CS 315 PROJECT 2

Parser for a Programming Language for IoT Devices

“KAPIBARO”

Instructor: Karani Kardaş

Section: 3

Team 32

Yağız Özkarahan 22002276

Hazal Aksu 21703841

Mostafa Higazy 22001062

BNF

PROGRAM

<program> ::= BEGIN_PROGRAM <main> END_PROGRAM

<main> ::= <statements>

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

<statement> ::= <expression> | <assignment_exp>| <loop> | <function> | <comment> |
<if_stmt> | <print_stmt> | <scan_stmt>| <idntf_definition> | <built_in_function>

TYPES

<type> ::= <number_types>

|<null>

|<bool>

|<str>

|<letter>

|<void>

<type_name> ::= INTEGER | DOUBLE | NULL_VALUE | BOOLEAN | STRING | LETTER

|VOID

<bool> ::= TRUE | FALSE

<null> ::= NULL_VALUE

<void> ::= VOID

<letter> ::= LETTER_VALUE

<str> ::= STRING_VALUE

<int> ::= <signed_int> | <unsigned_int>

<unsigned_int> ::= UNSIGNED_INT

<signed_int> ::= SIGNED_INT

<double> ::= <signed_double> | <unsigned_double>

<signed_double> ::= SIGNED_DOUBLE

<unsigned_double> ::= UNSIGNED_DOUBLE

<comma> ::= ,

<dot> ::= .

IDENTIFIERS

<idntf_definition> ::= <type_name> <idntf>

<idntf> ::= IDENTIFIER

ARITHMETIC OPERATORS

<higher_precedence> ::= <multiply_opr> | <division_opr>|<modulus_opr>

<lower_precedence> ::= <addition_opr> | <substraction_opr>

<multiply_opr> ::= *

<division_opr> ::= /

<addition_opr> ::= +

<subtraction_opr> ::= -

<modulus_opr> ::= %

<assign_opr> ::= =

<LP> ::= (

<RP> ::=)

<LCURLY> ::= {

<RCURLY> ::= }

EXPRESSIONS

<expression> :== <logical_exp>

| <increment_exp>

| <subtract_exp>

| <arithmetic_exp>

<increment_exp> :== <int> ADD_ONE | <double> ADD_ONE | <idntf> ADD_ONE

<subtract_exp> :== <int> SUB_ONE | <double> SUB_ONE | <idntf> SUB_ONE

<assignment_exp> :== <idntf> <assign_opr> <expression>

| <idntf> <assign_opr> <type>

<arithmetic_exp> ::= <arithmetic_exp> <lower_precedence> <term>

| <term>

<term> ::= <term> <higher_precedence> <factor>

| <factor>

<factor> ::= <LP><arithmetic_exp><RP>

| <number_types>

| <LP><idntf><RP>

<logical_exp> ::= <logical_exp_operands> <compare_opr> <logical_exp_operands>

<logical_exp_operands> ::= number_types | <LP> expression <RP> | idntf

<number_types> ::= <int> | <double>

<compare_opr> ::= <not_equal_opr>

| <equal_opr>

| <gte_opr>

| <lte_opr>

| <gt_opr>

| <lt_opr>

| <and_opr>

| <or_opr>

<not_equal_opr> ::= !==!

<equal_opr> ::= ==

<gte_opr> ::= >=

<lte_opr> ::= <=

<gt_opr> ::= >

<lt_opr> ::= <

<and_opr> ::= AND

<or_opr> ::= OR

LOOPS

<loop> ::= <while_loop> | <for_loop>

<while_loop> ::= while <LP> <logical_exp> <RP> <LCURLY> <statements> <RCURLY>

<for_loop> ::= for <LP> <assignment_exp> ; <logical_exp> ; <for_exp> <RP> <LCURLY> <statements> <RCURLY>

<for_exp> ::= <increment_exp>

| <subtract_exp>

| <arithmetic_exp>

CONDITIONAL STATEMENTS

<if_stmt> ::= <single_if_stmt>| <if_else_stmt>

<single_if_stmt> ::= IF <LP> <logical_exp> <RP><LCURLY> <statements> <RCURLY>

<if_else_stmt> ::= IF <LP> <logical_exp> <RP> <LCURLY> <statements> <RCURLY> ELSE <LCURLY> statements <RCURLY>

FUNCTION CALLING AND DEFINITION

<function> ::= <function_definition> | <function_call>

<function_definition>::= <type_name> <idntf> <LP> <multiple_params_in_def> <RP> <LCURLY> <statements> <RCURLY>

|<type_name> <idntf> <LP> <idntf_definition> <RP> <LCURLY>

<statements> <RCURLY>

<multiple_params_in_def> ::= <idntf_definition> ; <multiple_params_in_def> | *empty*

function_call ::= <idntf> <LP> <params_list><RP>

| <idntf> <LP> <idntf>< RP>

<params_list>::= <idntf> ; <params_list> | *empty*

COMMENTS

<comment> ::= COMMENT

PRINTING AND SCANNING

<print_stmt> ::= print<LP><str><RP> | print<LP><idntf><RP>

<scan_stmt> ::= scan<LP><RP>

BUILT-IN FUNCTIONS

<built_in_function> ::= <readTemp>|<readHumidity>|<readPressure>|<readAirQuality>
|<readLight>|<readSound>|<readTime>|<setActuator>|<connect>|<send>|<receive>|
<set_actuator_func>

<readTemp> ::= readTemp <LP><components_list><RP>

<readHumidity> ::= readHumidity <LP><components_list><RP>

<readPressure> ::= readPressure <LP><components_list><RP>

<readAirQuality> ::= readAirQuality <LP><component_list><RP>

<readLight> ::= readLight <LP><component_list><RP>

<readSound> ::= readSound<LP><component_list><RP>

<readTime> ::= readTime <LP><component_list><RP>

<setActuator> ::= setActuator <LP><boolr><RP>

<connect> ::= connect <LP><str><RP>

<send> ::= send <LP><int><comma><str><RP>

<receive> ::= receive <LP><str><RP>

<components_list> ::= <component> ; <components_list> | *empty*

<component> ::= <tempComponent> | <humidityComponent> | <pressureComponent>

| <lightComponent> | <soundComponent> | <timeComponent>

<actuator> ::= motor | valve | hydraulic_cylinder | servo | solenoid | thermal_actuator |
linear_motor | pneumatic_actuator | stepper_motor | comb_drive

<set_actuator_func> ::= <actuator> <dot> <setActuator>

ADDITIONAL EXPLANATIONS

CONFLICTS:

Although many conflicts were cleared during implementation, the final version of the language contains one reduce/reduce conflict. This conflict happens during arithmetic operations, as both “type: number_types" and “factor: number_types” rules are reduced at the same time. This conflict is cleared by changing the “factor” rule to take the number in parentheses (or any other symbol of choice) to differentiate it from the “type” rule. However, doing so would mean that the user would need to write the numbers in parentheses (or another symbol) during arithmetic operations [for example: (9) + (7) * (52)] which would greatly reduce writability and readability. Since this conflict doesn’t cause any problems, errors or ambiguities during execution, we decided to leave it as it is, because letting the user write the numbers just by itself, without any other symbols, is extremely necessary for writability.

PROGRAM

<program> ::= BEGIN_PROGRAM <main> END_PROGRAM

<main> ::= <statements>

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

<statement> ::= <expression> | *<assignment_exp>*| *<loop>* | *<function>* | *<comment>* | *<if_stmt>* | *<print_stmt>* | *<scan_stmt>*| *<idntf_definition>*

This is how a programmer can start writing a program in our language. A program starts with a BEGIN_PROGRAM token, which is “begin{“ and ends with an END_PROGRAM token, which is }end. A program is composed of statements separated by semicolons.

TYPES

<type> ::= <number_types>

|*<null>*

|*<bool>*

|*<str>*

|*<letter>*

|*<void>*

<type_name> ::= INTEGER | *DOUBLE* | *NULL_VALUE* | *BOOLEAN* | *STRING* | *LETTER* | *VOID*

<bool> ::= TRUE | FALSE

<null> ::= NULL_VALUE

<void> ::= VOID

<letter> ::= LETTER_VALUE

<str> ::= STRING_VALUE

<int> ::= <signed_int> | <unsigned_int>

<unsigned_int> ::= UNSIGNED_INT

<signed_int> ::= SIGNED_INT

<double> ::= <signed_double> | <unsigned_double>

<signed_double> ::= SIGNED_DOUBLE

<unsigned_double> ::= UNSIGNED_DOUBLE

- These are the data types available in our language. The types are integer, double, null, boolean, string, letter and void. The null type exists for variables, while void only exists to be able to define a function that does not return anything. Both double and integers can have negative or positive signs. A string in our language is denoted in between two ~ characters.

IDENTIFIERS

<idntf_definition> ::= <type_name> <idntf>

<idntf> ::= IDENTIFIER

- An identifier in our language can be any collection of characters in our language. It could also have numbers as long as the numbers are after the characters or sandwiched between them.

ARITHMETIC OPERATORS

<higher_precedence> ::= <multiply_opr> | <division_opr>|<modulus_opr>

<lower_precedence> ::= <addition_opr> | <substraction_opr>

<multiply_opr> ::= *

<division_opr> ::= /

<addition_opr> ::= +

<subtraction_opr> ::= -

<modulus_opr> ::= %

<assign_opr> ::= =

- The multiplication, division, and modulus operators have higher precedence than addition and subtraction operators and thus will be performed before them if the arithmetic operations are close to each other.

EXPRESSIONS

<expression> :== <logical_exp>

| <increment_exp>

| <subtract_exp>

| <arithmetic_exp>

<increment_exp> :== <int> ADD_ONE | <double> ADD_ONE | <idntf> ADD_ONE

<subtract_exp> :== <int> SUB_ONE | <double> SUB_ONE | <idntf> SUB_ONE

<assignment_exp> :== <idntf> <assign_opr> <expression>

| <idntf> <assign_opr> <type>

<arithmetic_exp> ::= <arithmetic_exp> <lower_precedence> <term>

| <term>

<term> ::= <term> <higher_precedence> <factor>

| <factor>

<factor> ::= (<arithmetic_exp>)

| <number_types>

| (<idntf>)

<logical_exp> ::= <logical_exp_operands> <compare_opr> <logical_exp_operands>

<logical_exp_operands> ::= number_types | (expression) | idntf

<number_types> ::= <int> | <double>

<compare_opr> ::= <not_equal_opr>

| <equal_opr>

| *<gte_opr>*

| *<lte_opr>*

| *<gt_opr>*

| *<lt_opr>*

| *<and_opr>*

| *<or_opr>*

<not_equal_opr> ::= !==!

<equal_opr> ::= ==

<gte_opr> ::= >=

<lte_opr> ::= <=

<gt_opr> ::= >

<lt_opr> ::= <

<and_opr> ::= AND

<or_opr> ::= OR

- An assignment statement in our language is an expression or a type assigned to an identifier.
- The increment expression adds a +1 to an integer or double variable or to an identifier. The subtract expression subtracts -1 from the same types.
- The arithmetic expressions are defined in accordance with the operator precedence described above in the Arithmetic Operators bit.
- If an identifier is used in an arithmetic expression, it needs to be between parentheses. This was done to remove conflicts that were present in the yacc.
- Logical expression operands can be either a number, an identifier or an expression. If it's an expression, it should be between parentheses.

LOOPS

<loop> ::= <while_loop> | <for_loop>

<while_loop> ::= while <lp> <logical_exp> <rp> <lcurly> <statements> <rcurly>

<for_loop> ::= for <lp> <assignment_exp> ; <logical_exp> ; <for_exp> <rp> <lcurly> <statements> <rcurly>

<for_exp> ::= <increment_exp>

| *<subtract_exp>*

| *<arithmetic_exp>*

- The loops in our language are the regular while and for loops. The for loop lists its condition parameters separated by semicolons.

CONDITIONAL STATEMENTS

<if_stmt> ::= <single_if_stmt>| <if_else_stmt>

<single_if_stmt> ::= IF LP <logical_exp> RP LCURLY <statements> RCURLY

<if_else_stmt> ::= IF LP <logical_exp> RP LCURLY <statements> RCURLY ELSE LCURLY statements RCURLY

- Our conditional statements separate the statements from the if/else keywords by curly brackets. This means that chaining if/else's such as "if if if else else" is not possible; but if-else statements can be nested inside the curly brackets to allow nested conditions and "elseif" statements. Examples can be seen in the test program.

FUNCTION CALLING AND DEFINITION

<function> ::= <function_definition> | *<function_call>*

<function_definition>::= <type_name> <idntf> <LP> <multiple_params_in_def> <RP> <LCURLY> <statements> <RCURLY>

|<type_name> <idntf> <LP> <idntf_definition> <RP> <LCURLY>

<statements> <RCURLY>

<multiple_params_in_def> ::= <idntf_definition> ; <multiple_params_in_def> | *empty*

function_call ::= <idntf> <LP> <params_list><RP>

| *<idntf> <LP> <idntf>< RP>*

<params_list>::= <idntf> ; <params_list> | *empty*

- Programmers can define functions in our language by specifying their return type and parameters. A function that does not take any parameters can also be defined. The parameters are separated by semicolons [such as function(param1;param2;param3;). The function call either takes parameters separated by semicolons or no parameters at all.

BUILT-IN FUNCTIONS

<readTemp> ::= readTemp <LP><components_list><RP>

<readHumidity> ::= readHumidity <LP><components_list><RP>

<readPressure> ::= readPressure <LP><components_list><RP>

<readAirQuality> ::= readAirQuality <LP><component_list><RP>

<readLight> ::= readLight <LP><component_list><RP>

<readSound> ::= readSound<LP><component_list><RP>

<readTime> ::= readTime <LP><component_list><RP>

- These are the main functions that interact with components. readTemp, readHumidity, readPressure, readAirQuality, readLight, readSound, and readTime are all functions that take in a list of components and read the value of the component/s they are responsible for.

<setActuator> ::= setActuator <LP><bool><RP>

- Changes the state of the actuator from on to off and vice versa.

<connect> ::= connect <LP><str><RP>

<send> ::= send <LP><int><comma><str><RP>

<receive> ::= receive <LP><str><RP>

- send, connect, and receive are all functions used to get and send values to URL's. send, and connect have return type bool, while receive returns an int if value is read or null if no value is read. A url is of type string.

<components_list> ::= <component> ; <components_list> | *empty*

<component> ::= <tempComponent> | *<humidityComponent>* | *<pressureComponent>*

| <lightComponent> | <soundComponent> | <timeComponent>

- A component can be thought of as the reading from a sensor. It consists of the digital interface linking the sensor and our language. A list of components can be used to pass through the values of multiple sensors to the functions: readTemp, readHumidity, readPressure, readAirQuality, readLight, readSound, and readTime.

<actuator> ::= motor | *valve* | *hydraulic_cylinder* | *servo* | *solenoid* | *thermal_actuator* | *linear_motor* | *pneumatic_actuator* | *stepper_motor* | *comb_drive*

- There are 10 actuators defined in our language covering the most popular use cases for IOT devices.

<set_actuator_func> ::= <actuator> <dot> <setActuator>

- This function sets any type of actuator by specifying which actuator it is setting and whether the actuator is being set on or off via the boolean variable.

DESCRIPTION OF NON-TRIVIAL TOKENS

- BEGIN_PROGRAM

  This denotes the keyword ‘begin’ which’ specifies the beginning of execution of a program.

- END_PROGRAM

  This denotes the keyword ‘end’ which specifies the end of execution of a program.

- COMMENT

  This denotes a comment. Comments are sandwiched between ## symbols and they end with a semicolon, since they are considered a statement. They can be multi-line or single-line. ## This is an example comment ##;

- FOR_LOOP

  This denotes the keyword ‘for’ which is used to create for loops.

- WHILE_LOOP

  This denotes the keyword ‘while’ which is used to create while loops.

- IF

  This denotes the keyword ‘if’ which is used to create the if conditional statement.

- ELSE

  This denotes the keyword ‘else’ which is used in if-else statements.

- PRINT

  This denotes the keyword ‘print’ which is used to print things to the console.

- SCAN

  This denotes the keyword ‘scan’ which is used to take input from the console.

- FUNCTION

  This denotes the keyword ‘function’ which is used to mark the beginning of a function definition.

- COMPONENT

  A component can be thought of as the reading from a sensor. It consists of the digital interface linking the sensor and our language. The sensors that interface with the component are air, temp, humidity, pressure, light, sound and time sensors.

- ACTUATOR

  An actuator can be turned on or off. It has 10 different types for different use cases. motor,valve,hydraulic cylinder,servo,solenoid,thermal actuator,inear motor,pneumatic actuator,stepper motor, and comb drive. These are some of the most prominent actuators used in IOT devices.

- CONNECT

  This denotes the connect function, which takes a URL address as input and returns a boolean value.

- SEND

  This denotes the send function, which takes an integer and a URL address for inputs and returns a boolean value.

- RECEIVE

  This denotes the receive function which takes a URL address as input and returns int value.

EVALUATION OF THE LANGUAGE

Readability, Writability & Reliability

Our language was made with readability in mind, and that will be evident to any programmer who wants to pick it up and write code in it. We have chosen a variety of construct names that are widely used in other popular languages such as Java, C++ etc. We have created this language to be understood by English speakers, and we believe any college student studying computer science or related subjects with an understanding of the English language could conceive what is written in a piece of code in KAPIBARO quite easily.

Our language also has easy and flexible writability. It almost resembles spaghetti code in that it can be written in a stream of consciousness type of way, with a limited set of functionalities provided, and there are no classes or objects to be worried about. The language is also created in a way that would make it easy to broaden it with new functionalities in the future, if needed.

While we chose to give the reliability of our language big importance in this project early on, we soon came to realize we would have to compromise parts of reliability in order to be able to make our language work as a whole. We implemented a primitive version of type checking for integer and double types in the yacc part of our project. We also wanted our language to have precedence in logical statements, however that caused one reduce/reduce conflict in the implementation, which was explained previously in the report.