CS 315 Project 2

Section: 2
Barkın Saday - 21902967
Utku Kurtulmuş - 21903025
Language Name: A++

Grammar of the Language in BNF Form

Program Start:

<program> → begin <stmts> end

<stmts> → <stmt> | <stmt> <stmts>

<stmt> → <matched> | <unmatched> | <non_if_stmt>;

<matched> → iff (<bool_ex>){<stms>} else{<stmts>}

<unmatched> → if(<bool_ex>){<stmts>}

<non_if_stmt> → <assign> | < func_call> | <return_stmt> | <loop>
| <static_func>

<assign> → <type> IDENTIFIER = <expr> | <chain> = <expr>

<return_stmt> → return <expr>

Expressions:

<chain> → IDENTIFIER | <call_array> | <chain>.IDENTIFIER
| <chain>.<call_array>

<expr> → <arith_expr> | TEXT || <object>
| <array> | <bool_ex> | <call_array>

<arith_expr> → <arith_expr> + <term>
| <arith_expr> - <term>
| <term>

<term> → <term> * <exp>
| <term> / <exp> | <exp>

<exp> → (<arith_expr) | IDENTIFIER | <constant_value> | <func_call>

<bool_ex> → <bool_ex> | <bool_term>
| <bool_term>

<bool_term> → <bool_term> & <bool_factor>
| <bool_factor>

<bool_factor> → !<bool_exp> | <bool_exp>

<bool_exp> → (<bool_ex>) | <bool_value>
| <arith_expr> <comparison> <arith_expr>

<call_array> → IDENTIFIER [<expr>]

Functions:

<func_call> → <chain>(<call_list>) |< chain>() | <built_in_func>

<static_func> → static <type> func IDENTIFIER(<decleration_list>){<stms>}
| static void func IDENTIFIER(<decleration_list>){<stmts>}
| static <type> func IDENTIFIER(){<stmts>}
| static void func IDENTIFIER(){<stmts>}

<object_func> → <type> func IDENTIFIER(<decleration_list>{<stmts>}
| void func IDENTIFIER(<decleration_list>){<stmts>}
| <type> func IDENTIFIER(){<stmts>}
| void func IDENTIFIER(){<stmts>}

<built_in_func> → <chain>.connectToURL(<call_list>)
| print(<call_list>) | getTime()
| <chain>.getInput()
| <chain>.sendInput(<call_list>)
| <chain>.length()
| this.getSensors()
| <chain>.getSensors()
| <chain>.getSensorInput()

| <chain>.turnOnSwitch(<call_list>)
| <chain>.turnOffSwitch(<call_list>)
| <chain>.getSwitch(<call_list>)
| <chain>.getName(<call_list>)

<call_list> → <expr>, <call_list>
| <expr>

<decleration_list> → <type> IDENTIFIER, <decleration_list>
| <type> IDENTIFIER

Types:

<type> → <prim> | list | obj | sensor

<prim> → bool | int | float | string

<bool_value> → true | false

<constant_value> → INT_VALUE | FLOAT_VALUE

Object Decleration:

<object> → {<properties><methods>}| {<properties>}

<properties> → <type> IDENTIFIER: <expr>
| <type> IDENTIFIER: <expr>, <properties>

<methods> → <object_func> | <object_func> , <methods>

List Declaration:

<array> → [<call_list>]

Loops:

<loop> → loop(<bool_ex>){<stmts>}

Operators:

<comparison> → == | != | < | <= | > | >=

Comments:

<comment> → //TEXT

Descriptions of the Language Constructs

<program> → This non-terminal represents the whole program.

<stmts> → A program is made out of statements. Statements represent one or more statements.

<stmt> → Represents a single statement. A statement can be <matched>, <unmatched> which is for our programs control-statements. While creating this matched-unmatched design, we are inspired by the course book *(Sebesta, 2016, 148-149)*. Control statements are written by using the reserved word **if**, optionally followed by the reserved word **else**. <non-if-stmt> statements are any other statement that is used for assignments, functions, calls, or loops.

<matched> → Represents the control statement if-else. Reserved word **iff** followed by a boolean expression inside parentheses **(** and **)**. Then followed by any kind of statements inside curly brackets **{**, **}**. Which is followed by an **else** reserved keyword. **else** word is also followed by curly brackets, which have any kind of statements inside.

<unmatched>→ Represents an if statement without else. Reserved word **if** followed by boolean expression inside parentheses followed by statements inside parentheses .

<non-if-stmt> → Represents any statement that is not an if statement, which can be a return statement, assignment, function call, or loop.

<assisgn> → Represents assign statement. Declaring a variable with its type and assigning it to the result of an expression with the assign operator "=". It can also be used for assigning a new value to a variable that is already declared by not writing the type of the variable. Variables of objects can also be assigned to an expression for objects by using the "**.**" operator. An element of the given index of an array can also be assigned to an expression.

<return_stmt>→ Represents return statement. Used inside a function to exit from the function, and it returns either a variable, constant value, object, boolean value, text, or array.

<chain> → Represents chain expressions. A chain expression can be resulted as an identifier or an array call. A chain expression can be followed by another chain expression using "." symbol.

<expr> → Represents any expression; expressions are assignable.

<arith_expr> → Represents arithmetic expressions that can be operands of arithmetic operators.

<term> → A derivation of **<arith_expr>**, helps with operator precedence.

<exp> → Expression in parenthesis. It also helps with finalizing the derivation. The expression in parentheses must be done in order to do the rest of the expression.

<bool_ex> → Represents a boolean expression. These expressions can have either of the two values, which is reserved as **true** or **false**.

<func_call> → Represents a function call. A function call is either a static one or an object call. Static functions do not require an object, while object functions are methods that belong to the object. Non-void calls can be assigned to the corresponding type.

<static_call> → Represents a static function call, just like in java and C++. Does not require its object in order to call the function.

<object_call> → Represents an object function call, just like in java and C++. Object functions can be called only by the object that owns the method.

<func> → Represents a function declaration. The reserved word **static** is used to distinguish static functions. Others are all object functions. Object functions are declared in the object declaration. A function call must have a type, or it must be declared with the **void** reserved keyword. A function must also have a name, which is an identifier. It is optional for a function to take arguments.

<static_func> → Represents a static function declaration. Needs the reserved word **static** before type and identifier.

<object_func> → Represents a object function declaration. All non-static functions are object functions. An object function can only be declared inside the object declaration.

<call_list> → Represents arguments for function call. Similar to the argument list; however, this time is used while calling the function.

<decleration_list> → Represents parameters that a function takes. Used while declaring a function (static or object type).

<types> → Represents all variable types in the language. It is either a primitive type, a list type (arrays), or an obj (object). A variable with the list type can hold any type of variables, such as int, string, or even an object or list. Obj type is a special type similar to python's dictionary type. Instead of creating classes for objects, a user can define an object as a variable.

<prim> → Represents primitive types, String, Int, Float, Boolean.

<bool_value> → Represents boolean variables, **true** or **false**.

<constant_value> → Represent numerical values. Basically, any integer value or any float value.

<sensor> → Represetens sensor type. Objects (which are IoT devices) have sensors and some built-in functions such as "getSensors()" returns sensor type.

<object> → Represents the definition of an object. Instead of defining objects in a class, methods and properties of an object are defined in a **{}**.
By default properties of an object and methods of an object is public. **<object>** is assigned to a variable whose type is obj. Methods of an object (object functions) or properties of an object can be accessed by using chain expressions followed by variable or method name. The purpose of the objects are to define and declare IoT devices.

<properties> → Represents the properties of an object, properties of an object are variables and can be any type which is prim, list, or obj. Properties can be accessed by using the variable name.

<methods> → Represents the functions of an object. A method is an object function for that specific object.

<array> → Represents the array definition. The reserved word **list** followed by an identifier is used for declaring an array. An array/list can hold variables of any type (prim, object, list)**.** One list variable can hold different types of variables/constants. Arrays are initialized with a **<call_list>** inside square brackets "**[]"**.

<call_array> → An expression that returns the value/variable from the given array's given index. The index is given inside the square bracket **"[]"** after the identifier as an expression. The returned value/variable itself can also be assigned to an expression to change an element in the array. An element of an array is not defined as a variable. However, **<assign>** non-terminal allows us to modify the contents of the given element of the array as if it is a variable.

<loop> → Represents the only loop in the language. Its function is the same as java while loop. A loop is created by the reserved word **loop** followed by a boolean expression inside parentheses **( )**. Inside the loop (inside curly brackets {} after the loop), there are statements that are executed over and over as long as the boolean expression is **true**. After each execution boolean expression is rechecked.

<comparison> → Comparison operators are operators that return a boolean value. The comparison operators are **<, >, <=, >=, ==**. Which can be applied to the same-typed variables/values with ordinal attributes.

How Nontrivial Tokens are Defined in A++

Identifiers:

-The way identifiers are constructed is: an identifier must start with an alphabetic character, and that alphabetic character can be followed by any number of alphanumeric characters. This convention is used in many imperative programming languages, such as Python, Java, and C++. This convention makes the language **readable** due to the fact that an identifier must start with an alphabetic value; the identifiers are not mistaken for numeric values, operators, or reserved keywords. Also, since the user can use all alphanumeric values, identifiers can be created with names relevant to their uses in the program. Another attribute of the way identifiers are constructed is they are **writable**; since there is no boundary limit, the user can create and use as many identifiers as fits. Reserved keywords are pre-defined in the language. Thus, it is **reliable** since reserved words are not mistaken with identifiers and can perform their specifications.

Comments:

-There are no multiple-lined comments in "A++". This slightly reduces writability since the user will have to repeat the comment symbol "//" for long and multiple-lined comments. However, using only single-lined comments is safer in terms of compilation, and it also decreases the cost of maintenance of the programming language in case of future problems.

Reserved Words and Built-in Functions:

-Reserved words and reserved functions (built-in functions) have special tasks and are set apart from each other by lex. Both reserved words and built-in functions are named related to their tasks in order to have a more **readable** language. Alphanumeric reserved words are defined before identifiers. Thus, a reserved word is not mistaken for an identifier, even if it holds the rules of how identifiers are constructed. Built-in functions are usually functions that might be more frequently needed; hence, they help with **writability**. Some built-in

functions and reserved words are for general usages, such as creating conditional control statements, looping, creating data structures, and manipulating data. There are also more specialized tasks that are for the purpose of the project (IoT Devices), such as connection to a given URL and I/O functions to take inputs.

Literals:

-Literals are either int literals, float literals, or texts. Integer literals are any number of digits followed by any number of digits. They represent the value of the integer. A float literal is any number of digits followed by a dot “.” and followed by one or more digits (45.12 or .5, for example). Text can be literally anything between “ ” (between quotes).

REFERENCES

Sebesta, R. W. (2016). *Concepts of programming languages* (Eleventh edition ; global edition / global edition contributions by Soumen Mukherjee, Arup Kumar Bhattacharjee.). Pearson.