Fall 2022-2023

CS315
Project 1

Team 48
Alperen Koca, Section 2, 21502810
Yüksel Berkay Erdem, Section 2, 21801863
Deniz Tuna Onguner, Section 2, 22001788

PLUTO
The Programming Language

October 14, 2022

BNF

```
<program> ::= <stmnts>
<stmnts> ::= <stmnt> ; | <stmnt> ; <stmnts>
<stmnt> ::= <assign_stmnt> | <if_else_stmnt> |
<loop_stmnt> | <device_stmnt>
<var_type> ::= int | float | str | char | bool
<expr> ::=  <logic_expr> | <math_expr>

<id_list> ::= IDENTIFIER | <id_list> IDENTIFIER

<math_expr> ::= <math_expr> <add_sub_op> <term> |
<term>
<term> ::= <term> <mult_div_op> <factor> | <factor>
<factor> ::= (math_expr) | IDENTIFIER
<add_sub_op> ::= + | -
<mult_div_op> ::= * | /

<assign_stmnt> ::= <var_type> IDENTIFIER = <math_expr>
; | bool IDENTIFIER = <logic_expr> ;

<logic_expr> ::= <logic_expr> <logic_op> <logic_term>
| <logic_term>
<logic_term> ::= (<logic_term>) | <logic_term>
<logic_op> <logic_term> | <math_expr> <comparator_op>
<math_expr>

<logic_op>  ::= && | || | !
<comparator_op> ::= < | > | <= | >= | ==


<if_else_stmnt> ::= if (<logic_expr>) {<stmnts>} |
if(<logic_expr>)  {<stmnts>} else {<stmnts>}

<loop_stmnt> ::= <while_stmnt> | <for_stmnt>
<while_stmnt> ::= while(<logic_expr>){<stmnts>}


<for_stmnt> ::= for(<assign_stmnt>; <logic_expr>;
<iter_expr>) {<stmnts>}
<iter_expr> ::= IDENTIFIER ++ | ++ IDENTIFIER | --
IDENTIFIER | IDENTIFIER –
```

```
<func_def> ::= func IDENTIFIER (<parameters>)
{<stmnts>}
<parameters> ::= <parameter> | <parameter>,
<parameters>
<parameter> ::= IDENTIFIER | <var_type> IDENTIFIER
<func_call> ::= IDENTIFIER (<parameters>);
<return_stmnt> ::= return (<id_list>); | return
IDENTIFIER;

<comment> ::= // <str> | /* <str> */

<sign> ::= + | -
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<numeric> ::= <digit> <numeric> | <digit>
<int>    ::= <sign> <numeric> | <numeric>
<float> ::= <int> . <numeric>

<hour_time> ::= PM <hour> : <minutes> | AM <hour> :
<minutes>
<hour> ::= | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
11 | 12

<minutes> ::= <min_sec_digit>
<min_time> ::= <minutes> : <seconds>
<seconds> ::= <min_sec_digit>
<min_sec_digit> ::= <zero_to_five><digit>
<zero_to_five> ::= 0|1|2|3|4|5

<device_stmnt> ::= <get_device_data>; |
<set_device_data>; | <connect_device>; |
<switchOn_device>; | <switchOff_device>;

<device_list> ::= <device> | <device> <device_list>
<device> ::= <device_type> IDENTIFIER
<device_type> ::= Temperature | Light | Humidity |
AirPressure | AirQuality | SoundLevel
<get_device_data> ::= getDataOf (<device_list>)
<set_device_data> ::= setDataOf (<device_list>,
<numeric>) | setDataOf(<device_list>, <hour_time>) |
setDataOf(<device_list>, <min_time>)
```

```
<connect_device> ::= connect(<device_list>,
<device_list>)

<switchOn_device> ::= open(<device_list>) |
on(<device_list>)
<switchOff_device> ::= close(<device_list>) |
off(<device_list>)

<bool> ::= <True> | <False> | 1 | 0
<True> ::= 1
<False> ::= 0

<char_upper_case> ::= 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

<char_lower_case> ::= 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

<char> ::= <char_upper_case> | <char_lower_case>
<str> ::= “ <char> <str> “ | “ <char> “ | “”
```

RESERVED WORDS

if: if semantic structure for case utilization within the language.
else: the structure corresponding to the reverse case of else.
int: The keyword denoting the integer variable type.
string: The keyword referring to the string variable type.
char: The keyword for single character variable type.
float: The keyword for pointed numeric valued variable type.
bool: The keyword for boolean variables.
setDataOf: The keyword for the function that sets data into an IoT device.
getDataOf: The keyword for the function that gets data into an IoT device.
for: The keyword that refers to the traditional "for loop" in programming languages.
func: The keyword that is used while defining a function.
temperature: The keyword that refers to temperature sensor devices in IoT systems.
airQuality: The keyword that refers to air quality sensor devices in IoT systems.
airPressure: The keyword that refers to ait pressure sensor devices in IoT systems.
light: The keyword that refers to light sensor devices in IoT systems.
humidity: The keyword that refers to humidity sensor devices in IoT systems.
soundLevel: The keyword that refers to sound level sensor devices in IoT systems.

Non-Terminal Definitions

<program> : the starting statement of the language BNF
<stmnts> : BNF structure referring to both plural and singular form of statements
<stmnt> : Singular form of statements that can be extended into assignment, if-else, loop and device statements.
<var_type> : BNF structure denoting the specific variable type. The variable types in the language are integer (int), float (float), string (str), character (char) and boolean (bool).
<expr> : Abstract expression entity that can be extended into logical and mathematical expressions.
<id_list> : Identifier list that can consist of a singular identifier or an identifier list.
<math_expr> : Mathematical expression entity that utilizes operator precedence of addition and subtraction.
<term> : Mathematical expression segment defining the precedence of multiplication and division.
<factor> : Mathematical expression segment defining the precedence of parentheses in expressions.

<add_sub_op> : BNF structure consists of terminals “+” and “-”.
<mult_div_op> : BNF structure consisting of terminals “*” and “/”.
<assign_stmnt> : Statement type defining the grammar of assignments. In the language, an assignment can be done by “variable type-variable name = variable value” structure.
<logic_expr> : Logical expression structure which also defines the operator precedence “and”, “or”, “not” logical operators.
<logic_term> : Logical expression structure which also defines the comparator usage within logical expressions.
<logic_op> : The BNF non-terminal referring into terminals of “&&”, “ ||”,“!”, which imply the logical operators of “and”, “or”, “not” in the given order.
<comparator_op> : Non-terminal denoting to comparator operators of equality, less than, more than and others.
<if_else_stmnt> : Statement type for if-else structure which also emphasizes if else matching.
<loop_stmnt> : Statement type for traditional while and for loops in programming languages.
<while_stmnt> : Statement type for traditional while
<for_stmnt> : Statement type for traditional for
<iter_expr> : BNF structure for defining the iterative parts within for loops.
<func_def> : Abstract BNF entity referring to function declaration statements using func keyword.
<parameters> : Plural form of parameter that is being used in function declarations.
<parameter> : Singular form of parameter which requires variable type specifications and names in advance.
<func_call> : Non-terminal regulating the callings of defined functions.
<return_stmnt> : S
<comment> : An expression structure for the format of comments.
<sign> : BNF structure consisting of terminals “+” and “-” to understand the sign of the integer (int) or float (float) values.
<digit> : BNF structure that distinguishes the digits from char and string inputs so that these digits can be used to assign the identifiers’ values.
<numeric> : BNF structure enables the language to aware of numbers that have one or more digits.
<int> : Type-decider structure for both integer (int) values that have a sign or not.
<float> : Type-decider structure for float (float) values. Also, this structure makes the language have ability to distinguish float (float) values from integer (int) values.
<hour_time> : This structure defines the format of received hour time which can be “AM” or “PM” and the order of “hour” and “minutes” variables.

<hour> : BNF structure consisting of numbers from “1” to “12” is used for understanding that the input value is an hour value.
<minutes> : BNF structure that uses another BNF structure type “<min_sec_digit>”. This structure can be used for defining the minute value by not being confused about other data types.
<min_time> : This structure is used for defining the format and the order of minutes and seconds such as “37 : 52” which can be written as “minutes : seconds”.
<seconds> : Similar to the “<minutes>” structure, this structure uses another BNF structure type “<min_sec_digit>”. This structure can be used for defining the second value by not being confused about other data types.
<min_sec_digit> : BNF structure which controls that the first digit of the input is in a range from “0” to “5” and the second digit of the input is in a usual digit.
<zero_to_five> : BNF structure to check the input is one of the elements of the list consisting “0”, “1”, “2”, “3”, “4”, and “5”. This structure also helps the other structure called “<min_sec_digit>”.
<device_stmnt> : The structure to denote the device related statements which are briefly manipulating datas of devices.
<device_list> : The BNF entity referring to both singular sensor device and plural sensor devices.
<device> : Structure corresponding to a single sensor device.
<device_type> : Device variable type which can be terminals of Temperature, Light, Humidity, AirPressure, AirQuality and SoundLevel.
<get_device_data> : Integrated get function for getting device datas by “getDataOf (<device_list>)”
<set_device_data> : Integrated set functions which has the ability of manipulating numeric data values and time information of devices by “setDataOf (<device_list>, <numeric>)”, “setDataOf(<device_list>, <hour_time>)” and “setDataOf(<device_list>, <min_time>)”.
<connect_device> : The BNF structure specializing in connection of devices by grammatically using the keyword as “connect(<device_list>, <device_list>)”.
<switchOn_device>:Switch off structure using “open(<device_list>)” and “on(<device_list>)” keyword grammars.
<switchOff_device>: Switch off structure using “close(<device_list>)” and “off(<device_list>)” keyword grammars.
<bool> : Type-decider structure for boolean (bool) values.
<True> : A structure that is used for changing the input to a true boolean structure for the structure called “<bool>”.
<False> : A structure that is used for changing the input to a false boolean structure for the structure called “<bool>”.
<char_upper_case> : An uppercase char list that consists of all capital letters from “A” to “Z”.

<char_lower_case> : An lowercase char list that consists of all lower letters from “a” to “z”.
<char> : BNF structure that defines what could be a char.
<str> : String type expression structure for char lists following each other.

Language Evaluation Criteria

Readability

Pluto programming language has been designed for IoT purposes and therefore should be readable by IoT designers. The language is implemented to be simple, orthogonal and traditional in syntax. Pluto uses conceptual loop statements and if-else structure so that anyone who knows a programming language can also read Pluto codes. For sake of simplicity, the iteration of integers are limited with "integer++" and "++integers" and similar subtraction versions. Operator and function overloading is not allowed so that an operator or function definition would refer only to one functionality. There is no pointer structure in Pluto so that readability is not hindered, although it hinders the writability of Pluto. The classical data types of integers, strings also exist in Pluto, while Pluto also offers many IoT sensor device variables that are integrated into base Pluto so that any IoT designer can easily grasp the semantics. The time value denotations are also designed explicitly so that IoT designers can easily manipulate and read time variables of devices. These features make Pluto more readable than other programming languages for IoT programmers.

Writability

The design motivations are not limited to but heavily emphasizing writability with Pluto language. Pluto allows IoT designers and engineers to define their own abstract devices. It also offers classical primitive data types and function declarations that are easy to use for anyone with a basic programming background. Additionally, it is easy to access sensors and switches by simply typing their identifier, which makes Pluto very practical in the sense of connecting with the hardwares of IoTs. Pluto also allows comments so that developers provide necessary information to their colleagues and their future selves, which may help them to type faster and more accurately. Moreover, Pluto has advanced operators like the power operator (**), which reduces the necessity of additional libraries for basic instructions. Pluto only supports one type of loop, for loop, in order to prevent developers confuse between different loop options.