CS 315

Programming Languages

Project 2

Group 10

Tutku Language

2022-2023 Fall Semester

Group Members:

Kutay Şenyiğit 21902377 Section 3

Alperen Utku Yalçın 22002187 Section 3

Ömer Oktay Gültekin 21901413 Section 3

Language Name: Tutku

Part A – BNF Description of the Language

Note: Changes from the previous report are indicated by red font color.

1) Constants, Types and Auxiliary Definitions

<null>	::=
<terminate_op>	::= ;
<assign_op>	::= =
<add_op>	::= +
<sub_op>	::= -
<mult_op>	::= *
<div_op>	::= /
<lp>	::= (
<rp>	::=)
<quote>	::= "
<dot>	::= .
<comment_symbol>	::= //
<lb>	::= {
<rb>	::= }
<separator>	::= ,
<rel_op>	::= < | > | <= | >= | == | !=

<or_op> ::= ||;

<and_op> ::= &&

<logic_op> ::= <or_op> | <and_op>

<char_letter> ::= [a-zA-Z]

<digit> ::= [0-9]

<num_literal> ::= <int_literal> | <float_literal>

<int_literal> ::= <digit> <int_literal> | <digit>

<float_literal> ::= <int_literal> <dot> <int_literal> | <int_literal>

<string_literal> ::= <quote> <string> <quote>

<string> ::= <string> <escape_op> <quote> <text>|<text>

<text> ::= <char_letter> <text> | <digit> <text> | <symbols> <text> | <null>

<bool_literal> ::= true | false

<var_name> ::= <char_letter> <var_name> | <var_name> <digit> | <char_letter>

<var_type> ::= <primitive_var_type> | <conn_type>

<primitive_var_type> ::= <num_var_type> | string | bool

<num_var_type> ::= int | float

<conn_type> ::= conn

<def_keyword> ::= fun

<return_keyword> ::= return

<inp_keyword> ::= input

<out_keyword> ::= print

2) Program Entry

<Tutku> ::= <stmts>

<stmts> ::= <null> | <stmt> <stmts>

3) Statements

<stmt> ::= <safe_stmt> | <unsafe_stmt>

<safe_stmt> ::= <terminated_stmt> <terminate_op> |
<non_terminated_stmt>

<unsafe_stmt> ::= <fun_def_stmt>

<terminated_stmt> ::= <assign_stmt> | <declare_stmt> | <fun_call> |
<io_stmt>

<non_terminated_stmt> ::= <comment_stmt> | <cond_stmt> | <loop_stmt>

<safe_stmts> ::= <null> | <safe_stmts> <safe_stmt>

4) Assignments

<assign_stmt> ::= <declare_stmt> <assign_op> <assignable>

| <var_name> <assign_op> <assignable>

<assignable> ::= <expression> | <string_literal> | <fun_call>

5) Expressions

<expression> ::= <arithmetic_expression> | <bool_expression>

| <lb><arithmetic_expression><rb>
| <lb><bool_expression><rb>

<arithmetic_expression> ::= <arithmetic_expression> <add_op> <term_var_name> |
<arithmetic_expression><sub_op><term_var_name> |
<funct_var_name><add_op><term_var_name> |
<funct_var_name><sub_op><term_var_name> | <term>

<funct_var_name> ::= <var_name> | <fun_call> | <string_literal>

<term_var_name> ::= <var_name> | <string_literal> | <term> | <fun_call>

<term> ::= <term> <higher_op> <term_type> | <type_needed_op>
<higher_op> <term_type> | <term> <higher_op>
<type_needed_op> | <type_needed_op> <higher_op>
<type_needed_op> | <term_type>

<higher_op> ::= <mult_op> | <div_op>

<type_needed_op> ::= <fun_call> | <var_name>

<term_type> ::= <num_literal> | <lp> <arithmetic_expression><rp> |
<input_stmt>

<bool_expression> ::= <logical_expression>

<relational_expression> ::= <relational_expression> <rel_op> <num_data> | <num_data> <rel_op><num_data>

<rel_op> ::= <less_equal_op> | <less_than_op> | <greater_equal_op> | <greater_than_op> | <equals_op> | <not_equals_op>

<num_data> ::= <var_name> | <arithmetic_expression> | <fun_call>

<logical_expression> ::= <logical_expression> <or_op> <logic_term> | <logical_expression> <or_op> <fun_call> | <fun_call> <or_op> <logic_term> | <logic_term>

<logic_term> ::= <logic_term> <and_op> <bool_term> | <logic_term> <and_op> <fun_call> | <fun_call> <and_op> <bool_term> | <bool_term

<bool_term> ::= <bool_data> | <lp> <bool_data> <rp> | <lp> <fun_call> <rp>

<bool_data> ::= <relational_expression> | <bool_literal> | <var_name>

6) Comments

<comment_stmt> ::= <comment_symbol> <string>

7) Functions

<fun_def_stmt> ::= <def_keyword> <fun_name> <lp> <param_list> <rp> <lb>

<safe_stmts > <fun_return> <rb>

<fun_name> ::= <var_name>

<fun_return> ::= <null> | <return_keyword> <returnable> <terminate_op>

<returnable> ::= <expression> | <fun_call> | <string_literal>

<param_list> ::= <null> | <var_type> <var_name> <params>

<params> ::= <null> | <separator> <var_type> <var_name> <params>

<fun_call> ::= <fun_name> <lp> <arg_list> <rp> | <read_sensor> | <read_timer> | <connect_url> | <get_data_from_conn>

| <send_data_by_conn>
| <change_switch_state_and_control_actuator>

<arg_list> ::= <null> | <callable> <args>

<callable> ::= <returnable>

<args> ::= <null> | <separator> <callable> <args>

8) Primitive Built-in Functions

<read_sensor> ::= readTemp<lp><rp> | readHumidity<lp><rp> | readAirPressure<lp><rp> | readAirQuality<lp><rp> | readLightLevel<lp><rp> | readSoundLevel<lp><rp>

<read_timer> ::= getTimestamp<lp><rp>

<connect_url> ::= connectURL<lp><var_name><rp> | connectURL<lp><string_literal><rp>

<change_switch_state_and_control_actuator> ::= changeSwitchStateAndControlActuator<lp><switch_num><rp>

<switch_num> ::= <digit> | <var_name> | <fun_call>

<get_data_from_conn> ::= <conn><dot>getDataFromConn<lp><rp>

<send_data_by_conn> ::= <conn><dot>sendDataByConn<lp><callable><rp>

<conn> ::= <connect_url> | <conn_name>

<conn_name> ::= <var_name>

9) IO Statements

<io_stmt> ::= <input_stmt> | <output_stmt>

<input_stmt> ::= <inp_keyword><lp><rp>

<output_stmt> ::= <out_keyword><lp><callable><rp> | <out_keyword> <lp> <rp>

10) Loop Statements

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

<while_stmt> ::= while <lp> <bool_expression> <rp> <lb> <safe_stmts> <rb> | while <lp> <bool_expression> <rp> <safe_stmt>

<for_base> ::= for <lp> <assign_and_null> <terminate_op> <bool_and_null> <terminate_op> <arithmetic_and_null> <rp>

<assign_and_null> ::= <assign_stmt> | <null>

<bool_and_null> ::= <logical_expression> | <null>

<arithmetic_and_null> ::= <var_name> <assign_op> <arithmetic_expression> | <null>

<for_stmt> ::= <for_base> <lb> <safe_stmts> <rb> | <for_base> <safe_stmt>

11) Conditional Statements

<if_stmt> ::= <if_base> <lb> <safe_stmts> <rb>

<cond_stmt> ::= <if_stmt> | <if_stmt> <else> <lb> <safe_stmts> <rb> | <if_stmt> <else> <safe_stmt>

<if_base> ::= if <lp> <bool_expression> <rp> | <if> <lp> <fun_call> <rp>

12) Declaration Statement

<declare_stmt> ::= <var_type> <var_name>

BNF Elements Descriptions

<null>: Represents empty value

<terminate_op> : Language choice for specifying end of some statements

<assign_op>, <add_op>, <sub_op>, <mult_op>, <div_op> : Universally accepted math operators

<lp> <rp> : Left and right parentheses

<quote> : Language choice for specifying string values

<dot> : Language choice for specifying function call from connection type

<comment_symbol> : Language choice for the start of comments

<lb>, <rb> : Left and right curly brackets

<separator> : Language choice for separator of parameters

<rel_op> : Universally accepted relational operators

<logic_op> : Universally accepted logic operators

<char_letter> : Allowed char letters to use

<digit> : Allowed digits to use

<num_literal> : Allowed numerical literals

<string_literal> : String syntax of the language

<string> : Specifies an escape sequence in case string starting symbol is in the string

<text> : Allow some characters to be in the string, the same list as python

<int_literal> : Defines decimal numbers

<float_literal> : Defines floating point numbers

<bool_literal> : Defines type of truth values

<var_name> : Defines allowed variable names to only be consists of letters and digits except for the names starting with digits

<var_type> : Defines variable types

<primitive_var_type> : Defines data types of the language

<num_var_type> : Defines numerical data types of the language

<conn_type> : Defines a way to connect an Internet

<def_keyword> : Function declaration keyword

<return_keyword>: Return keyword

<inp_keyword>: Input keyword

<out_keyword>: Output keyword

<Tutku> : Defines program entry point

<stmts> : Defines there can be any number of statements in the program

<stmt> : Defines statements to be either a safe or unsafe statements.

<safe_stmt> : Defines some statements as safe to use between curly braces and divide them into the statements depending on the need of termination, like the Java does not allow the programmers to put terminate operator after curly braces

<unsafe_stmt> : Defines statements that cannot be in between curly braces

<terminated_stmt> : Defines statements that need to be terminated

<non_terminated_stmt> : Defines non-terminated statements

<safe_stmts> : Defines any number of safe statements which will be used between curly braces later

<assign_stmt> : Defines assignment syntax of the language. The followings are valid:

string text = “text”;, text = “text”;. The language automatically infers the type of the latter one.

<assignable> : Defines assignable values

<expression> : Defines expressions arithmetic and boolean expressions

<aritmetic_expression> : Defines low priority operators

<funct_var_name> : Allows variable names, function calls and strings to use in arithmetic expressions. (String is for using print(“text1” + “text2”) capability)

<term_var_name> : Allows variable names, function calls terms and strings to use in arithmetic expressions’ right side. (String is for using print(“text1” + “text2”) capability)

<term> : Defines high priority operators

<higher_op> : multiplication and division operations

<type_needed_op> : Defines types that need at least one operator to use in arithmetic expressions

<term_type> : Defines values that can be an operand of an arithmetic expressions

<bool_expression> : Defines the ways that result in bool values

<relational_expression> : Defines relational expressions on numerical values

<num_data>: Defines the ways that result in numerical values

<logical_expression> : Defines logical expressions

<logic_term> : Defines rules for high priority operator (and) in logical expressions

<bool_term> : Defines parenthesis version of the bool data. EX: (4 + 1)

<bool_data>: Defines all possible operands of the logical expression

<comment_stmt> : Defines comment syntax as: // comment

<fun_def_stmt> : Defines function definition syntax as follows: fun funName(params){any number of safe statements}. Note that function return type is inferred.

<fun_name>: Defines function name using the same rules that variable name uses

<fun_return> : Function can either return anything or returnable

<returnable> : Returnable can be only variables, other functions calls, literals and expressions.

<param_list> : Defines parameters list as any number of parameters which will be variable type and variable name format.

<params> : Defines a rule that if the parameter list will have more than one element, then the separation symbol “,” is required.

<fun_call> : Defines function call as either custom functions or builtin functions. Example of custom function call: funName(arg1, arg2);

<arg_list> : Defines argument list of functions can consists of any number of arguments

<callable> : Defines arguments types the same as returnable types.

<args> : Defines a rule that if the argument list will have more than one element, then the separation symbol “,” is required.

<read_sensor> : Definition for built-in methods that read sensors. More sensors can be added easily to this rule if needed

<read_timer> : Definition for reading the timestamp from timer

<connect_url> : Definition to connect to the given URL which is given as a string literal or variable that holds a string value. Note that BNF is context-free grammer and if the string is valid URL or not is not the concern of the language grammer. Even if the grammer checks whether the beginning of the URL starts with https, there is always a possibility that given URL may not corresponds to a valid website. It is acceptable to leave this to the check algorithm implemented by the programmer.

<change_switch_state_and_control_actuator>: A built-in method to change a given switch’s state and control actuator that is connected to the given switch. This method will turn on the given switch if it is off or turn off the given switch if it is on. Switch numbers can be only digits since there are 10 different switches. Then this built-in method will call another built-in method to control the actuator by given switch number. .

Note that one-to-one assignment of switches and actuators is used. For example, switch 9 controls actuator 9. This method will turn on the given actuator if it is off or turn off the given switch if it is on. This sub-built-in method is necessary since there may be different components of the system that needs to be activated when some of the actuators are on or off, like transaction in SQL. If the state of some of the components fails to be activated or deactivated, all components must be going back to their initial states. Also, the method to control actuators need not to be available for use by the programmer since actuators are only controlled by switches and the actuators need to be turned on or off as soon as the corresponding switch is turned on or off. Example usage for switch 8 is currently off and programmer wants to turn on: changeSwitchStateAndControlActuator(8); => In success, switch 8 is currently on, actuator 8 is currently on and all corresponding components of the system are currently on. In failure of any step, switch 8 is currently off even if the error is about turning on the actuator, actuator 8 is currently off and all corresponding components of the system are currently on.

<get_data_from_conn> : Defines the way to get integer data, which is either a direct literal or value of int type variable, from the internet by an instance of the connection type.

<send_data_by_conn> : Defines the way to send integer , which is either a direct literal or value of int type variable, to the internet by an instance of the connection type.

<conn> : Defines possible ways that are either defining connection instance beforehand or use the definition directly to use that method.

<conn_name> : Defines a rule of the connection name the same as the rule of the variable names.

<io_stmt> : Defines possible input output statements

<input_stmt> : Defines input statement. The usage: input();

<output_stmt> : Defines output statement. The sample usage: print(130);

<loop_stmt> : Defines possible ways to use loop statements.

<while_stmt> : Defines the syntax of the while statement. Example usage of the while statement: while(True){any number of safe statements}, while(15 < 20)one safe statement;

<for_base> : Defines a root of the for loop. Example usages of the for root: for(;;), for(int a = 10; a < 15; a = a + 15), for(;;c = c + d)

<assign_and_null> : Defines possible assignment statements for for loop

<bool_and_null> : Defines possible bool statements for for loop

<arithmetic_and_null> : Defines possible arithmetic statements for for loop

<for_stmt> : Defines the syntax of the for loop. Example usage of the for loop: for(;;;){any number of safe statements}, for(;;;) safe statement;

<cond_stmt> : Defines possible conditions. Note that this terminal taking care of the if and if else statements.

<if_base> : Defines if root. Example usages of the if root: if(true), if(15 < 25)

<if_stmt> : Defines classic if statement

<declare_stmt>: Defines a rule to declare a variable of predefined types. Example usage: int intVar;, string stringVar;

TUTKU Language Specifics

1) About

Tutku language is designed to have primary imperative language features and built-in features that allow the programmer to read sensors, control actuators, and communicate with any URL via Internet, making the language a strong opponent to any other IoT language. The language tries to combine the best parts of Java, Python, and Kotlin programming languages. However, it is mainly based on Java programming language as it inherits the syntax of loops, conditional statements, variable declaration, and line terminators from Java. Kotlin language inspires Tutku to have null-safety features, and Tutku language tries to minimize the usage of null values. Additionally, the Tutku language uses Python's IO statements syntax for cleaner syntax. Currently, the Tutku language does not have OOP and library support. However, the simplicity of the language is enough for the current task and makes the language easier to learn for those who know the programming basics.

2) Readability

The Tutku language aims to be very readable, even for those that do not know the language. The language does not have any entry method; thus, programmers can write their codes directly starting from the first line, just like Python, which is very intuitive. Additionally, the language has two types of loops and conditional statements, consistent variable and function declarations, one type of function call, intuitively named built-in primitive functions, and a line terminator. Also, the language has safe and unsafe statements which do not allow the unsafe statements between any curly

brackets that otherwise result in deleting the unsafe statements from memory by the garbage collector of the language. That feature prevents users from confusion about why they could not use the functions declared between any curly parenthesis beforehand.

3) Writability

Tutku tries to minimize the code written in several ways. Firstly, the sensor data is automatically converted to the int value; there is no sensor primitive type. Secondly, all primitive types have default constants that variable declaration without assignment directly links that variable to the memory place of the constant of that type until the programmer changes the value of that variable; in which case, Tutku automatically creates another place for that variable in the memory. The automatic assignment is practical when the programmer wants to use that type's constant value but shifts some responsibility to the programmer. Thirdly, there are two different types of loops that the programmer can choose. Fourthly, Tutku automatically infers the functions' return value, allowing a simple, shorter syntax for the function definitions. Fifthly, the programmer does not need to put curly braces in loops and conditional statements if one line of code will be inserted between them. Lastly, there are no different types for decimal and floating-point numbers; int and float type automatically shrinks or extends when needed.

4) Reliability

Reliability is a top priority for Tutku since it is designed for IoT devices that make the language mostly deal with low-level operations like reading sensor data, controlling actuators, and communicating through the internet, possibly with other nodes. The language does not allow null values for the primitive types; in other words, the programmer can not initiate a primitive type with a null value; s/he should assign a valid value.Additionally, unsafe statements can not be written between curly braces, preventing the exception of reaching a memory place that has already been deleted. Also, Tutku enforces strict syntax rules that make the code less error prone.