CS 315 Programming Languages
Project 2 Report

DOT Language Design

TEAM 20

Sertaç **D**erya - 22003208 - Section 1
Bilginer **O**ral - 22103163 - Section 2
Halil **T**ataroğlu - 22102198 - Section 2

Table of Contents

1. Introduction

In this report, the aim is to introduce the programming language called “DOT”. We named the language DOT, as each letter stands for the initial letter of the team members’ surname. This report will provide detailed information about BNF description of the language. This report also explains our programming language's key elements, like variables and terminals, and describes nontrivial tokens (comments, identifiers, etc.), discussing how they impact readability, writability, and reliability.

2. BNF Description

The complete description of DOT language is given below in BNF format. The rules are hyperlinked to their definitions, therefore you may want to click on them whenever you want to see the definition.

2.1) Program Definitions:

```
<program>  ::=  <stmt_list>

<stmt_list>::=  “ ”                          |
                <stmt> <SC> <stmt_list>      |
                <block_stmt> <stmt_list>     |
                <comment> <stmt_list>


<block_stmt> ::=  <if_stmt> | <while_stmt> | <for_stmt> |
                  <func_dec_stmt>

<stmt>     ::=  <assign_stmt>          |
                <declaration_stmt>     |
                <output_exp>           |
                <expression_stmt>      |
                <return_stmt>

<block>    ::=  <LCB> <stmt_list> <RCB>

<comment>  ::=  <HASHTAG> <complex_str> <HASHTAG>
```

2.2) Statement Definitions:

```
<assign_stmt>     ::= <id> <EQ> <expression_stmt> |
                      <id> <LSB> <expression_stmt> <RSB> <EQ>
                      <expression_stmt>

<declaration_stmt>::= <dec_w_assign>  |
                      <dec_wo_assign>

<dec_w_assign>    ::= int <id> <EQ> <expression_stmt> |
                      int_arr <id> <LSB> <expression_stmt> <RSB>
                      <EQ> <array>

<dec_wo_assign>  ::=  int <id> |
                      int_arr <id> <LSB> <expression_stmt> <RSB>

<return_stmt>    ::=  return <expression_stmt>

<expression_stmt>::=  <arit_exp>

<if_stmt>        ::= if <LP> <expression_stmt> <RP> <block> |
                     if <LP> <expression_stmt> <RP> <block> else
                     <block>

<while_stmt>     ::= while <LP> <expression_stmt> <RP> <block>

<for_stmt>       ::= for <LP> <dec_w_assign> <SC>
     <expression_stmt> <SC> <assign_stmt> <RP> <block>      |
                      for <LP> <assign_stmt> <SC>
     <expression_stmt> <SC> <assign_stmt> <RP> <block>
```

2.3) Expression Definitions:

```
<arit_expr>      ::= <arit_expr> <OR> <l7_expr>  |
                     <l7_expr>

<l7_expr>        ::= <l7_expr> <AND> <l6_expr>   |
                     <l6_expr>

<l6_expr>        ::= <l6_expr> <EQUALS> <l5_expr>      |
                    <l6_expr> <NOT_EQUALS> <l5_expr>  |
                    <l5_expr>

<l5_expr>        ::= <l5_expr> <LT> <l4_expr>    |
                     <l5_expr> <LTE> <l4_expr>   |
                     <l5_expr> <GT> <l4_expr>    |
                     <l5_expr> <GTE> <l4_expr>   |
                     <l4_expr>
```

```
<l4_expr>         ::= <l4_expr> <PLUS> <l3_expr>   |
                      <l4_expr> <MINUS> <l3_expr> |
                      <l3_expr>

<l3_expr>         ::= <NOT> <l3_expr> | <l2_expr>

<l2_expr>         ::= <l2_expr> <MUL> <l1_expr>    |
                      <l2_expr> <DIV> <l1_expr>    |
                      <l2_expr> <MOD> <l1_expr>    |
                      <l1_expr>

<l1_expr>         ::= <l1_expr> <EXP> <l0_expr>|
                      <l0_expr>

<l0_expr>         ::=  <LP> <arit_expr> <RP> |
                       <id>                  |
                       <signed_int>          |
                       <func_call>           |
                       <input_exp>           |
                       <id> <LSB> <expression_stmt> <RSB>
```

2.4) IO Definitions:

```
<input_exp>       ::= read <LP> <RP>

<output_exp>      ::= echo <LP> <DQ> <complex_str> <DQ> <RP> |
                      echo <LP> <expression_stmt> <RP>
```

2.5) Symbol Definitions:

```
<DQ>          ::=  “
<RCB>         ::=  }
<LCB>         ::=  {
<LP>          ::=  (
<RP>          ::=  )
<PLUS>        ::=  +
<MINUS>       ::=  -
<MUL>         ::=  *
<DIV>         ::=  /
<MOD>         ::=  %
<EXP>         ::=  **
<AND>         ::=  &
<OR>          ::=  |
<NOT>         ::=  !
<EQUALS>      ::=  ==
<NOT_EQUALS>::=  !=
<LT>          ::=  <
```

```
<LTE>       ::=  <=
<GT>        ::=  >
<GTE>       ::=  >=
<SC>        ::=  ;
<EQ>        ::=  =

<digit>     ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2.6) Function Definitions:

```
<func_dec_stmt>::= func <id> <LP> <fd_parameters> <RP> <block>

<func_call>    ::= <id> <LP> <fc_parameters> <RP>
```

2.7) Type Definitions:

```
<params>       ::= int <id> | int_arr <id> <LSB> <RSB> |
                   int <id> <COM> <fd_parameters> |
                   int_arr <id> <LSB> <RSB> <COM> <fd_parameters>

<fd_parameters>::= “ ” | <params>

<fc_parameters>::= “ ” | <expression_stmt> |
                   <expression_stmt> <COM> <fc_parameters>

<id>           ::= <string>

<array>      ::= <LSB> <array_elements> <RSB>

<alphanumeric> ::= <normal_chars> <alphanumeric> |
                   <normal_chars> |
                   <digit> <alphanumeric> |
                   <digit>

<array_elements>::= “ ” | <expression_stmt> |
                   <expression_stmt> <COM> <array_elements>

<signed_int>   ::= <pos_int> | <neg_int>

<neg_int>      ::= <MINUS> <integer>

<pos_int>      ::= <PLUS> <integer>| <integer>

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

<string>       ::= <normal_chars> | <normal_chars> <string>
```

```
<complex_str>  ::= <alphanumeric> <complex_str> |
                   <special_chars> <complex_str> |
                   <alphanumeric> | <special_chars>

<normal_chars> ::=
      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|_

<special_chars>::=
       <NOT> | @ | # | $ | <MOD> | ^ | <AND> | <OR> | <LP> | <RP>
       | <PLUS> | <EQ> | <DIV> | <MUL> | <MINUS> | ' | <DQ> | <SC>
       | '|' | <LCB> | <RCB> | <LSB> | <RSB> | ! | & | *
```

3. Explanation

- **<program> ::= <stmt_list**

This defines the core structure of a program in the language, stating that a program consists of a sequence of statements.

- **<stmt_list>::= “ “**
  <block_stmt> <stmt_list> |
  <stmt> <SC> <stmt_list> |
  <comment> <stmt_list>

A statement list is composed of statements, block statements or comments. Statement list can also be empty, representing an empty program.

- **<block_stmt> ::= <if_stmt> | <while_stmt> | <for_stmt> |**
  <func_dec_stmt>

Block statements are special statements that have **<block_stmt>** in them. They are also separate from normal statements because they don’t have **<SC>** after them.

- **<stmt> ::= <assign_stmt> | <declaration_stmt> |**
  <output_exp> | <expression_stmt> |
  <return_stmt>

There are various statements such as assignments, declarations, return statements, output expressions and expression statements.

- **<block> ::= <LCB> <stmt_list> <RCB>**

Blocks are used to group multiple statements together, just like programs. But a block is specifically used in if statements, else statements, loops, and functions.

- **<comment> ::= <HASHTAG> <complex_str> <HASHTAG>**

Any alphanumeric value written between hashtags (**#**) is regarded as a comment. Comments are a part of the statement list. **<complex_str>** represents all values except newline value, because of this it is used to create comments.

- **<assign_stmt> ::= <id> <EQ> <expression_stmt> | <id> <LSB> <expression_stmt> <RSB> <EQ> <expression_stmt>**

This is used to carry out a very fundamental feature which is assigning a value to an already declared variable.

- **<declaration_stmt>::= <dec_w_assign> | <dec_wo_assign>**

We separated the declaration into two: declaration with assignment and declaration without assignment. We are using declaration without assignment in function declarations.

- **<dec_w_assign> ::= int <assign_stmt> | int_arr <id> <LSB> <expression_stmt> <RSB> <EQ> <array>**

This is the case of a declaration where the assignment is made at the same time.

- **<dec_wo_assign> ::= int <id> | int_arr <id> <LSB> <integer> <RSB>**

This is the case of a declaration where the assignment doesn't take place. This is used in function declarations.

- **<expression_stmt>::= <arit_exp>**

Everything that returns a value is denoted as an expression in the DOT language.

- **<return_stmt> ::= return <expression_stmt>**

As expressions return a value, the reserved word return is used with **<expression_stmt>** to return a value.

()	level 0 precedence
**	level 1 precedence
*	level 2 precedence
/	
%	
+	level 3 precedence
-	
!	
<	level 4 precedence
<=	
>	
>=	
==	level 5 precedence
!=	
&	level 6 precedence
|	level 7 precedence

In arithmetic operations, some operations have precedence over the other one. As this was the case, the following expressions are designed according to that precedence rule.

- **<arit_expr>** **::= <arit_expr> <OR> <l7_expr> | <l7_expr>**

<arit_expr> consists of all the values that return a value. At the end; variables, constant integers, func_calls that return an integer, and input_exp that take integers can be used as operands. This also contains logic operations, conditional operations, and arithmetic operations.

```
<l7_expr>        ::= <l7_expr> <AND> <l6_expr> |
                     <l6_expr>

<l6_expr>        ::= <l6_expr> <EQUALS> <l5_expr>      |
                     <l6_expr> <NOT_EQUALS> <l5_expr> |
                     <l5_expr>

<l5_expr>        ::= <l5_expr> <LT> <l4_expr>    |
                     <l5_expr> <LTE> <l4_expr>   |
                     <l5_expr> <GT> <l4_expr>    |
                     <l5_expr> <GTE> <l4_expr>   |
                     <l4_expr>

<l4_expr>        ::= <l4_expr> <PLUS> <l3_expr>  |
                     <l4_expr> <MINUS> <l3_expr> |
                     <l3_expr>

<l3_expr>        ::= <NOT> <l3_expr> | <l2_expr> |
```

Not operation converts any non-zero integer to zero, while converting zero to value 1 similar to C++. This enhances writability as it can be used in various places such as loops and if statements.

```
<l2_expr>        ::= <l2_expr> <MUL> <l1_expr>   |
                     <l2_expr> <DIV> <l1_expr>   |
                     <l2_expr> <MOD> <l1_expr>   |
                     <l1_expr>

<l1_expr>        ::= <l1_expr> <EXP> <l1_expr> |
                     <l0_expr>

<l0_expr>        ::=  <LP> <arit_expr> <RP> |
                      <id>                  |
                      <signed_int>          |
                      <func_call>           |
                      <input_exp>           |
                      <id> <LSB> <expression_stmt> <RSB>
```

In this point any value can be used as an operand as it is the most basic precedence level. Users can put parentheses to create higher precedence for certain operations, users can also use <func_call> and <input_exp> to return a value from a function and get a value from the command line to use respectively.

- **<if_stmt> ::= if <LP> <expression_stmt> <RP> <block> |**
  if <LP> <expression_stmt> <RP> <block> else <block>

<if_stmt> is defined using blocks, it may use an else statement so this is also given as a possibility.

- **<while_stmt> ::= while <LP> <expression_stmt> <RP> <block>**

<while_stmt> is also defined with a code block which is dependent on an expression.

- **<for_stmt> ::= for <LP> <dec_w_assign> <SC>**
  <expression_stmt> <SC> <assign_stmt> <RP> |
  for <LP> <assign_stmt> <SC>
  <expression_stmt> <SC> <assign_stmt> <RP>

<for_stmt> is similar to "**while**" however for convenience **<for_stmt>** includes assignment and declaration parts so that the user can declare and assign the loop variable.

- **<input_exp> ::= read <LP> <RP>**

<input_exp> lets users take values from the user. And all this statement needs is read().

- **<output_exp> ::= echo <LP> <DQ> <complex_str> <DQ> <RP> |**
  echo <LP> <expression_stmt> <RP>

Output statement lets us communicate with the terminal. It prints out an <complex_str> value and this value needs to be between double quotes. It can also print the result of an expression.

```
<DQ>         ::=  "
<RCB>        ::=  }
<LCB>        ::=  {
<LP>         ::=  (
<RP>         ::=  )
<PLUS>       ::=  +
<MINUS>      ::=  -
<MUL>        ::=  *
<DIV>        ::=  /
<MOD>        ::=  %
<EXP>        ::=  **
<AND>        ::=  &
<OR>         ::=  |
<NOT>        ::=  !
<EQUALS>     ::=  ==
<NOT_EQUALS>::=  !=
<LT>         ::=  <
```

```
<LTE>       ::=  <=
<GT>        ::=  >
<GTE>       ::=  >=
<SC>        ::=  ;
<EQ>        ::=  =
```
These are the symbols used for various different purposes.

- **<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

Digits can be from 0 through 9.

- **<func_dec_stmt>::=**
  func <id> <LP> <fd_parameters> <RP> <block>

A function is declared using this. There is a specific reserved word “func” that needs to be used before the function name. For taking function parameters fd_parameters is used. Also, the function code will be written in a block.

- **<func_call> ::= <id> <LP> <fc_parameters> <RP>**

The function is called using the function’s name and then parentheses, between the parentheses the function parameters will be written.

- **<params> ::= int <id> | int_arr <id> <LSB> <RSB> |**
  int <id> <COM> <fd_parameters> |
  int_arr <id> <LSB> <RSB> <COM> <fd_parameters>

params is used in the function definitions part. These specifically have their type names in front of them and are also separated by commas.

- **<fd_parameters>::= “ “ | <params>**

There can be no parameters declared in the function, however, if there are parameters declared, params are used.

- **<fc_parameters>::= “ “ |**
  <expression_stmt> |
  <expression_stmt> <COM> <fc_parameters>

According to the declaration, the function is called with necessary parameters. There can also be an empty function call.

- **<id> ::= <string>**

In DOT language identifiers are made of strings. Users of the language cannot use strings in their programming logic since strings are only used for identifiers.

- **<array> ::= <LSB> <array_elements> <RSB>**

<array> is defined as a group of integers.

- **<integer> ::= <integer> <digit> | <digit>**

<integer> represents a numerical part of an **int** value. **<integer>**s can also represent **<pos_int>**.

- **<signed_int> ::= <pos_int> | <neg_int>**

<signed_int> is the union of negative and positive int.

- **<neg_int> ::= <MINUS> <integer>**

<neg_int> is an integer with a minus sign, so a negative integer is called **<neg_int>**.

- **<pos_int> ::= <PLUS> <integer> | <integer>**

<pos_int> is an integer with a plus sign or an integer without any sign. This can be used in different places which require positive numbers. This also includes number 0 as **<integer>** is included here, which implies writing "**+0**" is possible in DOT.

- **<alphanumeric> ::= <normal_chars> <alphanumeric> |**
  <normal_chars> |
  <digit> <alphanumeric> |
  <digit>

<alphanumeric> type consists of strings that can be created with integers and normal_chars.

- **<array_elements>::= " " | <expression_stmt> |**
  <expression_stmt> <COM> <array_elements>

<array_elements> is a group of zero or more elements used in array assignments. Individual elements are separated with commas and written in between square brackets.

- **<string> ::= <normal_chars> | <normal_chars> <string>**

<string> can be interpreted as a combination of chars. In this part <string> is defined either as a character or character combined with a string.

- **<complex_str> ::= <alphanumeric> <complex_str> |**
  <special_chars> <complex_str> |
  <alphanumeric> | <special_chars>

<complex_str> is used specifically for output statements. Output statements may need various kinds of symbols, so this type of string is used.

- **<normal_chars> ::=**
  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|_

The **<normal_chars>** set includes all lowercase letters (**a** to **z**), uppercase letters (**A** to **Z**), and the underscore (_) character.

- **<special_chars>::=**
  <NOT> | @ | # | $ | <MOD> | ^ | <AND> | <OR> | <LP> | <RP>
  | <PLUS> | <EQ> | <DIV> | <MUL> | <MINUS> | ' | <DQ> | <SC>
  | '|' | <LCB> | <RCB> | <LSB> | <RSB> | ! | & | *

The **<special_chars>** set consists of various special characters, such as exclamation mark (**!**), at symbol (**@**), hash symbol (**#**), dollar sign (**$**), percent sign (**%**), caret (**^**), ampersand (**&**), asterisk (*****), left parenthesis (**(**), right parenthesis

( )), plus sign (+), equal sign (=), forward slash (/), minus sign (-), single quote ('), double quote ("), semicolon (;), vertical bar (|), curly braces ({ and }), and square brackets ([and]).

4. Token Evaluation

4.1) Reserved Words:

In the context of the programming language/system, the words “func”, “read”, “echo”, "for," "if," "while," "return," "else," "int," and "int_arr" are considered reserved words, each serving a specific purpose in the code structure. Those common words are chosen to increase readability and writability because they are memorable.

4.2) Comments:

The definition of comments in this context involves using hashtags (denoted as <HASHTAG>) to enclose complex_str text (denoted as <complex_str>) within the code, effectively making it a part of the statement list. Comments serve as annotations within the code to explain or provide context for specific sections. They contribute to readability by enhancing code documentation, making it easier for developers to understand the purpose of various code segments. Comments also promote writability as they help developers communicate their intentions clearly, making it easier to write and maintain code. Furthermore, comments improve reliability by aiding in code maintenance and debugging, as they enable quick identification of potential issues or the purpose of specific code sections. In summary, comments, defined in the manner described, positively impact the readability, writability, and reliability of the codebase.

4.3) Identifiers:

In the program we designed, we didn't incorporate the use of string data types as they were not included for user input or data manipulation. Instead, we utilized strings as a means to represent and define identifiers within the code. These strings are essentially sequences of characters, including letters and underscores. Serving as labels for variables, functions, or other program elements. By using strings and underscore symbols for identifiers, we maintained a simplified and consistent structure throughout the code. This approach helped enhance code readability and made it easier to understand the purpose and naming of various program elements, even in the absence of explicit string manipulation or user input functions.

4.4) Literals:

In our program, literals played a crucial role in representing fixed and unchanging values directly within the source code. These values, such as numbers, characters, and constants, were used to provide specific, unalterable data to various parts of our program. For instance, when we declare "int count = 10;" the value "10" is an example of literal. These literals not only provided data for calculations and comparisons but also improved code readability by making our intentions clear and easily identifiable. Their immutability ensured that these values remained constant throughout the program's execution, enhancing the reliability of our code. Also, for int_arr declarations we used integers separated by commas to declare array which is easy in terms of writability.

5. Analysis

5.1) Readability:

Overall, the program was designed so that it will be easy to organize. Because of this, we didn't use a Python style indentation type organization but we chose to use curly brackets to separate between loops, if statement, and function definitions. Also, we decided not to implement some shortcuts, such as increment and decrement operators, which in our opinion is a plus in terms of readability because the user might not know the operation's precedence. Furthermore, the reserved tokens we use are commonly used in other languages and they are understandable.

5.2) Writability:

In terms of writability because of the increment and decrement operators not being implemented there is a decrease in writability but we choose readability over writability in this case. We didn't introduce any other loops or branch statements but the ones we implemented are easy to use and write.

5.3) Reliability:

Our language is designed to be reliable specifically for calculations of logic, arithmetic, and conditional types. The operator precedences are clearly set and the user can write and calculate any statement they want given that it is mathematically correct. The block statement used for loops, branches, and functions also increases reliability. We don't currently have any type checking, exception handling, etc. but these would make the language more complicated so our features should be enough for a basic language.

6. Language Details

6.1) Beginning of the Execution:

Our language starts execution from the beginning of the file like Python. Users don't have to write their programs in a specific function or enter special keywords for the beginning of the program. This increases writability as users don't have to follow special instructions to begin their program.

6.2) Parameter Passing Methods:

DOT language uses pass by value for function parameters passings. Integers and array are all passed by their values.

7. Test Codes

7.1) Test 1 (with syntax error):

```
int ok = 0;

while (!ok) {
        echo("Enter x: ");
        int x = read();

        echo("\nEnter y: ");
        int y = read();

        echo("\nEnter z: ");
        int z = read();

     # syntax error here, missing ‘(‘ from in if statement #
        if (x == 0) | (y == 0) | (z == 0)) {
                echo("You should enter nonzero values\n");
        }
        else {
                ok = 1;
        }
}

echo("(x times y times z) is: ");
echo (x * y * z);
echo("\n");
```

7.2) Test 2:

```
func foo(int p, int q) {
	echo("foo\n");
	echo("p= ");
	echo(p);
	echo(", q= ");
	echo(q);
	echo("\n");

	if (p > q) {
		return p;
	}
	return q;
}

int_arr a[4] = [5, 0, 3, -7];
int_arr b[3] = [9, -2, -1];

for (int i = 0; i < 4; i = i + 1) {
	for (int j = 0; j < 3; j = j + 1) {
		int c = foo(a[i], b[j]);
		echo("a= ");
		echo(a[i]);
		echo(", b= ");
		echo(b[j]);
		echo(", c= ");
		echo(c);
		echo("\n");
	}
}
```

7.3) Test 3 (with syntax error):

```
func printArr(int_arr arr, int n) {
	for (int i = 0; i < n; i = i + 1) {
		echo(arr[i]);
		echo(", ");
	}
	echo("\n");
}

int size = 7;

# syntax error here, missing ‘)’ #
int_arr myArr[size] = [1, -2, (4 + 2, 22, 9, 14, 13];
printArr(myArr, size);

int threshold = 10;
```

```
int count = 0;
for (int i = 0; i < size; i = i + 1) {
      if (arr[i] > 10) {
            count++;
      }
}

echo("There are ");
echo(count);
echo(" numbers above the threshold and ");
echo(size - count);
echo(" numbers below or at the threshold.\n");
```

7.4) Test 4:

```
# Define a very inefficient function that checks if the number is
even #

func isEven(int n) {
      if (n == 0) {
            return 1;
      }
      if (n == 1) {
            return 0;
      }
      return isEven(n - 2);
}

echo("Enter a number: ");
int input = read();
echo("You entered ");
echo(input); echo(".\n");

# A complex arithmetic operation #
int operation = 2 ** (3 + 6 % 8 ** 2) - 5;
int t = 2 * 5 - (3 ** 6 % 6 - (6 | 4));
# program continues #

int even_func = isEven(input);
int even_mod = (input % 2) == 0;

if (even_func == even_mod) {
      if (even_func) {
            echo("Your number is even!\n");
      }
      else {
            echo("Your number is odd!\n");
      }
```

```
}
else {
      echo("FATAL ERROR!\n");
}
```

7.5) Test 5 (with syntax error):

```
# BUBBLE SORT ALGORITHM IMPLEMENTED IN DOT LANGUAGE #

func bubbleSort(int_arr arr, int n) {
      int i;
      int j;
      int flag = 1;

      for (i = 0; (i < n - 1) & flag; i = i + 1) {
            flag = 0;

            for (j = 0; j < n - i - 1; j = j + 1) {
                  if (arr[j] > arr[j+1]) {
                        # swap values #
                        int tmp = arr[j];
                        arr[j] = arr[j+1];
                        arr[j+1] = tmp;
                        flag = 1; # true #
                  }
            }
            # syntax error here, missing ‘}’ #
}

int SIZE = 5;
int_arr arr[5] = [10, 6, -2, 3, 5];
bubbleSort(arr, SIZE);
echo("Sorted Array: \n");
echo(arr);
echo("\n");
```