boolean_list>
                       | <primitive_functions>
                       | <establish_connection>
                       | <switch>
                       | <fcn_call>

<comment_singleline_stmt> ::= // <acceptable_str>
<comment_multiline_stmt> :: /* <acceptable_str> */

<loop_expr> ::= <while_loop>
              | <for_loop>
              | <do_while_loop>

<increment_expr> ::= <increment_operation>;
<increment_operation> ::= <var_id>++
                        | ++<var_id>

<decrement_expr> ::= <decrement_operation>;
<decrement_operation> ::= <var_id>--
                        | --<var_id>

<while_loop> ::= while ( <boolean_list> ) { <stmt_list_with_if> }

<do_while_loop> ::= do { <stmt_list_with_if> } while ( <boolean_list> );

<math_stmt> ::= <math_stmt> <add_operator> <term>
              | <math_stmt> <subtract_operator> <term>
              | <term>

<term> ::= <term> <mult_operator> <factor>
         | <term> <div_operator> <factor>
         | <term> <modulo_operator> <factor>
         | <factor> <power_operator> <term>
         | <factor>
```

```
<factor> ::= <num_const>
             | <double_const>
             | <var_id>
             | <fcn_call>
             | <primitive_functions>
             | (<math_stmt>)

<add_operator> ::= +
<subtract_operator> ::= -
<mult_operator> ::= *
<div_operator> ::= /
<modulo_operator> ::= %
<power_operator> ::= **

<for_loop> ::= for ( <for_expr> ) { <stmt_list_with_if> }
<for_expr> ::= <for_init>; <boolean_list>; <for_update>
<for_init> ::= <var_type> <assignment_expr_no_sc>
               | <assignment_expr_no_sc>

<for_update> ::= <math_stmt>
                 | <assignment_expr_no_sc>
                 | <decrement_operation>
                 | <increment_operation>

<define_fcn> ::=
   <fcn_return_type> <fcn_name>( <param_list> ) { <stmt_list_with_if> }

<param_list> ::= <empty>
                 | <var_type> <var_id>
                 | <var_type> <var_id>, <param_list>

<param_list_no_type> ::= <empty>
                        | <var_id>
                        | <var_id>, <param_list_no_type>


<fcn_name> ::= <var_id>
<fcn_return_type> ::= <var_type>
                      | void

<fcn_call> ::= <fcn_name>( <param_list_no_type> )
<fcn_call_expr> ::= <fcn_call>;
```

```
<primitive_function_expr> ::= <primitive_functions>;
<primitive_functions>::= <get_timestamp>
                        | <get_temperature>
                        | <get_humidity>
                        | <get_air_pressure>
                        | <get_air_quality>
                        | <get_light>
                        | <get_sound_level>

<get_timestamp> ::= getTimestamp()
<get_temperature> ::= getTemperature()
<get_humidity> ::= getHumidity()
<get_air_pressure> ::= getAirPressure()
<get_air_quality> ::= getAirQuality()
<get_light> ::= getLight()
<get_sound_level> ::= getSoundLevel(<frequency>)

<frequency> ::= <var_id>
                | <double_const>

<establish_connection_expr> ::= <establish_connection>;
<establish_connection> ::= establishConnection(<str_stmt>)

<send_data_expr> ::= <send_data>;
<send_data> ::= <connection_var> . send(<send_item>)
                | <establish_connection> . send(<send_item>)

<receive_data_expr> ::= <receive_data>;
<receive_data> ::= <connection_var> . read()
                   | <establish_connection> . read()

<send_item> ::= <int_var>
                | <num_const>

<switch> ::= switch[<switch_no>]
<switch_no> ::= <digit>
<switch_stmt> ::= <switch> = <boolean_literal>;
                  | <switch> = <switch>;
                  | <switch> = <negation_op> <switch>;

<return_expr> ::= <return_stmt>;
<return_stmt> ::= return
                  | return <factor>
                  | return <num_comparable>
                  | return <str_stmt>

<empty> ::= /* empty */
```