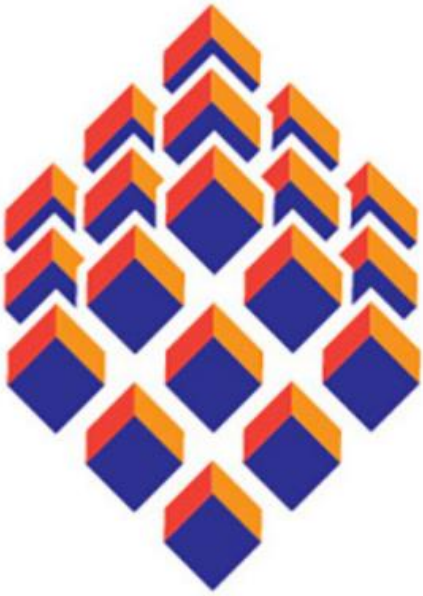
King Abdulaziz University
Faculty of Computing and Information Technology
Department of Computer Science
CPCS302 Project, Fall Semester 2023



Cubic³ Programming Language

Group 1

Member Name	Member ID
Randa Abdullah Nahhas	1906161
Shaima Abdullah Bashammakh	1914892
Khadija Salem Balfagih	1914895
Manar Abdulrahman Saleh	1914943

Instructor Name: Sultanah Alshammari

Table of Contents

List of Tables

Task Assignment

Member Name	Member ID	Responsibility
Randa Abdullah Nahhas	1906161	All works are done by all members
Shaima Abdullah Bashammakh	1914892	
Khadija Salem Balfagih	1914895	
Manar Abdulrahman Saleh	1914943	

1. Phase 1: Lexical Analyzer

1.1 Introduction

Cubic³ is a new Programming Language that is based in a certain BNF and designed using JavaCC, the main purpose of Cubic³ project to building new high-level language that conduct simple lines of code that most of Regular Expression contain only the first three letters of token name making it easy to use and easy to remember in creating projects in using many expressions such as Arithmetic Statements, Relational Statements, Conditional Statements, Logical Statements, Iterative Statements, Print Statements, Data Types, Data Structure, and more. Each statement ends with a Hash.

Small part from code as example of Cubic³ Programming Language:

```
CMT group Project CPCS302

INT $x #
$x ASS 100 #
IF {FLS AND TRU}
THN PRT:"false" END
ELS PRT:"true" END
END #

STR $numbers #
ARR $numbers ASS [ "One" | "Two" | "Three" ] #

LOP {10 LTE 20}
PRT:" COMPILER CONSTRUCTION "
END #

EXT
```

1.2 Tokens

Token Type	Token Name	Regular Expression
Arithmetic Unary Operations	Increment	PLSPLS
	Decrement	MISMIS
Arithmetic Binary Operations	Plus	PLS
	Minus	MIS
	DivIDe	DEV
	Reminder	REM
	Multiply	MUL
Assignment Operation	Assign	ASS
Relational Operations	Equality	EQU
	Inequality	NEQ
	GreaterThan	GRT
	GreaterThanOrEqual	GTE
	LessThan	LET

		LessThanOrEqual	LTE
Logical Operations		And	AND
		Or	OR
		Not	NOT
Alphabets		Digits	[0-9]
		Letters	[A-Z] | [a-z]
		Symbols	(|) | , | ; | # | @ | _ | : | | ?
Keywords		Print	PRT
		If	IF
		Else	ELS
		Then	THN
		Loop	LOP
		Exit	EXT
		End	END
		True	TRU
		False	FLS
Data Type	Integer	INT	(Digit)+
	Float	FLO	(Digit)+ Dot (Digit)+
	Boolean	BOL	TRU | FLS
	String	STR	DoubleQuotation (Letter | Digit | Symbols)* DoubleQuotation
IDentifiers		ID	$ (Letter)+(Letter | Digit)*
Constants		Constant	CON
Data Structure		Array	ARR
Punctuation Marks		DoubleQuotation	“
		Dot	.
		DollarSign	$
		Colon	:
		Separator	|
		LeftSquareBracket	[
		RightSquareBracket	]
		LeftCurlyBracket	{
		RightCurlyBracket	}
		Hash	#
White Spaces		Space	
		Tab	\t
		NewLine	\n
Comments		SingleLineComment	CMT (~"\n")+
		MultipleLineComment	CMS (< Letter >| < Digit >| < Symbol >|"\n")+ CMS

Table1: Tokens

1.3 Statements

Statement Type	Regular Expression	Example of Code
Variable Declaration Statement	IntDeclare ID Hash	`INT $X#`
	FloatDeclare ID Hash	`FLO $Y#`
	BooleanDeclare ID Hash	`BOL $IsRight#`
	StringDeclare ID Hash	`STR $name#`
	Constant DataType ID Hash	`CON INT $size#`

Assignment Statement	ID Assign (INT|FLO|BOL| STR) Hash	`$name ASS "Khadija"#`
		`$X ASS 7#`
		`$IsRight ASS Tru#`

Arithmetic Statement	(ID | INT | FLO) BinaryOperations (ID | INT | FLO) Hash	`$X PLS $Y#` `5 MIS 1.3#`
	(ID | INT) UnaryOperations Hash | UnaryOperations (ID | INT) Hash	`$Y ++#` `++ $X#`
Logical Statement	BOL LogicalOperations BOL Hash	`FLS OR TRU#`
Relational Statement	(ID | INT | FLO) RelationalOperations (ID | INT | FLO) Hash	`$X LET $Y#` `10 GTE 7#`
Print Statement	Print Colon STR Hash	`PRT: "Hi World"#`

Array	Array ID Assign LeftSquareBracket ((ID | INT | FLO | BOL | STR)(Separator (ID | INT | FLO | BOL | STR))*)* RightSquareBracket Hash	`ARR $array`
		`[5.1|7|2|4.6|"alaa"]#`
		`ARR $size [40|37|34]#`

Conditional Statement	If LeftCurlyBracket BooleanExpression RightCurlyBracket Then (Stmts)+ End End Hash	`IF {$X<2}`
		`THN PRT: "Hi World"`
		`END`
		`END #`

	If LeftCurlyBracket BooleanExpression RightCurlyBracket Then (Stmts)+ End Else (Stmts)* End End Hash	`IF {$X<2}`
		`THN PRT: "Hi World"`
		`END`
		`ELS PRT: "Welcome"`
		`END`
		`END #`

Iterative Statement	Loop LeftCurlyBracket BooleanExpression RightCurlyBracket (Stmts)+ End Hash	`LOP {$X>=3}`
		`PRT : "Hi World"`
		`END #`

Table2: Statements

2. Phase 2: Syntax Analysis

2.1. BNF Form

1. **Start** → Stmts Hash| Exit
2. **Stmts** → Declaration | Assignment | Arithmetic | Relational | logical | Array | Print | Conditional | Iterative
3. **Declaration** → Constant DataType ID
4. **DataType** → Int_declare | Float_declare | Boolean_declare | String_declare
5. **Assignment** → ID Assign (INT|FLO|BOL| STR)
6. **Arithmetic** → (ID | INT | FLO) BinaryOperations (ID | INT | FLO) l (ID | INT) UnaryOperations | UnaryOperations (ID | INT)
7. **Relational** → (ID | INT | FLO) RelationalOperations (ID | INT | FLO)
8. **Logical** → BOL LogicalOperations BOL
9. **Array** → Array ID Assign LeftSquareBracket ((ID | INT | FLO | BOL | STR)(Separator (ID | INT | FLO | BOL | STR))*)* RightSquareBracket
10. **Print** → Print Colon STR
11. **Conditional** → If LeftCurlyBracket BooleanExpression RightCurlyBracket Then (Stmts)+ End (Else (Stmts)+ End)? End
12. **Iterative** → Loop LeftCurlyBracket BooleanExpression RightCurlyBracket (Stmts)+ End
13. **RelationalOperations** → Equality l Inequality l GreaterThan l GreaterThanOrEqual l LessThan l LessThanOrEqual
14. **LogicalOperations** → And l Or l Not
15. **BinaryOperations** → Plus l Minus l Multiply l Divide l Reminder
16. **UnaryOperations** → Increment l Decrement
17. **BooleanExpression** → Logical | Relational | BOL

18. **ID** → DollarSign (Letter)+ (Letter | Digit)*
19. **Letters**→ [A-Z] | [a-z]
20. **Digit** → [0-9]
21. **INT**→ (Digit)+
22. **FLO**→ (Digit)+ Dot (Digit)+
23. **BOL**→ TRU | FLS
24. **STR**→ DoubleQuotation (Letter | Digit | Symbols)* DoubleQuotation

25. **If**→ IF
26. **Then**→ THN
27. **Else**→ ELS
28. **Loop**→ LOP
29. **Exit**→ EXT
30. **End**→ END
31. **Print**→ PRT
32. **StringDeclare**→ STR
33. **BooleanDeclare**→ BOL
34. **IntDeclare**→ INT
35. **FloatDeclare**→ FLO
36. **Constant**→ CON
37. **Array**→ ARR
38. **Increment** → PLSPLS
39. **Decrement** → MISMIS
40. **Plus** → PLS
41. **Minus**→ MIS
42. **Divide**→ DEV
43. **Multiply**→ MUL
44. **Reminder**→ REM
45. **Assign**→ ASS
46. **Equality**→ EQU
47. **Inequality**→ NEQ
48. **GreaterThan**→ GRT
49. **GreaterThanOrEqual**→ GTE
50. **LessThan**→ LET
51. **LessThanOrEqual**→ LTE
52. **And**→ AND
53. **Or**→ OR
54. **Not**→ NOT
55. **DoubleQuotation**→ “
56. **Dot**→.

57. **DollarSign**→ $
58. **Colon**→:
59. **Separator**→ |
60. **LeftSquareBracket**→ [
61. **RightSquareBracket**→]
62. **LeftCurlyBracket**→ {
63. **RightCurlyBracket**→ }
64. **Hash**→ #
65. **Symbols**→ (|) | , | ; | # | @ | _ | : | | ?

2.2. BNF Explanation

1. **Start** → Stmts Hash| EXIT

A statement begins with a statement type followed by a Hash, or the keyword “Exit” to quit

2. **Stmts** → Declaration | Assignment | Arithmetic | Relational | logical | Array | Print | Conditional | Iterative

A statement in Emerald has several types, it can be a declaration statement, an assignment statement, an Arithmetic statement, a logical statement, a Relational statement, a print statement, an array statement, a Conditional statement, or an Iterative statement.

3. **Declaration** → Constant DataType ID

A declaration statement can be started by a constant, followed by a datatype and an IDentifier

4. **DataType** → Int_declare | Float_declare | Boolean_declare | String_declare

A datatype can be a integer ID, float ID, Boolean ID, or string ID

5. **Assignment** → ID Assign (INT|FLO|BOL| STR)

As assignment begins with an IDentifier assignment operator, followed by integer, float, Boolean, or string between Double Quotation

6. **Arithmetic** → (ID | INT | FLO) BinaryOperations (ID | INT | FLO) l (ID | INT) UnaryOperations | UnaryOperations (ID | INT)

An arithmetic statement begins by IDentifier, integer or float followed by a binary operational symbol and a second ID, integer or float, or a unary operational symbol instead of binary operational symbol

7. **Relational** → (ID | INT | FLO) RelationalOperations (ID | INT | FLO)

A relational statement is two IDentifiers, integer or float with a relational operational symbol between them

8. **Logical** → BOL LogicalOperations BOL

A logical statement is two Booleans with a logical operational symbol between them

9. **Array** → Array ID Assign LeftSquareBracket ((ID | INT | FLO | BOL | STR)(Separator (ID | INT | FLO | BOL | STR))*)* RightSquareBracket

An array statement begins with the keyword ARR followed by an IDentifier and an assignment symbol then open Left Square Bracket and Right Square Bracket for close between them IDentifier, integer, float, Boolean or string between Double Quotation, separating them with separator symbol

10. **Print** → Print Colon STR

A print statement begins with the keyword PRT followed by two Double Quotation between them a string

11. **Conditional** → If LeftCurlyBracket BooleanExpression RightCurlyBracket Then (Stmts)+ End (Else (Stmts)+ End)? End

An conditional statement begins with the keyword IF followed by open Left Curly Bracket and close with Right Curly Bracket between them Boolean Expression followed by THN keyword and one or more then END keyword Or statement begins with the keyword IF followed by open Left Curly Bracket and close with Right Curly Bracket between them Boolean Expression followed by THN keyword and one or more then END keyword then ELS keyword and zero or more Statement then END keyword for else and another END keyword for if

12. **Iterative** → Loop LeftCurlyBracket BooleanExpression RightCurlyBracket (Stmts)+ End

An iterative statement begins with the keyword LOP followed by open Left Curly Bracket and close with Right Curly Bracket between them Boolean Expression followed by one or more Statement and END keyword

13. **RelationalOperations** → Equality l Inequality l GreaterThan l GreaterThanOrEqual l LessThan l LessThanOrEqual

An relational Operations can be Equality or Inequality or Greater Than or Greater Than Or Equal or Less Than or Less Than Or Equal.

14. **LogicalOperations**→ And l Or l Not

An logical Operations can be And or Or or Not

15. **BinaryOperations**→ Plus l Minus l Multiply l Divide l Reminder

An Arithmetic Binary Operations can be Plus or Minus or Divide or Reminder or Multiply

16. **UnaryOperations** → Increment l Decrement

An arithmetic Unary Operations can be Increment or Decrement

17. **BooleanExpression** → Logical | Relational | BOL

The Boolean Expression can be logical statement, relational statement or Boolean, we used on Conditional and Iterative statments

18. **ID**→ DollarSign (Letter)+ (Letter | Digit)*

A ID begin with dollar sign then one or more letter then zero or more Letter or Digit

19. **INT**→ (Digits)+

A INT can be one or more digits

20. **FLO**→ (Digits)+Dot (Digits)+

A FLO start and end with one or more digits between them dot

21. **BOL**→ TRU | FLS

A BOL can be True or False

22. **STR**→ DoubleQuotation (Letter | Digit | Symbols)* DoubleQuotation

A STR can be zero or more Letter or Digit or Symbols between double quotation

The following part consists of the keywords and symbols of Emerald's BNF:

1. **Letters**→ [A-Z] | [a-z]
2. **Digit** → [0-9]
3. **If**→ IF
4. **Then**→ THN
5. **Else**→ ELS
6. **Loop**→ LOP
7. **Exit**→ EXT
8. **End**→ END
9. **Print**→ PRT
10. **StringDeclare**→ STR
11. **BooleanDeclare**→ BOL
12. **IntDeclare**→ INT
13. **FloatDeclare**→ FLO
14. **Constant**→ CON
15. **Array**→ ARR
16. **Increment** → PLSPLS
17. **Decrement** → MISMIS
18. **Plus** → PLS
19. **Minus**→ MIS
20. **Divide**→ DEV
21. **Multiply**→ MUL
22. **Reminder**→ REM
23. **Assign**→ ASS
24. **Equality**→ EQU
25. **Inequality**→ NEQ
26. **GreaterThan**→ GRT
27. **GreaterThanOrEqual**→ GTE
28. **LessThan**→ LET
29. **LessThanOrEqual**→ LTE
30. **And**→ AND
31. **Or**→ OR

32. **Not**→ NOT
33. **DoubleQuotation**→ “
34. **Dot**→.
35. **DollarSign**→ $
36. **Colon**→:
37. **Separator**→ |
38. **LeftSquareBracket**→ [
39. **RightSquareBracket**→]
40. **LeftCurlyBracket**→ {
41. **RightCurlyBracket**→ }
42. **Hash**→ #
43. **Symbols**→ (|) | , | ; | # | @ | _ | : | | ?

3. Phase 3: Semantic Analysis

3.1. JJ Sample Output

The input file:

```
CMT group Project

INT $x #
INT $y #
$x ASS 55 #
$y ASS 100 #
IF {$x EQU $y} THN PRT:"Wrong" END END #

$x PLS 15 #
$y MISMIS #
BOL $z #
$z ASS TRU #
IF {FLS AND FLS} THN PRT:"TRUE" END ELS PRT:"FALSE" END END #

STR $name #
ARR $name ASS [ "Khadija" | "Shaimaa" | "Rnda" | "Manar" ] #

INT $count #
$count ASS 10 #
LOP {$count LTE 20} PRT:"statments" END #

EXT
```

The Output:

```
Reading from file...
comment statment
Integer data type declarative statment
Syntctically correct statments.

Integer data type declarative statment
Syntctically correct statments.

assignment statment
Syntctically correct statments.

assignment statment
Syntctically correct statments.

if statment
relational statment
then statment
print statment
end statment
end statment
Syntctically correct statments.

binary arithmetic statment
Syntctically correct statments.

unary arithmetic statment
Syntctically correct statments.

Boolean data type declarative statment
Syntctically correct statments.

assignment statment
Syntctically correct statments.


if statment
logical statment
then statment
print statment
end statment
else statment
print statment
end statment
end statment
Syntctically correct statments.

String data type declarative statment
Syntctically correct statments.

array statment
assignment statment
Syntctically correct statments.

Integer data type declarative statment
Syntctically correct statments.

assignment statment
Syntctically correct statments.

iterative statment
relational statment
print statment
end statment
Syntctically correct statments.

Exit statement
 Thank you for using our system
```

3.2. JJT Sample Output

The input file:

```
CMT group Project

INT $x #
INT $y #
$x ASS 55 #
$y ASS 100 #
IF {$x EQU $y} THN PRT:"Wrong" END END #

$x PLS 15 #
$y MISMIS #
BOL $z #
$z ASS TRU #
IF {FLS AND FLS} THN PRT:"TRUE" END ELS PRT:"FALSE" END END #

STR $name #
ARR $name ASS [ "Khadija" | "Shaimaa" | "Rnda" | "Manar" ] #

INT $count #
$count ASS 10 #
LOP {$count LTE 20} PRT:"statments" END #

EXT
```

The Output:

```
Reading from file...
comment statment
Integer data type declarative statment
 < Start
 <  Stmts
 <   Declaration
 <    DataType
 <     intDeclare:INT
 <    ID:$x
 <  Hash:#
Syntctically correct statments.

Integer data type declarative statment
 < Start
 <  Stmts
 <   Declaration
 <    DataType
 <     intDeclare:INT
 <    ID:$y
 <  Hash:#
Syntctically correct statments.
```

```
assignment statment
 < Start
 <  Stmts
 <   Assignment
 <    ID:$x
 <    assignOP:ASS
 <    integer:55
 <  Hash:#
Syntctically correct statments.

assignment statment
 < Start
 <  Stmts
 <   Assignment
 <    ID:$y
 <    assignOP:ASS
 <    integer:100
 <  Hash:#
Syntctically correct statments.

if statment
relational statment
then statment
print statment
end statment
end statment
 < Start
 <  Stmts
 <   Conditional
 <    iff:IF
 <    leftCurly:{
 <    BooleanEx
 <     Relational
 <      ID:$x
 <      RelationalOperations
 <       equalOP:EQU
 <      ID:$y
 <    rightCurly:}
 <    then:THN
 <    Stmts
 <     Print
 <      priint:PRT
 <      colon::
 <      string:"Wrong"
 <    end:END
 <    end:END
 <  Hash:#
Syntctically correct statments.
```

```
binary arithmetic statment
 < Start
 <  Stmts
 <   Arithmetic
 <    ID:$x
 <    BinaryOperations
 <     plusOP:PLS
 <    integer:15
 <  Hash:#
Syntctically correct statments.

unary arithmetic statment
 < Start
 <  Stmts
 <   Arithmetic
 <    ID:$y
 <    UnaryOperations
 <     decrementOP:MISMIS
 <  Hash:#
Syntctically correct statments.

Boolean data type declarative statment
 < Start
 <  Stmts
 <   Declaration
 <    DataType
 <     booleanDeclare:BOL
 <    ID:$z
 <  Hash:#
Syntctically correct statments.

assignment statment
 < Start
 <  Stmts
 <   Assignment
 <    ID:$z
 <    assignOP:ASS
 <    booleann:TRU
 <  Hash:#
Syntctically correct statments.
```

```
if statment
logical statment
then statment
print statment
end statment
else statment
print statment
end statment
end statment
 < Start
 <  Stmts
 <   Conditional
 <    iff:IF
 <    leftCurly:{
 <    BooleanEx
 <     Logical
 <      booleann:FLS
 <      LogicalOperations
 <       andOP:AND
 <      booleann:FLS
 <    rightCurly:}
 <    then:THN
 <    Stmts
 <     Print
 <      priint:PRT
 <      colon::
 <      string:"TRUE"
 <    end:END
 <    elsee:ELS
 <    Stmts
 <     Print
 <      priint:PRT
 <      colon::
 <      string:"FALSE"
 <    end:END
 <    end:END
 <  Hash:#
Syntctically correct statments.

String data type declarative statment
 < Start
 <  Stmts
 <   Declaration
 <    DataType
 <     stringDeclare:STR
 <    ID:$name
 <  Hash:#
Syntctically correct statments.
```

```
array statment
assignment statment
 < Start
 <  Stmts
 <   Array
 <    Array:ARR
 <    ID:$name
 <    assignOP:ASS
 <    leftSquare:[
 <    string:"Khadija"
 <    separator:|
 <    string:"Shaimaa"
 <    separator:|
 <    string:"Rnda"
 <    separator:|
 <    string:"Manar"
 <    rightSquare:]
 <  Hash:#
Syntctically correct statments.

Integer data type declarative statment
 < Start
 <  Stmts
 <   Declaration
 <    DataType
 <     intDeclare:INT
 <    ID:$count
 <  Hash:#
Syntctically correct statments.


assignment statment
 < Start
 <  Stmts
 <   Assignment
 <    ID:$count
 <    assignOP:ASS
 <    integer:10
 <  Hash:#
Syntctically correct statments.
```

```
iterative statment
relational statment
print statment
end statment
 < Start
 <  Stmts
 <   Iterative
 <    loop:LOP
 <    leftCurly:{
 <    BooleanEx
 <     Relational
 <      ID:$count
 <      RelationalOperations
 <       lessOrEqualOP:LTE
 <      integer:20
 <    rightCurly:}
 <    Stmts
 <     Print
 <      priint:PRT
 <      colon::
 <      string:"statments"
 <    end:END
 <  Hash:#
Syntctically correct statments.

Exit statement
 Thank you for using our system
```

4. Appendix

4.1. JJ Grammar

```
/**
 * Group 1
 * Khadija Salem - Shaimaa Abdullah - Randa Nahhas - Manar Saleh
 */
options
{
  static = true;
}

PARSER_BEGIN(MyNewGrammar)
package Project;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class MyNewGrammar
{
  public static void main(String args []) throws ParseException
  {
    try
    {
      MyNewGrammar parser = new MyNewGrammar(new FileInputStream("testSample.txt"));
    }
    catch (FileNotFoundException e)
```

```
    {
      System.out.println("There is no available file");
      System.out.println(e.getMessage());
    }
    System.out.println("Reading from file...");
    while (true)
    {
      try
      {
        MyNewGrammar.Start();
        System.out.println("Syntctically correct statments.\n");
      }
      catch (Exception e)
      {
        System.out.println("NOK.");
        System.out.println(e.getMessage());
        MyNewGrammar.ReInit(System.in);
      }
      catch (Error e)
      {
        System.out.println("Oops.");
        System.out.println(e.getMessage());
        break;
      }
    }
  }
}

PARSER_END(MyNewGrammar)

SKIP :
{
  < Space : " " >
| < NewLine :
    "\r"
  | "\n" >
| < Tab : "\t" >
}

SPECIAL_TOKEN :
{
  < Single_Line_Comment : "CMT" (~["\n"])+ | "CMS" (< Letter >| < Digit >| < Symbol
>|"\n")+ "CMS" > { System.out.println("comment statment");}
}

TOKEN : /* Keywords */
{
  < If : "IF" >
  {
    System.out.println("if statment");
  }
| < Then : "THN" >
  {
    System.out.println("then statment");
  }
| < Else : "ELS" >
```

```
  {
    System.out.println("else statment");
  }
| < Loop : "LOP" >
  {
    System.out.println("iterative statment");
  }
| < Exit : "EXT" >
  {
    System.out.println("Exit statement \n Thank you for using our system");
    System.exit(0);
  }
| < End : "END" >
  {
    System.out.println("end statment");
  }
| < Print : "PRT" >
  {
    System.out.println("print statment");
  }
  /*Data Types*/
| < StringDeclare : "STR" >
  {
    System.out.println("String data type declarative statment");
  }
| < BooleanDeclare : "BOL" >
  {
    System.out.println("Boolean data type declarative statment");
  }
| < IntDeclare : "INT" >
  {
    System.out.println("Integer data type declarative statment");
  }
| < FloatDeclare : "FLO" >
  {
    System.out.println("float data type declarative statment");
  }
| < Constant : "CON" >
  {
    System.out.println("constant variable");
  }
| < Array : "ARR" >
  {
    System.out.println("array statment");
  }
}

TOKEN : /* OPERATORS */
{
  // UNARY OPERATORS
  < Increment : "PLSPLS" >
  {
    System.out.println("unary arithmetic statment");
  }
| < Decrement : "MISMIS" >
  {
```

```
    System.out.println("unary arithmetic statment");
  }

  // BINARY OPERATORS
| < Plus : "PLS" >
  {
    System.out.println("binary arithmetic statment");
  }
| < Minus : "MIS" >
  {
    System.out.println("binary arithmetic statment");
  }
| < Multiply : "MUL" >
  {
    System.out.println("binary arithmetic statment");
  }
| < Divide : "DIV" >
  {
    System.out.println("binary arithmetic statment");
  }
| < Reminder : "REM" >
  {
    System.out.println("binary arithmetic statment");
  }
| < Assign : "ASS" >
  {
    System.out.println("assignment statment");
  }
  // RELATIONAL OPERATORS
| < Equality : "EQU" >
  {
    System.out.println("relational statment");
  }
| < Inequality : "NEQ" >
  {
    System.out.println("relational statment");
  }
| < GreaterThan : "GRT" >
  {
    System.out.println("relational statment");
  }
| < GreaterThanOrEqual : "GTE" >
  {
    System.out.println("relational statment");
  }
| < LessThan : "LET" >
  {
    System.out.println("relational statment");
  }
| < LessThanOrEqual : "LTE" >
  {
    System.out.println("relational statment");
  }
  // LOGICAL OPERATORS
| < And : "AND" >
  {
```

```
    System.out.println("logical statment");
  }
| < Or : "OR" >
  {
    System.out.println("logical statment");
  }
| < Not : "NOT" >
  {
    System.out.println("logical statment");
  }
}

TOKEN : // PUNCTUATION MARKS
{
  < DoubleQuotation :     "\"" >
| < Dot :    "." >
| < DollarSign :    "$" >
| < Colon :  ":" >
| < Separator : "|" >
| < LeftSquareBracket : "[" >
| < RightSquareBracket :  "]" >
| < LeftCurlyBracket : "{" >
| < RightCurlyBracket : "}" >
| < Hash : "#" >
}

TOKEN : // IDENTIFIER
{
  < ID :
    < DollarSign > (< Letter >)+
    (
      < Letter >
    | < Digit >
    )* >
}

TOKEN : // ALPHABETS
{
  < Letter: [ "a"-"z" ] | [ "A"-"Z" ] >
| < Digit : [ "0"-"9" ] >
| < Symbol :
    "("
  | ")"
  | ","
  | ";"
  | "#"
  | "$"
  | "@"
  | "_"
  | ":"
  | "?"
  | " " >
}

TOKEN : // DATA TYPES
{
```

```
  < INT : (< Digit >)+ >
| < FLO : (< Digit >)+ < Dot > (< Digit >)+ >
| < BOL :
    (
      "TRU"
    | "FLS"
    ) >
| < STR :
    < DoubleQuotation >
    (
      < Letter >
    | < Digit >
    | < Symbol >
    )+
    < DoubleQuotation > >
}

void Start() :
{}
{
  Stmts() < Hash >
| < Exit >
}

void Stmts() :
{}
{
  LOOKAHEAD(2)
  Declaration()
| LOOKAHEAD(2)
  Assignment()
| LOOKAHEAD(2)
  Arithmetic()
| LOOKAHEAD(3)
  Relational()
| Logical()
| Array()
| Print()
| Conditional()
| Iterative()
}

void Declaration() :
{}
{
  (< Constant >)? DataType() (< ID >)
}

void DataType() :
{}
{
  < IntDeclare >
| < FloatDeclare >
| < BooleanDeclare >
| < StringDeclare >
}
```

```
void Assignment() :
{}
{
  < ID > < Assign >
  (
    < INT >
  | < FLO >
  | < BOL >
  | < STR >
  )
}

void Arithmetic() :
{}
{
  (
    LOOKAHEAD(2)
    (
      < ID >
    | < INT >
    | < FLO >
    )
    BinaryOperations()
    (
      < ID >
    | < INT >
    | < FLO >
    )
  |
    (
      < ID >
    | < INT >
    )
    UnaryOperations()
  | UnaryOperations()
    (
      < ID >
    | < INT >
    )
  )
}

void Relational() :
{}
{
  (
    < ID >
  | < INT >
  | < FLO >
  )
  RelationalOperations()
  (
    < ID >
  | < INT >
  | < FLO >
```

```
  )
}

void Logical() :
{}
{
  (
    < BOL >
  )
  LogicalOperations()
  (
    < BOL >
  )
}

void Array() :
{}
{
  < Array > < ID > < Assign > < LeftSquareBracket >
  (
    (
      < ID >
    | < INT >
    | < FLO >
    | < BOL >
    | < STR >
    )
    (
      < Separator >
      (
        < ID >
      | < INT >
      | < FLO >
      | < BOL >
      | < STR >
      )
    )*
  )*
  < RightSquareBracket >
}

void Print() :
{}
{
  < Print > < Colon > < STR >
}

void Conditional() :
{}
{
  < If > < LeftCurlyBracket  > BooleanExpression() < RightCurlyBracket > < Then >
  (
    Stmts()
  )+
  < End >
  (
```

```
    < Else >
    (
      Stmts()
    )+
    < End >
  )?
  < End >
}

void Iterative() :
{}
{
  < Loop > < LeftCurlyBracket > BooleanExpression() < RightCurlyBracket >
  (
    Stmts()
  )+
  < End >
}


void RelationalOperations() :
{}
{
  < Equality >
| < Inequality >
| < GreaterThan >
| < GreaterThanOrEqual >
| < LessThan >
| < LessThanOrEqual >
}

void LogicalOperations() :
{}
{
  < And >
| < Or >
| < Not >
}

void BinaryOperations() :
{}
{
  < Plus >
| < Minus >
| < Multiply >
| < Divide >
| < Reminder >
}

void UnaryOperations() :
{}
{
  < Increment >
| < Decrement >
}
```

```
void BooleanExpression() :
{}
{
  LOOKAHEAD(2)
  Logical()
|
  Relational()
|
  < BOL >
}
```

4.2. JJT Grammar

```
/**
 * Group 1
 * Khadija Salem - Shaimaa Abdullah - Randa Nahhas - Manar Saleh
 */
options
{
  static = true;
}

PARSER_BEGIN(MyNewGrammar)
package Project;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class MyNewGrammar
{
  public static void main(String args []) throws ParseException
  {
    try
    {
      MyNewGrammar parser = new MyNewGrammar(new FileInputStream("testSample.txt"));
    }
    catch (FileNotFoundException e)
    {
      System.out.println("There is no available file");
      System.out.println(e.getMessage());
    }
    System.out.println("Reading from file...");
    while (true)
    {
      try
      {
        SimpleNode x = MyNewGrammar.Start();
        x.dump(" < ");

       System.out.println("Syntctically correct statments.\n");
      }
      catch (Exception e)
      {
        System.out.println("NOK.");
        System.out.println(e.getMessage());
        MyNewGrammar.ReInit(System.in);
```

```
      }
      catch (Error e)
      {
        System.out.println("Oops.");
        System.out.println(e.getMessage());
        break;
      }
    }
  }
}

PARSER_END(MyNewGrammar)

SKIP :
{
  < Space : " " >
| < NewLine :
    "\r"
  | "\n" >
| < Tab : "\t" >
}

SPECIAL_TOKEN :
{
  < Single_Line_Comment : "CMT" (~["\n"])+ | "CMS" (< Letter >| < Digit >| < Symbol
>|"\n")+ "CMS" > { System.out.println("comment statment");}
}

TOKEN : /* Keywords */
{
  < If : "IF" >
  {
    System.out.println("if statment");
  }
| < Then : "THN" >
  {
    System.out.println("then statment");
  }
| < Else : "ELS" >
  {
    System.out.println("else statment");
  }
| < Loop : "LOP" >
  {
    System.out.println("iterative statment");
  }
| < Exit : "EXT" >
  {
    System.out.println("Exit statement \n Thank you for using our system");
    System.exit(0);
  }
| < End : "END" >
  {
    System.out.println("end statment");
  }
| < Print : "PRT" >
```

```
  {
    System.out.println("print statment");
  }
  /*Data Types*/
| < StringDeclare : "STR" >
  {
    System.out.println("String data type declarative statment");
  }
| < BooleanDeclare : "BOL" >
  {
    System.out.println("Boolean data type declarative statment");
  }
| < IntDeclare : "INT" >
  {
    System.out.println("Integer data type declarative statment");
  }
| < FloatDeclare : "FLO" >
  {
    System.out.println("float data type declarative statment");
  }
| < Constant : "CON" >
  {
    System.out.println("constant variable");
  }
| < Array : "ARR" >
  {
    System.out.println("array statment");
  }
}

TOKEN : /* OPERATORS */
{
  // UNARY OPERATORS
  < Increment : "PLSPLS" >
  {
    System.out.println("unary arithmetic statment");
  }
| < Decrement : "MISMIS" >
  {
    System.out.println("unary arithmetic statment");
  }

  // BINARY OPERATORS
| < Plus : "PLS" >
  {
    System.out.println("binary arithmetic statment");
  }
| < Minus : "MIS" >
  {
    System.out.println("binary arithmetic statment");
  }
| < Multiply : "MUL" >
  {
    System.out.println("binary arithmetic statment");
  }
| < Divide : "DIV" >
```

```
  {
    System.out.println("binary arithmetic statment");
  }
| < Reminder : "REM" >
  {
    System.out.println("binary arithmetic statment");
  }
| < Assign : "ASS" >
  {
    System.out.println("assignment statment");
  }
  // RELATIONAL OPERATORS
| < Equality : "EQU" >
  {
    System.out.println("relational statment");
  }
| < Inequality : "NEQ" >
  {
    System.out.println("relational statment");
  }
| < GreaterThan : "GRT" >
  {
    System.out.println("relational statment");
  }
| < GreaterThanOrEqual : "GTE" >
  {
    System.out.println("relational statment");
  }
| < LessThan : "LET" >
  {
    System.out.println("relational statment");
  }
| < LessThanOrEqual : "LTE" >
  {
    System.out.println("relational statment");
  }
  // LOGICAL OPERATORS
| < And : "AND" >
  {
    System.out.println("logical statment");
  }
| < Or : "OR" >
  {
    System.out.println("logical statment");
  }
| < Not : "NOT" >
  {
    System.out.println("logical statment");
  }
}
TOKEN : // PUNCTUATION MARKS
{
  < DoubleQuotation :   "\"" >
| < Dot :   "." >
| < DollarSign :   "$" >
| < Colon :   ":" >
```

```
| < Separator : "|" >
| < LeftSquareBracket : "[" >
| < RightSquareBracket :   "]" >
| < LeftCurlyBracket : "{" >
| < RightCurlyBracket : "}" >
| < Hash : "#" >
}

TOKEN : // IDENTIFIER
{
  < ID :
    < DollarSign > (< Letter >)+
    (
      < Letter >
    | < Digit >
    )* >
}

TOKEN : // ALPHABETS
{
  < Letter: [ "a"-"z" ] | [ "A"-"Z" ] >
| < Digit : [ "0"-"9" ] >
| < Symbol :
    "("
  | ")"
  | ","
  | ";"
  | "#"
  | "$"
  | "@"
  | "_"
  | ":"
  | "?"
  | " " >
}

TOKEN : // DATA TYPES
{
  < INT : (< Digit >)+ >
| < FLO : (< Digit >)+ < Dot > (< Digit >)+ >
| < BOL :
    (
      "TRU"
    | "FLS"
    ) >
| < STR :
    < DoubleQuotation >
    (
      < Letter >
    | < Digit >
    | < Symbol >
    )+
    < DoubleQuotation > >
}
```

```
SimpleNode Start() :
{}
{
  Stmts() Hash()
   {
     return jjtThis;
   }
| Exit()
   {
     return jjtThis;
   }
}

void Hash () : { Token token; }
{
  token = < Hash > { jjtThis.jjtSetValue(token.image); }
}

void Exit () : { Token token; }
{
  token = < Exit > { jjtThis.jjtSetValue(token.image); }
}


void Stmts() :
{}
{
  LOOKAHEAD(2)
  Declaration()
| LOOKAHEAD(2)
  Assignment()
| LOOKAHEAD(2)
  Arithmetic()
| LOOKAHEAD(3)
  Relational()
| Logical()
| Array()
| Print()
| Conditional()
| Iterative()
}


void Declaration() :
{}
{
  (Constant())? DataType() (ID())
}

void Constant () : { Token token; }
{
  token = < Constant > { jjtThis.jjtSetValue(token.image); }
}
```

```
void ID () : { Token token; }
{
  token = < ID > { jjtThis.jjtSetValue(token.image); }
}



void DataType() :
{}
{
  intDeclare ()
| floatDeclare ()
| booleanDeclare ()
| stringDeclare ()
}

void intDeclare () : { Token token; }
{
  token = < IntDeclare > { jjtThis.jjtSetValue(token.image); }
}

void floatDeclare () : { Token token; }
{
  token = < FloatDeclare > { jjtThis.jjtSetValue(token.image); }
}

void booleanDeclare () : { Token token; }
{
  token = < BooleanDeclare > { jjtThis.jjtSetValue(token.image); }
}

void stringDeclare () : { Token token; }
{
  token = < StringDeclare > { jjtThis.jjtSetValue(token.image); }
}



void Assignment() :
{}
{
  ID() assignOP()
  (
    integer ()
  | floatt ()
  | booleann ()
  | string ()
  )
}

void assignOP () : {Token token; }
{
  token = < Assign > { jjtThis.jjtSetValue(token.image); }
}
void integer () : {Token token; }
{
```

```
  token = < INT > { jjtThis.jjtSetValue(token.image); }
}
void floatt () : {Token token; }
{
  token = < FLO > { jjtThis.jjtSetValue(token.image); }
}
void booleann () : {Token token; }
{
  token = < BOL > { jjtThis.jjtSetValue(token.image); }
}
void string () : {Token token; }
{
  token = < STR > { jjtThis.jjtSetValue(token.image); }
}



void Arithmetic() :
{}
{
  (
    LOOKAHEAD(2)
    (
      ID()
    | integer()
    | floatt()
    )
    BinaryOperations()
    (
      ID()
     | integer()
     | floatt()
    )
  |
    (
      ID()
     | integer()

    )
    UnaryOperations()
  | UnaryOperations()
    (
      ID()
    | integer()

    )
  )
}

void Relational() :
{}
{
  (
    ID()
  | integer()
  | floatt()
```

```
  )
  RelationalOperations()
  (
    ID()
  | integer()
  | floatt()
  )
}

void Logical() :
{}
{
  (
    booleann()
  )
  LogicalOperations()
  (
    booleann()
  )
}

void Array() :
{}
{
  array() ID() assignOP() leftSquare()
  (
    (
      ID()
    | integer()
    | floatt()
    | booleann()
    | string()
    )
    (
      separator()
      (
        ID()
    | integer()
    | floatt()
    | booleann()
    | string()
      )
    )*
  )*
  rightSquare()
}


void array () : { Token token; }
{
  token = < Array > { jjtThis.jjtSetValue(token.image); }
}
void separator () : { Token token; }
{
  token = < Separator > { jjtThis.jjtSetValue(token.image); }
}
```

```
void rightSquare () : { Token token; }
{
  token = < RightSquareBracket > { jjtThis.jjtSetValue(token.image); }
}
void leftSquare () : { Token token; }
{
  token = < LeftSquareBracket > { jjtThis.jjtSetValue(token.image); }
}

void Print() :
{}
{
  priint () colon() string()
}

void priint () : { Token token; }
{
  token = < Print > { jjtThis.jjtSetValue(token.image); }
}

void colon () : { Token token; }
{
  token = < Colon > { jjtThis.jjtSetValue(token.image); }
}


void Conditional() :
{}
{
  iff () leftCurly () BooleanEx() rightCurly () then ()
  (
    Stmts()
  )+
  end ()
  (
     elsee ()
    (
      Stmts()
    )+
    end ()
  )?
  end ()
}

void leftCurly () : { Token token; }
{
  token = < LeftCurlyBracket > { jjtThis.jjtSetValue(token.image); }
}
void rightCurly () : { Token token; }
{
  token = < RightCurlyBracket > { jjtThis.jjtSetValue(token.image); }
}
void iff () : { Token token; }
{
  token = < If > { jjtThis.jjtSetValue(token.image); }
}
```

```
void end () : { Token token; }
{
  token = < End > { jjtThis.jjtSetValue(token.image); }
}
void then () : { Token token; }
{
  token = < Then > { jjtThis.jjtSetValue(token.image); }
}
void elsee () : { Token token; }
{
  token = < Else > { jjtThis.jjtSetValue(token.image); }
}


void Iterative() :
{}
{
  loop () leftCurly () BooleanEx() rightCurly ()
  (
    Stmts()
  )+
  end ()
}

void loop () : { Token token; }
{
  token = < Loop > { jjtThis.jjtSetValue(token.image); }
}

void RelationalOperations() :
{}
{
  equalOP ()
| notEqualOP ()
| greaterOP ()
| greaterOrEqualOP ()
| lessOP ()
| lessOrEqualOP ()
}

void equalOP () : { Token token; }
{
  token = < Equality > { jjtThis.jjtSetValue(token.image); }
}

void notEqualOP () : { Token token; }
{
  token = < Inequality > { jjtThis.jjtSetValue(token.image); }
}

void lessOP () : { Token token; }
{
  token = < LessThan > { jjtThis.jjtSetValue(token.image); }
}

void lessOrEqualOP () : { Token token; }
```

```
{
  token = < LessThanOrEqual > { jjtThis.jjtSetValue(token.image); }
}

void greaterOP () : { Token token; }
{
  token = < GreaterThan > { jjtThis.jjtSetValue(token.image); }
}

void greaterOrEqualOP () : { Token token; }
{
  token = < GreaterThanOrEqual > { jjtThis.jjtSetValue(token.image); }
}



void LogicalOperations() :
{}
{
  andOP ()
| orOP ()
| notOP ()
}

void andOP () : { Token token; }
{
  token = < And > { jjtThis.jjtSetValue(token.image); }
}

void orOP () : { Token token; }
{
  token = < Or > { jjtThis.jjtSetValue(token.image); }
}

void notOP () : { Token token; }
{
  token = < Not > { jjtThis.jjtSetValue(token.image); }
}


void BinaryOperations() :
{}
{
  plusOP ()
| minusOP ()
| multiplyOP ()
| devideOP ()
| reminderOP ()
}

void plusOP () : { Token token; }
{
  token = < Plus > { jjtThis.jjtSetValue(token.image); }
}
void minusOP () : { Token token; }
{
```

```
  token = < Minus > { jjtThis.jjtSetValue(token.image); }
}
void multiplyOP () : { Token token; }
{
  token = < Multiply > { jjtThis.jjtSetValue(token.image); }
}
void devideOP () : { Token token; }
{
  token = < Divide > { jjtThis.jjtSetValue(token.image); }
}
void reminderOP () : { Token token; }
{
  token = < Reminder > { jjtThis.jjtSetValue(token.image); }
}

void UnaryOperations() :
{}
{
  incrementOP ()
| decrementOP ()
}

void incrementOP () : { Token token; }
{
  token = < Increment > { jjtThis.jjtSetValue(token.image); }
}

void decrementOP () : { Token token; }
{
  token = < Decrement > { jjtThis.jjtSetValue(token.image); }
}

void BooleanEx() :
{}
{
  LOOKAHEAD(2)
  Logical()
|
  Relational()
|
  booleann()
}
```