CS 315

Programming Languages

Project

AMIoT

İsmail Şahal

Muzaffer Yasin Köktürk

Şeyma Aybüke Ertekin

Introduction

AMIoT is a programming language that is designed for people using IoT devices. While we were developing our programming language, we tried to make it writable by eliminating difficulties we had while programming in other languages. We also tried to make it readable and reliable. We used similar keywords with other languages so that people who learnt programming languages can easily code in our language.

BNF Description of AMIoT

<program> := start <block> finish

<block> ≔{ <stmts> } | {} | {\n}

<stmts> ≔<stmt> | <stmt> <stmts> | \n <stmts>

<stmt> ≔<matched> | <unmatched>

<matched> ≔if(<logic_expr>) \n<matched> else \n<matched>| <non_if_stmt>

<unmatched> ≔if (<logic_expr>) \n<stmt> | if (<logic_expr>) \n<matched> else \n<unmatched>

<non_if_stmt> ≔<while_stmt> | <loop_stmt> | <declaration_stmt> | <assignment_stmt>

| <function_definition> | <function_stmt> | <primitive_funct_stmt> | <output_stmt> |

<input_stmt> | <return_stmt> | <comment>

<while_stmt> ≔while (<logic_expr>) <block> \n

<loop_stmt> ≔loop(<assignment>; <logic_expr>; <assignment>) <block> \n

<declaration> ≔<variable_declaration> | <constant_declaration>

<constant_declaration> ≔const <var_type> <const_list>

<const_list> ≔<const_name> | <const_name>, <const_list>

<variable_declaration> ≔<var_type> <ident_list>

<ident_list> ≔<identifier> | <identifier>, <ident_list>

<const_name> ≔$$<name>

<identifier> ::=$<name>

<name> ::=<letter> | <letter> { <alphanumeric>}

<var_type> ::=connection | int

<logic_expr> ::=<logic_expr> or <comp_expr> | <logic_expr> xor <comp_expr> | <comp_expr>

<comp_expr> ::=<comp_expr> and <comp_stmt> | <comp_stmt>

<comp_stmt> ::=<bool_cons> | <not><comp_stmt> | (<logic_expr >) | <art_expr> <relation_op> <art_expr>

<bool_cons> ::=true | false

<not> ::=~

<relation_op> ::=<= | >= | < | > | = | /=

<assignment> ::=<id_assignment> | <switch_assignment> | <const_assignment> | <declaration_assignment>

<switch_assignment> ::=<primitive_switches> <- <binary_digit>

<declaration_assignment> ::=<declaration> <- <art_expr>

<id_assignment> ::=<identifier> <- <art_expr>

<const_assignment> ::=<const_name> <- <art_expr>

<primitive_switches> ::=switch<digit>

<art_expr> ::=<art_expr> + <term> | <art_expr> - <term> | <term>

<term> ::=<term> * <factor> | <term> / <factor> | <factor>

<factor> ::=(<art_expr>) | <numeric>

<numeric> ::=<identifier> | <integer> | <binary_digit> | <function_call> | <primitive_funct_call> | <const_name> | <primitive_switches>

primitive_funct_call: time() | createConnection(<URL>) | <identifier> . sendTo (<numeric>

) | <identifier> . receiveFrom() | readTemperature() | readHumidity() | readAirPressure ()

| readAirQuality() | readLight() | readSoundLevel40() | readSoundLevel200()

|readSoundLevel400() | readSoundLevel1000() | readSoundLevel4000()

<function_definition> ::=<function_header> <block> \n

<function_header> ::='<'<return_type>'>' <function_name> (<parameter_type_list>)

<parameter_type_list> ::=empty | <var_type> <identifier> | <var_type> <identifier>;

<parameter_type_list>

<return_type> ::= empty | <var_type>

<function_call> ::=<function_name> (<argument_list>) |

<identifier>.<function_name> (<argument_list>)

<function_name> ::=<upper_case_letter> | <upper_case_letter> {<alphanumeric>}

<argument_list> ::= empty | <argument> | <argument>; <argument_list>

<argument> ::=<art_expr>

comment: <line_comment> \n |

<block_comment> \n ; comment:

<line_comment> \n | <block_comment> \n ;

<input_stmt> ::= take(<identifier>)

<output_stmt> ::= show (<output>)

<output> ::= empty | "<text_output>" | <art_expr> | <logic_expr>

<function_call_stmt> ::=<function_call> \n

<primitive_funct_stmt>: <primitive_funct_call> \n ;

<declaration_stmt> ::=<declaration> \n

<assignment_stmt> ::=<assignment> \n

<alphanumeric> ::=<letter> | <integer>

<integer> ::=<digit> | <digit> <integer>

<letter> ::= <upper_case_letter> | <lower_case_letter>

<upper_case_letter> ::= A | B | C … | Z

<lower_case_letter> ::= a | b | c … | z

<digit> ::= <binary_digit> | 2 | 3 | 4 | 5 | 6 | 7 |8 | 9

<binary_digit> ::= 0 | 1

<URL> ::= http://<text_output> | https://<text _output>

<text_output> ::= <char><text_output> | <char>

<char> ::= .

AMIoT Definition

<program> := start <block> finish

‘start’ and ‘finish’ are used to specify the beginning and end of each program. We thought that using these expressions will increase readability of our programming language.

<block> ::= { <stmts> } | {} | {\n}

<stmts> ::= <stmt> | <stmt> <stmts> | \n <stmts>

Most of the programming languages use ‘;’ between statements. However, a new line between statements would make program more writable. Programmer can start to write a new statement by just pressing enter. Blocks are usually used for loops and functions. Even if curly brackets decrease writability, they are useful to differentiate statements inside a function or loop statement. Therefore, they are important in terms of readability.

<matched> ::= if(<logic_expr>) \n<matched> else \n<matched>| <non_if_stmt>

<unmatched> ::= if (<logic_expr>) \n<stmt> | if (<logic_expr>) \n<matched> else \n<unmatched>

Statements can be matched or unmatched. These expressions are used to identify nested if else blocks. The construction of if statement is same with Java or C++. Because people usually get used to these programming languages, it is beneficial to leave them same.

<non_if_stmt> ≔<while_stmt> | <loop_stmt> | <declaration_stmt> | <assignment_stmt>

| <function_definition> | <function_stmt> | <primitive_funct_stmt> | <output_stmt> |

<input_stmt> | <return_stmt> | <comment>

<while_stmt> ≔while (<logic_expr>) <block> \n

While statement stays same with Java or C++. Because programmers get used to it, we did not change its construction.

<loop_stmt> ≔loop(<assignment>; <logic_expr>; <assignment>) <block> \n

We used ‘loop’ rather than ‘for’ in other languages. Again we did not change the construction. We just changed the name. It may be easier to remember.

<declaration> ≔<variable_declaration> | <constant_declaration>

<constant_declaration> ≔const <var_type> <const_list>

<const_list> ≔<const_name> | <const_name>, <const_list>

<variable_declaration> ≔<var_type> <ident_list>

<ident_list> ≔<identifier> | <identifier>, <ident_list>

<const_name> ≔$$<name>

<identifier> ≔$<name>

<name> ≔<letter> | <letter> { <alphanumeric>}

<var_type> ≔connection | int

Variable declaration consist of a type and one or more identifiers. Identifiers are restricted to start with a ‘$’ and a letter, then continue with letters or numbers. Any word that starts with a dollar sign and continues with a letter MUST be an identifier name. Constant
declaration consist of ‘const’, a type and one or more constant names. Constant identifiers are restricted to start with ‘$$’ and continue with letters or numbers. This increases readability and makes it easier to realize what an identifier is. Any word that starts with two dollar sign is recognized as a constant name. In our language there are two types of primitive data types. It could be a connection type which we define for connection to a device or an integer type.

<logic_expr> :=<logic_expr> or <comp_expr> | <logic_expr> xor <comp_expr> | <comp_expr>

<comp_expr> :=<comp_expr> and <comp_stmt> | <comp_stmt>

<comp_stmt> :=<bool_cons> | <not><comp_stmt> | (<logic_expr >) | <art_expr> <relation_op> <art_expr>

<bool_cons> :=true | false

<not> :=~

<relation_op> :=<= | >= | < | > | = | /=

Logical expressions may contain more than one comparison. Their relationship can be ‘and’, ‘or’, ‘xor’. We thought that using their exact name for logical operations would make our language more writable because it could be difficult to get used to the signs used instead of them. Also, we followed precedence rule. Precedence decreases in the following sequence:

1- parenthesizes, not

2- and

3- or, xor

‘<’, ‘>’, ‘<=’, ‘>=’, ‘=’, ‘/=’ are used for comparisons. Because they are so similar to the signs in math used for comparison, they are easy to get used to, readable and writable. We used one equality sign for equality comparison. Since, we recognized that putting one equality sign rather than two for comparison is one of common mistakes new programmers do.

<assignment> :=<id_assignment> | <switch_assignment> | <const_assignment> |
<declaration_assignment>

<id_assignment> :=<identifier> <- <art_expr>

<const_assignment> :=<const_name> <- <art_expr>

We used ‘<-‘ for assignment expressions. Equality sign is reserved for comparison. We thought that this sign is compatible with real world expressions. It is also more readable.

<art_expr> contains variety of operations as well as arithmetic operations. It can be a function call, an integer, an identifier, a constant or a switch.

<switch_assignment> ::=<primitive_switches> <- <binary_digit>

<primitive_switches> ::=switch<digit>

Our language contains primitive variables. They are switch0, switch1, …, switch9. They are used to set 10 switches, which IoT devices contain, on/off. Binary digits are used for this
expression. If one of the switches assigned to be '0', that means it is on and off for otherwise.

<declaration_assignment> ::=<declaration> <- <art_expr>

To increase writability, we let programmer to declare and initialize a variable in the same statement.

<art_expr> ::=<art_expr> + <term> | <art_expr> - <term> | <term>

<term> ::=<term> * <factor> | <term> / <factor> | <factor>

<factor> ::=(<art_expr>) | <numeric>

<numeric> ::=<identifier> | <integer> | <binary_digit> | <function_call> | <primitive_funct_call> | <const_name> | <primitive_switches>

We used '+', '-', '/' and '*' for arithmetic operations. Same signs are used with math except for "*". However, writing '*' for multiplication rather than '.' or 'x' makes our program more readable. Also, we considered precedence rule in math. Because people are trained to do operations according to this rule, it is beneficial to make our program compatible with this rule. A programmer can do operations with identifiers, integers, function calls, constants and switches.

primitive_funct_call: time() | createConnection("<URL>") | <identifier> . sendTo (<art_expr>) | <identifier> . receiveFrom() | readTemperature() | readHumidity() | readAirPressure () | readAirQuality() | readLight() | readSoundLevel<40>() | readSoundLevel<200>() |readSoundLevel<400>() | readSoundLevel<1000>() | readSoundLevel<4000>()

AMIoT has 14 primitive functions. Programmer can read data from sensors by using read functions. This function returns integer value of data. There are 6 sensors which data can be taken from. If sensor is sound level, there is a different situation. time() returns current timestamp. createConnection(<URL>) takes an URL and creates a connection to this URL. A URL starts with ‘https://’ or ‘http://’ and continues with any other character. This function returns connection created. sendTo(<art_expr>) can take an arithmetic operation, an integer, a function call, an identifier, a constant or a switch as an argument, sends it to a connection, and does not return anything. Using this function requires a connection type identifier. receiveFrom() receives an integer from a connection and returns integer taken.
Using this function requires a connection type identifier, as well. Primitive type functions start with lower case letters to separate them from user defined functions.

<function_definition> ::=<function_header> <block> \n

<function_header> ::=‘<’<return_type>’>’ <function_name> (<parameter_type_list>)

<parameter_type_list> ::=empty | <var_type> <identifier> | <var_type> <identifier>;
<parameter_type_list>

<return_type> ::= empty |

<var_type> return_stmt ::=return

<art_expr> \n ;

Function definition contains function header and block which contains statements. Function header has a return type inside angular brackets and then function name. We placed angular brackets to separate return type. Function name starts with an upper case letter and continues with letters or integers. Function return type and parameter type can be variable types which are int and connection. A function can return using return statement.

<function_call> ::=<function_name> (<argument_list>) |
<identifier>.<function_name> (<argument_list>)

<function_name> ::=<upper_case_letter> | <upper_case_letter> {<alphanumeric>}

<alphanumeric> ::=<letter> | <integer>

<integer> ≔<digit> | <digit> <integer>

<argument_list> ≔ empty | <argument> | <argument>; <argument_list>

<argument> ≔<art_expr>

Programmer can call functions by typing function name and their arguments inside parenthesis. If there are more than one argument, arguments are separated by ‘;’. Our aim was to make it easy to notify different arguments. It would be easier to read. Arguments would be an arithmetic operation, an integer, a function call, an identifier, a constant or a switch.

comment: <line_comment> \n | <block_comment> \n ;

There is also comments which can be a block comment which starts after ‘/~’ and ends with ‘~/’ or a line comment which starts with ~~ and ends with new line. Tilda sign is easy to see. Therefore, using this sign before and after comment makes it easier to notify comment, i.e. increases readability.

<input_stmt> ≔take(<identifier>)

<output_stmt> ≔show (<output>)

<output> ≔empty | “<text_output>” | <art_expr> | <logic_expr>

Our language contains input and output statements which enable programmer to print something or take something from the user and assign it to an identifier.

<function_call_stmt> ≔<function_call> \n

<primitive_funct_stmt>: <primitive_funct_call> \n ;

<declaration_stmt> ≔<declaration> \n

<assignment_stmt> ≔<assignment> \n

A function can be called for an arithmetic operation or in another function call. Therefore, we have written these statements to differentiate a line statement of a function call and its use for other purposes.

int, connection, and, or, xor, if, loop, while, else, start, finish, return, const, switch0, switch1, …, switch9, take, show are reserved words for our language.

Notes:

The programs gives error for the following cases:

1- New lines when the program is

empty. start {

}finish

2- New lines before left curly bracket of block statements or after right curly bracket before finish.

start
{
~~Programs
}
finish

The solutions that we come up with for these problems resulted in conflicts. Therefore, we decided to leave them since they are not critical for the usage of the language.