CS 315 - Programming Languages

Project 1

Language Name: Garry++

Ata COŞKUN
21503061
Section 3

Elif Beril ŞAYLİ
21502795
Section 3

Kaan ÜNLÜ
21602662
Section 1

Table of Contents

Table of Contents 1

Requirements 3

Review of Remaining Conflicts 3

Explanation of Garry++ 4

BNF of the Language 4

Explanation of the Language's BNF 7

<start>: Start 7

<program>: Program 7

<stmts>: Statements 7

<stmt>: Statement 7

<if>: If Statement 7

<nonif>:Non - If Statement 7

<matched>: Matched Case 7

<unmatched>: Unmatched Case 7

<loopstmt>: Loop Statement 8

<for>: For Statement 8

<while>: While Statement 8

<asgnmnt>: Assignment Statement 8

<returnstmt>: Return Statement 8

<blockstmt>: Block Statement 8

<funcstmt>: Function Statement 8

<expression>: Expression 8

<logicexp>: Logic Expression 8

<nonlogicexp>: Non-logic Expression 8

<mathexp>: Mathematical Expression 9

<term>: Term 9

<havepar>: Parentheses Check 9

<factor>: Factor 9

<constant>: Constant 9

<funccal>: Function Call 9

<declarations>: Declarations 9

<vardec>: Variable Declaration 9

<constdec>: Constant Declaration 9

<funcdec>: Function Declaration 9

<funcargs>: Function Arguments 10

<logicop>: Logic Operator 10

<readDataFromSensor>: Read Data from Sensor 10

<timestamp>: Time Stamp 10
<connectionOfInternet>: Internet Connection 10
<connect>: Connect 10
<disconnect>: Disconnect 10
<openSwitch> 10
<closeSwitch> 10
<sw>: Switch 11
<getURL>: Get URL 11
<sendURL>: Send URL 11
<getTemperature>: Get Temperature 11
<getHumidity>: Get Humidity 11
<getAirPressure>: Get Air Pressure 11
<getAirQuality>: Get Air Equality 11
<getLight>: Get Light 11
<getSoundLevel>: Get Sound Level 11
<otherFunctionCalls>: Other Function Calls 11
<param>: Parameter 11
<helper>: Helper 11
<empty>: Empty 12

Descriptions of Nontrivial Tokens **12**
Comments 12
İdentifiers 12
Equality Operator 12
Predefined Symbols 12

Example Program **14**

Requirements

To provide support for IOT edge devices the following features should be supported in the language:

1. Sensors; eg., temperature, humidity, air pressure, air quality, light, sound level (for a given frequency),
2. 10 switches that can be set as on/off to control some actuators,
3. A timer that returns the current timestamp, the number of seconds elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds.
4. Connection to the internet
5. Primitive functions: read data from each sensor, timestamp from timer.
6. Mechanism to define a connection to a given URL
7. Mechanisms to send/receive an integer at a time, from/to a connection .
8. Variable names,
9. Assignment operator, arithmetic operators,
10. Arithmetic expressions,
11. Conditional statements; e.g., if-then, if-then-else,
12. Loops; e.g., for, while,
13. Mechanisms for defining and calling functions,
14. Comments.

Review of Remaining Conflicts

The final state of the syntax results in no conflicts, and the test program runs without problems.

Explanation of Garry++

Garry++ is created for everyone. People who purchase these devices may not be computer engineers so our language is very close to daily language for them. On the other hand, we don't forget computer engineers. Its syntax is similar to other popular programming languages such as C and Java and easy to understand. During creating language stage, we benefit from other languages basic rules.

To prevent ambiguities stemming from the matched/unmatched conditions of if-else cases, we separated all if statements as matched/unmatched statements which using an intuitive recursive flow increase readability and reliability. Also, we used "is" as an assignment operator to make connections with daily language. Moreover, we used reserved words for block statements , if statements and for loop statements such as start, end, by, then, to and from. These words are chosen with a purpose because they are understandable, short and prevent confusions within language. It provides easy tracking when reading a code thus increasing readability. Naming of whole language including predefined functions is based on natural language and easy to understand. Being able to use to use spaces and tabs at will with no effect also helps with the readability. This intuitiveness of the language that makes it easy to read hence makes it easier to write too: As reading is unhindered by small technical terms, writability of the language is also high, making it easy to pick up without much training.

We have precedence rules for mathematical expressions to prevent ambiguity and increase readability. For example, multiplication, division and mod have a priority when we compared to addition and subtraction. We have used left recursion thus our mathematical operations are left associative.

BNF of the Language

```
<start>                 ::=     <program>

<program>               ::=     <stmts> | <empty>

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

<stmt>                  ::=     <if> | <nonif>

<nonif>                 ::=     COMMENT | <loopstmt> | <returnstmt>
                        | <funccal> | <asgnmnt> | <declarations>

<declarations>          ::=     <constdec> | <vardec> | <funcdec>

<constdec>              ::=     CONST TYPE <asgnmnt>

<vardec>                ::=     TYPE VARIABLE EOL| TYPE <asgnmnt>

<funcdec>               ::=     FUNC VARIABLE LP <funcargs> RP THEN
                        <funcstmt> END
```

```
<funcargs>          ::=     TYPE VARIABLE | <funcargs> COMMA
                    TYPE VARIABLE | <empty>

<funcstmt>          ::=     <stmts> | <returnstmt>  EOL

<asgnmnt>           ::=     VARIABLE ASSIGN <expression> EOL

<if>                ::=     <matched> | <unmatched>

<unmatched>         ::=     IF LP <logicexp> RP THEN
                    <blockstmt> | IF LP <logicexp> RP THEN
                    <matched> ELSE <unmatched> END

<matched>           ::=     IF LP <logicexp> RP THEN <matched>
                    ELSE <matched>| <blockstmt>

<loopstmt>          ::=     <for> | <while>

<while>             ::=     WHILE LP <logicexp> RP <blockstmt>

<for>               ::=     FOR LP VARIABLE FROM <term> TO
                    <term> BY <term> RP <blockstmt>

<blockstmt>         ::=     START <stmts> END

<returnstmt>        ::=     RETURN <expression> EOL

<expression>        ::=     <nonlogicexp>  | <logicexp>

<nonlogicexp>       ::=     <mathexp> |<funccal>

<logicexp>          ::=     <logicexp> <logicop> <nonlogicexp>
|
                    <nonlogicexp> <logicop> <nonlogicexp>

<logicop>           ::=     EQUAL | NOTEQUAL | BIGGEREQUAL |
                    SMALLEREQUAL | SMALLERTHAN | BIGGERTHAN |
                    AND | OR

<funccal>           ::=     <readDataFromSensor> | <timestamp>
|
                    <connectionOfInternet> | <getURL> |
                    <sendURL> | <getTemperature> |
<getHumidity>
                    | <getAirPressure> | <getLight> |
                    <getSoundLevel> | <getAirQuality> |
                    <openSwitch> | <closeSwitch> |
                    <otherFunctionCalls>
```

```
<readDataFromSensor>    ::=     READDATAFROMSENSOR LP VARIABLE RP

<timestamp>             ::=     TIMESTAMP LP RP

<connectionOfInternet>  ::=     CONNECT LP STRING RP | DISCONNECT
LP
                        RP

<getURL>                ::=     GETURL LP VARIABLE RP

<sendURL>               ::=     SENDURL LP VARIABLE COMMA <mathexp>
                        RP

<getTemperature>        ::=     GETTEMPERATURE LP RP

<getHumidity>           ::=     GETHUMIDITY LP RP

<getAirPressure>        ::=     GETAIRPRESSURE LP RP

<getLight>              ::=     GETLIGHT LP RP

<getSoundLevel>         ::=     GETSOUNDLEVEL LP RP

<getAirQuality>         ::=     GETAIRQUALITY LP RP

<openSwitch>            ::=     OPEN LP <sw> RP

<closeSwitch>           ::=     CLOSE LP <sw> RP

<otherFunctionCalls>    ::=     VARIABLE LP <param> RP

<param>                 ::=     <expression> | <expression> COMMA
                        <helper> | <empty>

<helper>                ::=     <expression> COMMA <helper> |
<expression>

<sw>                    ::=     CHOSENSWITCH

<mathexp>               ::=     <term> PLUS  <mathexp> | <term>
                        MINUS  <mathexp> | <term>

<term>                  ::=     <havepar> DIVIDE <term> | <havepar>
                        MULT  <term> | <havepar> REMAINDER <term> |
                        <havepar>

<havepar>               ::=     LP <mathexp> RP |factor

<factor>                ::=     <constant> | VARIABLE
```

```
<constant>              ::=     STRING | INTEGER|FLOAT|LOGIC

<empty>                 ::=
```

Explanation of the Language's BNF

<start>: Start

"Start" is the initializing rule of any program in Garry++. It is formed only of a program <program>.

<program>: Program

A non-terminal that consists of statements or is empty. It can be formed of <stmts> or <empty> that contains filled statements or an empty statement . This allows the user to have an empty program. Also, a program can start without the need for any indicated reserved word.

<stmts>: Statements

A "statements" is a set of one or more singular statements <stmt>.

<stmt>: Statement

A statement has two types. These are <if> and <nonif> statements.

<if>: If Statement

As every if statement has the potential to have an else linked to it, to identify what else belongs to which if, every if statement consists of either a matched statement, or an unmatched statement.

<nonif>:Non - If Statement

A non-if statement is either a comment COMMENT, a loop statement <loopstmt>, a return statement <returnstmt>, a function call <funccal>, an assignment statement <asgnmnt>, or a set of declarations <declarations>. The wide variety of different things a statement can be mean these statements make up a large bulk of any written code. As all function calls and assignments are single line statements by nature, we use the EOL (end of line) token to separate between them in their particular rules.

<matched>: Matched Case

A matched case always has an even number of if and elses that recursively match with each other. The condition of an if statement can be satisfied based on the truth value of the logic expression <logicexp> supplied in it. If the said logic expression is true, the statement <stmt> after the if matched case between the "then" and "else" tokens is executed, else, the matched case after the else is executed. This matched case can also be a block statement <blockstmt> to serve statements and eventually terminate the recursion. For a case to be matched, it can only consist of other matched <matched> cases and statements. If any statement inside a matched case is unmatched, the total of the recursive cases practically becomes unmatched, but the opposite is impossible as a case is always practically unmatched after an unmatched case is nested under it.

<unmatched>: Unmatched Case

Unlike a matched case, in an unmatched case the number of ifs and elses do not match, which means that some ifs do not have elses. Also unlike a matched <matched> case,

an unmatched <unmatched> case can have matched cases in it as well as st, as any case with at least one unmatched case nested in it is ultimately unmatched.

<loopstmt>: Loop Statement

Every loop statement is either a while <while> or a for <for> statement.

<for>: For Statement

Every for statement is a loop statement. This loop statement is constructed with the token FOR followed by a variable VARIABLE, followed by the token FROM, followed by a term <term>, followed by another token TO, which is followed by another term <term>, which then is followed the token BY and then by yet another term <term> where everything after the FOR token to here is between two tokens LP and RP for left and right parentheses, which then is followed by a block statement <blockstmt>. The statements inside the block statement is executed as long as the conditions inside the parentheses are fulfilled.

<while>: While Statement

Every while statement is a loop statement. This loop statement is constructed with the token WHILE followed by a logic expression <logicexp> between two tokens LP and RP for left and right parentheses, which then is followed by a block statement <blockstmt>. The statements inside the block statement is executed as long as the conditions inside the parentheses are fulfilled.

<asgnmnt>: Assignment Statement

The assignment statement consists of a variable VARIABLE, followed by the ASSIGN token “is”, followed by an expression <expression> and the end of line token EOL. It assigns the result of the expression to the variable.

<returnstmt>: Return Statement

This statement is written with the RETURN terminal followed by an expression <expression> followed by the end of line token EOL. It returns the value of the expression.

<blockstmt>: Block Statement

A block statement is a non-terminal set of statements <stmts> between two terminals START and END.

<funcstmt>: Function Statement

A function statement is either a set of statements <stmts> or a return statement <returnstmt> followed by the end of line token EOL. The reason for the presence of this rule is to have the return statement as a usable statement in functions, as the return statement is not a part of a normal statement.

<expression>: Expression

An expression is either a non-logic expression <nonlogicexp> or a logic expression <logicexp>.

<logicexp>: Logic Expression

A logic expression is either a logic expression <logicexp>, a logic operator <logicop> and a non-logic expression <nonlogicexp>, or a non-logic expression <nonlogicexp>, a logic operator <logicop> and another non-logic expression <nonlogicexp>. A truth value is returned based on the nature of the logic operator between the two expressions.

<nonlogicexp>: Non-logic Expression

A non-logical expression is either a mathematical expression <mathexp> or a function call <funccal>.

<mathexp>: Mathematical Expression

A mathematical expression is a term <term>, a terminal token + or - and a mathematical expression <mathexp>. Alternatively, it can be just a term <term>. It is used just to calculate addition or subtraction, and is hierarchically one step lower than <term> in the recursion order (as it is made out of terms, which have to be read before the mathematical expression for the mathematical expression to be read correctly) as multiplication and division are higher priority in algebra compared to addition and subtraction.

<term>: Term

A term's construction begins with a check for the presence of a parenthesis with a <havepar>, then is followed by a terminal *, / or %, and a term <term>. Alternatively, it can be just a parentheses check <havepar>.

<havepar>: Parentheses Check

A parentheses check is either just a factor <factor> or a mathematical expression <mathexp> between a left parenthesis LP and right parenthesis RP. It is used for cases where parentheses are used to group mathematical expressions.

<factor>: Factor

A factor is a constant <constant> or a variable VARIABLE.

<constant>: Constant

A constant is a non-terminal STRING, INTEGER, FLOAT or LOGIC. The content of these terminals are determined at compile time.

<funccal>: Function Call

A function call can be a call to read data from a sensor's ID <readDataFromSensorID>, get the timestamp from January 1st 1970 <timestamp>, get the connection state of the Internet <connectionOfInternet>, get a URL <getURL>, send a URL <sendURL>, get the temperature of the environment <getTemperature>, get the humidity of the environment <getHumidity>, get the local air pressure <getAirPressure>, get the local light level <getLight>, get the local sound level <getSoundLevel>, open a switch <openSwitch>, close a switch <closeSwitch> or it can be another function call <otherFunctionCalls>, if it isn't any one of the remaining specific functions calls.

<declarations>: Declarations

Declarations can be the declaration of a constant <constdec>, declaration of a variable <vardec>, or the declaration of a function <funcdec>.

<vardec>: Variable Declaration

A variable declaration can either be a type TYPE followed by a variable VARIABLE followed by the end of line token EOL, or it can be a type TYPE followed by and assignment <asgnmnt>. This allows for declaration and assignment to be done on a single line.

<constdec>: Constant Declaration

A constant declaration is the token CONST followed by a type TYPE and an assignment <asgnmnt>. This is so that a constant cannot be reassigned a value after its declaration.

<funcdec>: Function Declaration

A function declaration starts with the token FUNC followed by a variable VARIABLE, then function arguments <funcargs> between the two tokens LP and RP for left and right parentheses, which is followed by a function statement <funcstmt> between the two tokens THEN and END. This forms the entire structure of a function, and allows for the inside of it to be filled with statements and terminated with a return statement <returnstmt>.

<funcargs>: Function Arguments

Function arguments can either be a type TYPE followed by a variable VARIABLE, or another list of function arguments <funcargs> followed by the token COMMA, which in turn is followed by a type TYPE and a variable VARIABLE. Alternatively, function arguments can be empty: <empty>.

<logicop>: Logic Operator

This is another nonterminal and named as such as it checks state and conditions. In logical expressions, there are many operators in this language. These operators are represented with tokens EQUAL, NOTEQUAL, BIGGEREQUAL, SMALLEREQUAL, SMALLERTHAN, BIGGERTHAN, AND and OR. EQUAL represents the equality operator with "==", NOTEQUAL represents the inequality operator with "!=", BIGGEREQUAL represents the bigger than or equal operator with ">=", SMALLEREQUAL represents the smaller than or equal operator with "<=", SMALLERTHAN represents the smaller than operator with "<", BIGGERTHAN represents the bigger than operator with ">", AND represents the and operator with "&&", and OR represents the or operator with "||".

<readDataFromSensor>: Read Data from Sensor

This non-terminal is present one of the predefined primitive function and it provides to read data from corresponding sensor.

<timestamp>: Time Stamp

This non-terminal is present one of the predefined primitive function. Timer that returns the current timestamp, the number of seconds elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970. This function returns the current timestamp.

<connectionOfInternet>: Internet Connection

This non-terminal consists of two basic functions which are related with internet. These functions are connect and disconnect. This is created to provide a mechanism to define a connection to a given URL.

<connect>: Connect

Internet connection is the core feature of IOT devices. This non-terminal provides to connect to the internet

<disconnect>: Disconnect

Internet connection is the core feature of IOT devices. This non-terminal provides to disconnect to the internet

<openSwitch>

This non-terminal is present one of the predefined functions. This function provides to open chosen switch <sw>.

<closeSwitch>

This non-terminal is present one of the predefined functions. This function provides to close chosen switch <sw>.

<sw>: Switch

This non-terminal defines predefined words and is written as a chosen switch CHOSENSWITCH. Each switch have corresponding name for determined 10 switches such as SW1, SW2, SW3 and so on. To increase extensibility, we allow the addition of new switches such as SW11.

<getURL>: Get URL

This non-terminal is also one of the predefined functions. This function provides to receive the given URL.

<sendURL>: Send URL

This non-terminal is one of the main predefined functions. This function provides to send an integer at a time to determined connection object.

<getTemperature>: Get Temperature

Temperature is an important primitive function of IOT devices. This non-terminal provides to receive real-time temperature from the device.

<getHumidity>: Get Humidity

Humidity is an important feature of some IOT devices. This non-terminal provides to receive humidity data from the device.

<getAirPressure>: Get Air Pressure

Air Pressure is a significant feature of some IOT devices. This non-terminal provides to receive air pressure the device.

<getAirQuality>: Get Air Equality

Air quality is more complex and fundamental function for our IOT device. This non-terminal provides result of air pressure calculation of the device.

<getLight>: Get Light

Light is also main components of this IOT device. This non-terminal provides to get light density of environment from the device.

<getSoundLevel>: Get Sound Level

Sound is a significant primitive function of the IOT device. This non-terminal provides to get sound level from the device.

<otherFunctionCalls>: Other Function Calls

A function does not have to be a primitive function. It can be add more functions to program. This non-terminal captures these functions which are not primitive ones. It is constructed with a variable VARIABLE followed by a parameter <param> between two parentheses left parenthesis LP and right parenthesis RP.

<param>: Parameter

A parameter is used in generic function calls, and is constructed with an expression <expression> or an expression <expression> followed by a comma COMMA and a helper <helper>. Alternatively,it can be empty <empty>.

<helper>: Helper

Helper is a non-terminal which is added to prevent an ambiguity problem in parameters. It is written as an expression <expression> followed by a comma COMMA and another helper <helper>. Otherwise, it can be constructed as a single expression <expression>.

<empty>: Empty

Empty is a statement without content added to cover for empty blocks in the code such as for functions without parameters and for empty programs.

Descriptions of Nontrivial Tokens

Comments

The language Garry++ supports single line comment with "##". It doesn't support multi line comments because while multi-line comments increase writability , they decrease readability. Also, nowadays, most of the IDEs provide multi-line comments serial single line comments.

İdentifiers

We decided that identifiers can only start with a letter and can be followed by alphanumeric characters for readability. This makes the identifiers easy to detect for programmers and makes it less prone to bugs.

Equality Operator

We used the keyword "is" as the equality operator to increase readability. This keyword makes the language closer to a natural language.

Predefined Symbols

```
alphabetic                  [A-Za-z]
digit                       [0-9]
INTEGER                     [+-]?{digit}+
STRING                      \".*\"
space                       [ \t]+
alphanumeric                ({alphabetic}|{digit})
FLOAT                                {digit}*(\.){digit}+
LOGIC                              (true|false)
TYPE                        (int|float|string|boolean|connection)
VARIABLE                    {alphabetic}{alphanumeric}*
DOT                         \.
LP                          \(
RP                          \)
SMALLERTHAN                 \<
BIGGERTHAN                  \>
BIGGEREQUAL                 \>\=
SMALLEREQUAL                \<\=
EQUAL                       \=\=
ASSIGN                      is
NOTEQUAL                    \!\=
WHILE                       while
```

```
FOR                     for
IF                      if
ELSE                    else
THEN                    then
END                     end
START                   start
NOT                     not
RETURN                  return
CONST                   const
AND                     \&\&
OR                      \|\|
COMMA                   ,
COMMENT                 ##.*
TIMESTAMP               timestamp
CONNECT                 connect
DISCONNECT              disconnect
SWITCHES                switches
READDATAFROMSENSOR      readDataFromSensor
GETURL                  getURL
SENDURL                 sendURL
FUNC                    func
EOL                     ;
GETTEMPERATURE          getTemperature
GETHUMIDITY             getHumidity
GETAIRPRESSURE          getAirPressure
GETLIGHT                getLight
GETSOUNDLEVEL           getSoundLevel
GETAIRQUALITY           getAirQuality
OPEN                    open
CLOSE                   close
CHOSENSWITCH            SW[0-9]+
FROM                    from
TO                      to
BY                      by
PLUS                    \+
MINUS                   \-
DIVIDE                  \/
MULT                    \*
REMAINDER               %
```

Example Program

```
## Mechanism to define a connection to a given URL
connection var;
var is connect("www.bilkent.edu.tr");
## if the user knows sensor number they can directly read data
firstSensorData is readDataFromSensor(var); ## 01 for temperature
string url is getURL(var);
sendURL(var,3) ## Mechanisms to send/receive an integer at a time,from/to a
connection
const int iotDevices is 7 ;
int timestampData is timestamp();
float level;
int y is getAirPressure();
close(SW3)
for(z from 0 to 7 by 1) start
    while( timestamp() < 123456789 )start
      if ( 0 == timestamp()) then
      start
      x is x+(y/z) ;
      end
      else
      start
      x is x/z+y ;
      end
    end
  level is getSoundLevel();
  int light is 7;
  int temperature is getTemperature();
   if (temperature > getTemperature()) then
     start
     light is getLight();
      end
     if (light != 3) then
         start
         boolean air is  getAirQuality(); ## if air quality is fine,true
         end
     else
      start
      ##A(light);
      end
    end
func A (int x) then
      x is x + 1;
    open(SW3)
    return x;
end
float humudity is getHumidity();
disconnect()
```