CS-315
PROGRAMMING LANGUAGES
SECTION-2
PROJECT 1
Vox

Ali Emre Aydogmus – 21901358

Deniz Berkant Demirors – 21902356

Yekta Seckin Satir – 21903227

1. BNF

1.1 Fundamentals & Statements

<program> ::= **INIT** <statements> **FINISH**

<statements> ::= <statement>
| <statement> <statements>
| <comment> <statements>

<comment> ::= **//** <str> <new-line> | **/*** <comment-line> */

<comment-line> ::= <str> <new-line> | <str> <new-line> <comment-line>

<statement> ::= <declaration statement> | <operation statement>**;**
| <receive statement>**;** | <send statement>**;**
| <if statement> | <loop statement> | <call statement>**;**

<declaration statement> ::= <variable declaration> **;** | <function declaration>

<operation statement> ::=
<variable declaration> <assignment operator> <expression>
| <variable > <assignment operator> <expression>
| <postfix expression> **++**
| <postfix expression> **--**
| <postfix expression> **>>**
| <postfix expression> **<<**

<receive statement> ::= **receive** <scan>

<scan> ::= <variable> | <scan> **+** <variable>

<send statement> ::= **send** <print>

<print> ::= <print content> | <print> **+** <print content>

<print content> ::= <variable> | <string_literal>

<if statement> ::= **if (** <expression> **)** <block> <elif statement>
| **if (** <expression> **)** <block> <elif statement> **else** <block>

<elif statement> ::= <elif statement> **elif (** <expression> **)** <block >
|

<loop statement> ::= <for statement> | <while statement>

<call statement> ::= <function call>
| <embedded function>

<function call> ::= <variable> (<parameters>)

<variable declaration> ::= <identifier> <variable>

<block> ::= **{** <statements> **}** | **{ }** | <expressions>**;**

1.2 Function Declaration

<function declaration> ::= <variable declaration>(<declaration parameters>)
{ <statements> **return** <expression>**; }**

<declaration parameters> ::= <variable declaration> , <declaration parameters>
| <variable declaration>
|

<parameters> ::= <parameter> , <parameters>
| <parameter>
|

<parameter> ::= <literal> | <call statement> | <variable>

<embedded function> ::= **getH( );**
| **getA( );**
| **getT( );**
| **vertical(** <numerical parameter> **)**
| **horizontal(** <numerical parameter> **)**
| **turn(** <bool parameter> **)**
| **spray(** <bool parameter> **)**
| **connect(** <bool parameter>)

<numerical parameter> ::= <integer literal> | <float literal>
| <angle literal> | <variable>

<bool parameter> ::= <bool literal> | <variable>

1.3 Loops

<while statement> ::= **while (** <expression> **)** <block>

<for statement> ::= **for (** <variable> **in** <variable>**)** <block>
| **for (** <variable declaration> **in** <variable>**)** <block>
| **for** (<for ops> ; <expressions> ; <for ops>) <block>

<for ops> ::= <operation statement> | <for ops>, <operation statement> |

1.4 Expressions

<expressions> ::= <expression> | <expressions>, <expression> |

<expression> ::= <qm expression>

<qm expression> ::= <logic expression>
| <logic expression> ? <expression> : <qm expression>

<logic expression> ::= <bitwise expression>
| <logic expression> **and** <bitwise expression>
| <logic expression> **&&** <bitwise expression>
| <logic expression> **or** <bitwise expression>
| <logic expression> **||** <bitwise expression>

<bitwise expression> ::= <equality expression>
| <bitwise expression> ^ <equality expression>
| <bitwise expression> **&** <equality expression>

<equality expression> ::= <relational expression>
| <equality expression> **==** <relational expression>
| <equality expression> **!=** <relational expression>

<relational expression> ::= <shift expression>
| <relational expression> **<** <shift expression>
| <relational expression> **>** <shift expression>
| <relational expression> **<=** <shift expression>
| <relational expression> **>=** <shift expression>

<shift expression> ::= <additive expression>
| <shift expression> **>>** <additive expression>
| <shift expression> **<<** <additive expression>

<additive expression> ::= <multiplicative expression>
| <additive expression> **+** <multiplicative expression>
| <additive expression> **-** <multiplicative expression>

<multiplicative expression> ::= <cast expression>
| <multiplicative expression> ***** <cast expression>

| <multiplicative expression> / <cast expression>
| <multiplicative expression> **%** <cast expression>

<cast expression> ::= <unary expression>
| (<identifier>) <cast expression>

<unary expression> ::= <postfix expression>
| **!** <postfix expression>
| **~** <postfix expression>
| **-** <postfix expression>
| **+** <postfix expression>

<postfix expression> ::= <primary expression>
| <primary expression> **++**
| <primary expression> **--**
| <primary expression> **>>**
| <primary expression> **<<**

<primary expression> ::= <literal> | <function call> | (<expression>) | <variable>

1.5 Identifiers & Tokens

<variable> ::= <letter><variable str> | <letter><variable str> [<integer literal>]

| <letter><variable str> [<angle literal>]

| <letter><variable str> [<letter><variable str>]

<identifier> ::= int | float | char | string | bool | angle

<literal> ::= <integer literal> | <float literal> | <char literal>
| <string literal> | <bool literal> | <angle literal>

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

<float literal> ::= <integer literal>.<integer literal>
| <integer literal>
| .<integer literal>

<char literal> ::= **‘**<any char>**’**

<string literal> ::= **“**<str>**”**

<bool literal> ::= **true** | **false**

<angle literal> ::= **1** <digit> <digit> | **2** <digit> <digit> | **3** <digit> <digit> |

| <non_zero digit> <digit> | <digit>

<str> ::= <any char> | <str> <any char> |

<variable str> ::= <alphanumeric> | <variable str> <alphanumeric>

<any char> ::= <alphanumeric> | <symbol> | <whitespace>

<whitespace> ::= ‘ ‘

<alphanumeric> ::= <digit> | <letter>

<letter> ::= 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
| 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

<digit> ::= 0 | <non_zero digit>

<non_zero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<new-line> ::= \n

<symbol> ::= <LP> | <RP> | <LSB> | <RSB> | <LCB> | <RCB> | <smaller op>
| <bigger op> | <and op> | <or op> | <xor op> | <plus> | <minus> | <qm>
| <multiplication op> | <division op > | <underscore> | <eq operator>
| <comma> | <colon> | <dot>

<assignment operator> ::= <eq operator> | <plus eq operator> | <minus eq operator>
| <multiplied eq operator> | <division eq operator>
| <xor eq operator> | <and eq operator> | <or eq operator>

<eq operator> ::= =

<plus eq operator> ::= +=

<minus eq operator> ::= -=

<multiplied eq operator> ::= *=

<division eq operator> ::= /=

<xor eq operator> ::= ^=

<and eq operator> ::= &=

<or eq operator> ::= |=

<LP> ::= (

<RP> ::=)

<LSB> ::= [

<RSB> ::=]

<LCB> ::= {

<RCB> ::= }

<smaller op> ::= <

<bigger op> ::= >

<or op> ::= |

<and op> ::= &

<xor op> ::= ^

<plus> ::= +

<minus> ::= -

<multiplication op> ::= *

<division op> ::= /

<underscore> ::= _

<qm> ::= ?

<double eq op> ::= ==

<not eq op> ::= !=

<less than eq op> ::= <=

<bigger than eq op> ::= >=

<comma> ::= ;

<colon> ::= :

<dot> ::= .

1.6 Keywords

receive | send | if | elif | else | return | while | for | in | ; | true | false | int | float | char | string | bool | angle | + | getH | getA | getT | vertical | horizontal | turn | spray | connect | INIT | FINISH

2. Explanation & Description

<program> ::= **INIT** <statements> **FINISH**

- A program in the language is a set of statements, preceded by **INIT** and followed by **FINISH**.

<statements> ::= <statement>
| <statement> <statements>
| <comment> <statements>

- Statements are recursively defined to be made up of 2 types of components. Comments and program statements which will be explained.

<comment> ::= **//** <str> <new-line> | **/*** <comment-line> */

- The language has two ways to write comments with. Statements that start with // will be skipped until the end of the line. Statements that start with /* will be skipped until a */ is detected.

<comment-line> ::= <str> | <str> <new-line> | <str> <new-line> <comment-line>

- Comment-line is defined recursively to be able to span multiple lines.

<statement> ::= <declaration statement> | <operation statement>**;**
| <receive statement>**;** | <send statement>**;**
| <if statement> | <loop statement> | <call statement>**;**

- Statements are building blocks of a program in the language. Those might be operations, calls, receive/send statements, conditions and loops and declarations.

<declaration statement> ::= <variable declaration> **;** | <function declaration>

- A declaration might be for a function or a variable.

<operation statement> ::=
<variable declaration> <assignment operator> <expression>
| <variable > <assignment operator> <expression>
| <postfix expression> **++**
| <postfix expression> **--**
| <postfix expression> **>>**
| <postfix expression> **<<**

- An operation might be an assignment with an assignment operator. The assignment might be done to a variable, variable declaration or via a postfix expression.

<receive statement> ::= **receive** <scan>

- Receive statement is a readable way of scanning data. It is used with a scan component.

<scan> ::= <variable> | <scan> + <variable>

- Scan component is like a list of variables to be assigned to the scanned data in receive statement. An example use is:

```
int x;
int y;
receive x + y;
```

<send statement> ::= **send** <print>

- Send statement is a readable way of outputting data.

<print> ::= <print content> | <print> + <print content>

- Print component is like a list of print contents to be outputted in send statement.

<print content> ::= <variable> | <string literal>

- Print content may be a variable or a string literal.

<if statement> ::= **if (** <expression> **)** <block> <elif statement>
| **if (** <expression> **)** <block> <elif statement> **else** <block>

- The language uses the if statements according to the tradition. It is an if followed by an expression to check in parenthesis, then a block of code, optionally followed by an else with a block of code.

<elif statement> ::= <elif statement> **elif (** <expression> **)** <block >
|

- In the language elif component is the shortcut for else if. It comes after a conditional statement and has a condition check itself to execute its followed block of code.

<loop statement> ::= <for statement> | <while statement>

- Two types of loops exist in the language. While and For loops.

<call statement> ::= <function call>
| <embedded function>

- A call statement is a function call, either user-defined or embedded.

<function call> ::= <variable> (<parameters>)

- A function call is done by calling its name (variable) followed by parameters in parenthesis.

<variable declaration> ::= <identifier> <variable>

- Variables are declared by a name (variable) preceded by its identifier.

<block> ::= **{** <statements> **}** | **{ }** | <expressions>**;**

- Block component is a block of code surrounded by curly brackets or a single statement, designed to be followed by if/else and loop statements.

<function declaration> ::= <variable declaration>(<declaration parameters>)
{ <statements> **return** <expression>**; }**

- Declaration takes one variable declaration for name and declaration parameters for function parameters. In brackets, it has statements and it must have a return value.

<declaration parameters> ::= <variable declaration> , <declaration parameters>
| <variable declaration>

|

- Declaration is either empty, one variable declaration or recursively variable declaration and declaration parameters

<parameters> ::= <parameter> , <parameters>
| <parameter>
|

- Parameters are either one parameter or multiple parameters with recursion

<parameter> ::= <literal> | <call statement> | <variable>

- Parameter either is a literal or a call statement (literal and call statement are explained below)

<embedded function> ::= **getH( );**
| **getA( );**
| **getT( );**
| **vertical(** <numerical parameter> **)**
| **horizontal(** <numerical parameter> **)**
| **turn(** <bool parameter> **)**
| **spray(** <bool parameter> **)**
| **connect(** <bool parameter>)

- Built in functions for language

<numerical parameter> ::= <integer literal> | <float literal>
| <angle literal> | <variable>

- Numerical parameter for type specification in embedded functions.

<bool parameter> ::= <bool literal> | <variable>

- Boolean parameter for type specification in embedded functions.

<while statement> ::= **while (** <expression> **)** <block>

- A loop which runs the block until the expression becomes false

<for statement> ::= **for (** <variable> **in** <variable>**)** <block>
| **for (** <variable declaration> **in** <variable>**)** <block>
| **for** (<for ops> ; <expressions> ; <for ops>) <block>

- Two iterative loops where in one variable can be declared but in other variable must be declared earlier, then iterates over the items in the iterator variable executing the block. On the last loop, it is a classical loop where variable is declared and run until expression becomes wrong with for ops.

<for ops> ::= <operation statement> | <for ops>, <operation statement> |

- It is either one operation statement or more than one with recursion.

<expressions> ::= <expression> | <expressions>, <expression> |

- The component expressions is recursively defined to have 0 or more expression components.

<expression> ::= <qm expression>

- An expression is defined in a hierarchy as the following to follow the order of operations. **Primary** expressions are parsed first, followed by **postfix** expressions, then **unary** expressions, then **cast** expressions, then **multiplicative** expressions, then **additive** expressions, then **shift** expressions, then **relational** expressions, then **equality** expressions, then **bitwise** expressions, then **logic**, and lastly **qm** expressions.

<qm expression> ::= <logic expression>
| <logic expression> ? <expression> : <qm expression>

- Ternary expression.

<logic expression> ::= <bitwise expression>
| <logic expression> **and** <bitwise expression>
| <logic expression> **&&** <bitwise expression>
| <logic expression> **or** <bitwise expression>
| <logic expression> **||** <bitwise expression>

- Boolean logic expression.

<bitwise expression> ::= <equality expression>
| <bitwise expression> **^** <equality expression>
| <bitwise expression> **&** <equality expression>

- Bitwise logic expression.

<equality expression> ::= <relational expression>
| <equality expression> **==** <relational expression>
| <equality expression> **!=** <relational expression>

- Expression to check equality.

<relational expression> ::= <shift expression>
| <relational expression> **<** <shift expression>
| <relational expression> **>** <shift expression>
| <relational expression> **<=** <shift expression>
| <relational expression> **>=** <shift expression>

- Expression for comparisons.

<shift expression> ::= <additive expression>
| <shift expression> **>>** <additive expression>
| <shift expression> **<<** <additive expression>

- Bit shifting expression.

<additive expression> ::= <multiplicative expression>
| <additive expression> **+** <multiplicative expression>
| <additive expression> **-** <multiplicative expression>

- Arithmetic addition and subtraction expression

<multiplicative expression> ::= <cast expression>
| <multiplicative expression> ***** <cast expression>
| <multiplicative expression> **/** <cast expression>
| <multiplicative expression> **%** <cast expression>

- Arithmetic multiplication, division and modular expression.

<cast expression> ::= <unary expression>
| (<identifier>) <cast expression>

- Expression to provide priority for parenthesis.

<unary expression> ::= <postfix expression>
| **!** <postfix expression>
| **~** <postfix expression>
| **-** <postfix expression>
| **+** <postfix expression>

- Expression that can have prefixes

<postfix expression> ::= <primary expression>
| <primary expression> **++**
| <primary expression> **--**
| <primary expression> **>>**
| <primary expression> **<<**

- Expression that can have postfixes

<primary expression> ::= <literal> | <function call> | (<expression>) | <variable>

- Basic expressions that have the most priority in the language.

<variable> ::= <letter><variable str> | <letter><variable str> [<integer literal>]

| <letter><variable str> [<angle literal>]

| <letter><variable str> [<letter><variable str>]

- Variables must start with a letter and name can be continued with any string. Also, the variable can be a one-dimensional array.

<identifier> ::= int | float | char | string | bool | angle

- Types for variables

<literal> ::= <integer literal> | <float literal> | <char literal>
| <string literal> | <bool literal> | <angle literal>

- A general set of literals

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

- Integer literal is either a digit or multiple digits with recursion

<float literal> ::= <integer literal>.<integer literal>
| <integer literal>
| .<integer literal>

- Float literal is either an integer literal, a decimal point with fraction or an integral and fractional part with decimal point between.

<char literal> ::= **‘**<any char>**’**

- Any char

<string literal> ::= **“**<str>**”**

- A string literal is any string inside quotation marks.

<bool literal> ::= **true** | **false**

- Either true or false

<angle literal> ::= **1** <digit> <digit> | **2** <digit> <digit> | **3** <digit> <digit> |
| <non_zero digit> <digit> | <digit>

- It is integers between 0 and 360(excluded)

<str> ::= <any char> | <str> <any char> |

- Any char or char array

<variable str> ::= <alphanumeric> | <variable str> <alphanumeric>

- Either one alphanumeric or alphanumerics lined with recursion

<any char> ::= <alphanumeric> | <symbol>

<alphanumeric> ::= <digit> | <letter>

<letter> ::= 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
| 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

- All letters in English alphabet

<digit> ::= 0 | <non_zero digit>

- Integers between 0 and 9

<non_zero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Non zero integers between 1 and 9

<new-line> ::= \n

- New line

<symbol> ::= <LP> | <RP> | <LSB> | <RSB> | <LCB> | <RCB> | <smaller op>
| <bigger op> | <and op> | <or op> | <xor op> | <plus> | <minus> | <qm>
| <multiplication op> | <division op > | <underscore> | <eq operator>
| <comma> | <colon> | <dot>

- Symbol set for operations and brackets

<assignment operator> ::= <eq operator> | <plus eq operator> | <minus eq operator>
| <multiplied eq operator> | <division eq operator>
| <xor eq operator> | <and eq operator> | <or eq operator>

- Operations for assignments which includes bitwise operations

<eq operator> ::= =

- Left hand side equals to right hand side

<plus eq operator> ::= +=

- Left hand side is equal to itself plus right hand side

<minus eq operator> ::= -=

- Left hand side is equal to itself minus right hand side

<multiplied eq operator> ::= *=

- Left hand side is equal to itself times right hand side

<division eq operator> ::= /=

- Left hand side is equal to itself divided right hand side

<xor eq operator> ::= ^=

- Exclusive or

<and eq operator> ::= &=

- Bitwise and equal op

<or eq operator> ::= |=

- Bitwise or equal op

<LP> ::= (

<RP> ::=)

<LSB> ::= [
<RSB> ::=]
<LCB> ::= {
<RCB> ::= }

- Symbols in bracket set

<or op> ::= |
<and op> ::= &
<xor op> ::= ^

- Bitwise operators

<plus> ::= +
<minus> ::= -
<multiplication op> ::= *
<division op> ::= /

- Mathematical operations

<underscore> ::= _
<qm> ::= ?

- Symbols

<smaller op> ::= <
<bigger op> ::= >
<double eq op> ::= ==
<not eq op> ::= !=
<less than eq op> ::= <=
<bigger than eq op> ::= >=
<comma> ::= ,
<colon> ::= :
<dot> ::= .

- Comparing operators

receive | send | if | elif | else | return | while | for | in | ; | true | false | int | float | char | string | bool | angle | + | getH | getA | getT | vertical | horizontal | turn | spray | connect | INIT | FINISH

- These are defined as keyword so no other string can override and contain them

3. Evaluation

3.1 **Readability**

The language can be considered readable since the feature multiplicity is minimal. Overloading of the operators are not too much. It has only variables and function declarations with comments and loops. It does not have a main function so the file runs easily when wanted. Number of keywords are minimal with self-explanatory names which makes them intuitive for the user.

3.2 **Writability:**

Vox is very compact. It has some number of primitive types, a basic definition of arrays and very specific rules to combine them. Therefore, it is both orthogonal and simple to an extent.

Vox has a fluctuating expressivity. For the small pilot area of sprinkler-drone command, Vox is very simple and sufficient, however it is not very suitable for any project bigger than this, or even for a drone with different type of ability (e.g.: a military drone, a delivery drone) without any modification to the language. Primitive operators and functionalities are included in the language, as well as basic movement and connection functions for the drone, however for any project in a bigger scope, more functionality in the language is definitely needed.

Vox does not support abstraction at all, since there are no unions, structs, or classes. The reason why Vox is so deprived of abstraction is because the aim of the language was to be as simple as possible.

In general, Vox is a language that tries to be simple, however some at some parts this simplicity is discarded for the sake of user's intuition, assuming that most users of Vox will be people who are already engaged with programming.

3.3 **Reliability**

The language is not so reliable for bigger projects. It lacking type checking is the most severe reason for this. For example, arrays and their iterations by for loops might be risky to use unless the programmer is very aware of what they are dealing with. The language provides no exception handling. But on the bright side, the language allows no aliasing. This is due to its featurelessness that also helps with readability and writability of the language.