King Abdulaziz University, Jeddah, Saudi Arabia.
Faculty of Computing and Information Technology
Department of Computer Science
CPCS302-Compiler Construction

Team Project

Name	ID
Logain Sendi	2005341
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]

Table of contents

1. Teamwork and Responsibilities

Team Members	Responsibilities
Logain Sendi	**All work was done by all group members**
[redacted]	
[redacted]	
[redacted]	
[redacted]	

2. Phase 1: Lexical Analysis

2.1. Introduction

In Beta language, we are using BNF rules that are implemented using JavaCC. Its main purpose is to provide a high-level language that allows users to perform arithmetic operations, relation operations, Boolean statements, etc. A variety of different types of data structures are also contained within it.

An example of a source code:

```
*********Arithmetic statement *********
                    1 ß* 5.
*********Assignment statement ********
                    Y  ß= 5.
********* Conditional statements ********
    Ifß [ßY == 5 ]
          then  ß "True" ß.
   otherwise
          ß "False" ß.
***end***
```

A compiler for beta language was created. The procedure started with lexical analysis, which required breaking down a string of letters into a string of lexical tokens before sending the string to syntax analysis in the second phase. Then the parser examines the syntactical structure to determine whether or not the input conforms to the syntax required by programming language. Later on, the parse tree will then be generated. lastly, the syntax tree and symbol table are in the semantic analysis to check if the provided program is semantically compatible with the language definition.

2.2. Tokens

Token Type	Token Name	Regular Expression
Arithmetic_Operations	addition	“ß+”
	subtraction	“ß-“
	mult	“ß*“
	division	“ß/“
	assign	“ß=“
Relational_Operations	less	“ß<“
	lessOrEqual	“ß<=“
	greater	“ß>“
	greatOrEqual	“ß>=“
	equal	“ß==“
	notEqual	“ß!=“
Logical_operators	and	“ß&“
	or	“ß|“
	not	“ß!“
letters	letter	[“a”-“z”] | [“A”-“Z”]
Punctuation_Marks	dblquotation	” \” ”
	lftBraceSqu	“[“
	ritBraceSqu	“]”
	period	“.”
	comma	“,”
whitespace	whitespace	“ “
	newLine	“\n” | “\r”
	tab	"\t"
Identifiers	identifiers	"ß" (Letters) (Letters |integer|float)+
Keywords	if	“ifß”
	Else	“elseß”
	then	“thenß”
	exit	“exitß”
	beta	“ß”
	arrayKey	“ARRAYß”
	listKey	“LISTß”
	iterationKey	“iterateß”
comment	comment	“ßß”(Letters|Digits|Punctuation_Marks)+ “ßß”
Data type	integerKey	INTß
	FloatKey	FLOATß
	constantKey	CONSTß
	letterKey	letterß

	BooleanKey	BOOLß
Digits	#Digits	[“0”-“9”]
	integer	(Digits)+
	Float	(Digits)*(period)(Digits)+
Print_Statement	printKey	(“print”) (“ß”)(“)(Letters|Digits| Punctuation_Marks) (“)(“ß”)

2.3. Language Statements

Statement Type	**Code**
Arithmetic statement	1 ß* 5.
Relational statement	2 ß< 10.
Logical statement	4 ß& 1.
Boolean statements	BOOLß ßEx6 .
Conditional statements:	If [ßEx7 ß== 5] print ß "True" ß. If [ßEx8 ß!= 35] print ß "match" ß. else print ß "mismatch" ß.
Iterative statement	Iterate [ßEx9 ß!= 0] print ß "hello" ß.
Variable Declaration	ßEx10 ß= 93.
Constant variable declaration	CONSTß ßEx11 ß= 3.14.
Data type declaration	FLOATß ßEx11.
list	LISTß ßEx12 ß= [logain,1,2.5].
array	ARRAYß ßEx13 ß= [logain,wajd,deem,reem,reena].

3. Phase 2: Syntax Analysis

3.1. BNF Grammar

Start → Stmts.| Comment

Stmts → stringArr | List | iteration | Assignment | ifStmt | printStmt | arithmeticStmt | logicalStmt | Stmt| realtionStmt | declartionStmt

Comment → ßß (|Digits| Letter|PunctuationMarks)+ ßß

PunctuationMarks → " | [|] | . | ,

Assignment → identifier ß= stmt

Stmt → (Identifier | Letter | Digit) (arithmeticStmt| LogicalStmt| RelationalStmt | logicalStmt)?

stringArr → Array identifier ß= [(letter)+ (comma(letter))*]

List → LIST identifier ß= [(letter | digit)+ (comma(letter | digit))*]

ifStmt → If [condition] (stmts)+ . else(stmts)*.

Iteration → Iterate [condition](stmt)*.

Condition → (Identifier | digit) (RelationalOp | LogicalOp) (Identifier |Digits)

declareStmt → DataType identifier

DataType → intergerKey|floatKey|constantKey|letterKey|BooleanKey

relationalStmt → (identifier| digit) RelationalOP (identifier| digit)

arithmeticStmt → (identifier| digit) ArithmaticOP (identifier| digit)

LogicalStmt → (identifier| digit) LogicalOP (identifier| digit)

Identifier → ß (Letters) (Letters |Digits)+

Printstmt → print ß dblquotation (Letters|Digits| PunctuationMarks) dblquotation ß

Digit → integer | float

Integer → ["0"-"9"]+

Float → ["0"-"9"]+ . ["0"-"9"]+

Letter → (["A"-"Z" , "a"-"z"])+

RelationalOp → ß< | ß<= | ß> | ß>= | ß== | ß!=

LogicalOp → ß& | ß| | ß!

ArithmeticOp → ß+ | ß- | ß* | ß/ | ß=

3.2. BNF Grammar with Comments

Start → Stmts. | Comment

//We can start by writing a statement that ends with a dot, or a comment.

Stmts → stringArr | List | iteration | Assignment | ifStmt | printStmt | arithmeticStmt | logicalStmt | Stmt| realtionStmt | declartionStmt

//We have 10 types of statements in our language.

Comment → ßß (Letters|Digits|Punctuation_Marks)+ ßß

//Comments start with double ß signs and end with them too. We can write any letter, digit, or a punctuation mark in it.

PunctuationMarks → " | [|] | . | ,

//We have 6 Punctuation Marks in our language.

Assignment → identifier ß= stmt.

//Assign values to an identifier

Stmt → (Identifier | Letter | Digit) (arithmeticStmt| LogicalStmt| RelationalStmt | logicalStmt)?

//A statement can be either an identifier, letter, or digit. Also, you can add arithmetic, logical, and relational statements.

StringArr → ARRAY identifier ß= [(letter)+ (comma(letter))*]

//A string array should have at least 1 letter.

List → LIST identifier ß= [(letter | digit)+ (comma(letter | digit))*]

//All elements in a list can have different data types.

ifStmt → If [condition] (stmts)+ . else(stmts)*.

//we should start with "if" keyword followed by "[ ]" and a condition in between. A condition can be followed by 1 or more statements. An else statement is optional

Iteration → Iterate [condition](stmt)*.

//we should start with “iterate” keyword followed by “[ ]” and a condition in between. A condition can be followed by 0 or more statements.

Condition → (Identifier | digit) (RelationalOp | LogicalOp) (Identifier |Digits)

//Conditions start with either a digit or an identifier followed by relational or logical operationsand ends with a digit or an identifier.

Constant → CONST (IntegerDeclare | FractionDeclare | LettersDeclare).

//We should start constant declaration with “CONST” followed by the desired datatype, then “ß=”, ending with letters or digits.

declareStmt → DataType identifier

//We should start letter declaration with a “data type “, followed by an “identifier.

DataType → intergerKey|floatKey|constantKey|letterKey|BooleanKey

//The data type can be either integer or float or constant or letter

RelationalStmt → (identifier| digit) RelationalOP (identifier| digit)

//A relational statement starts with a digit or an identifier, followed by a relational operation, and ends with a digit or an identifier. (An identifier should be compared with an identifier, a digit should be compared with a digit)

ArithmeticStmt → (identifier| digit) ArithmaticOP (identifier| digit)

//An Arithmetic statement starts with a digit or an identifier, followed by an Arithmetic operation, and ends with a digit or an identifier. (An identifier should be compared with an identifier, a digit should be compared with a digit)

LogicalStmt → (identifier| digit) LogicalOP(identifier| digit)

//A Logical statement starts with a digit or an identifier, followed by a Logical operation, and ends with a digit or an identifier. (An identifier should be compared with an identifier, a digit should be compared with a digit)

Identifier→ ß (Letters) (Letters |Digits)+

//An identifier starts with ß followed by a letter, then followed by a combination of letters or digits.

Printstmt → print ß "(Letters|Digits| PunctuationMarks) " ß

//A print statement starts with the print keyword, followed by ß" and any combination of letters, digits, or punctuation marks, ending with "ß.

Digit → integer | float

//A digit can be an integer or a float.

Integer → ["0"-"9"]+

Float → ["0"-"9"]+ . ["0"-"9"]+

Letter → (["A"-"Z" , "a"-"z"])+

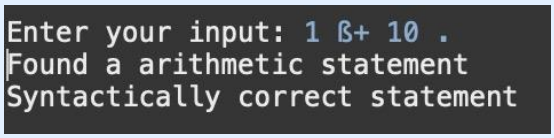
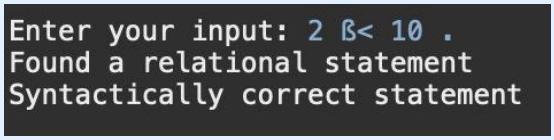
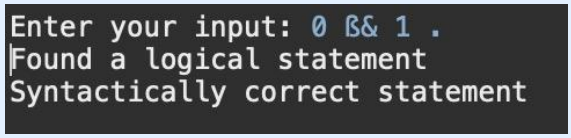
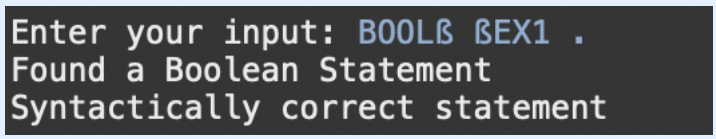
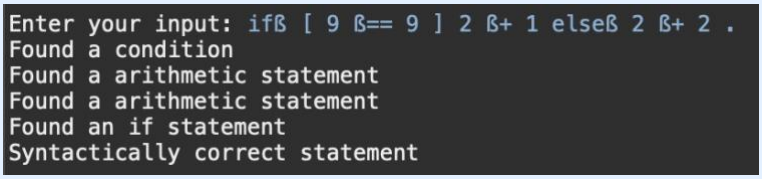
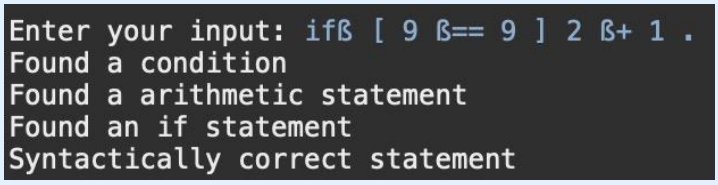
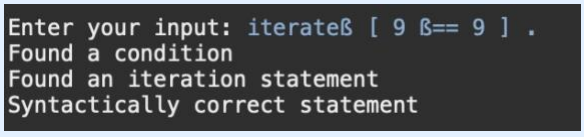
RelationalOp → ß< | ß<= | ß> | ß>= | ß== | ß!=

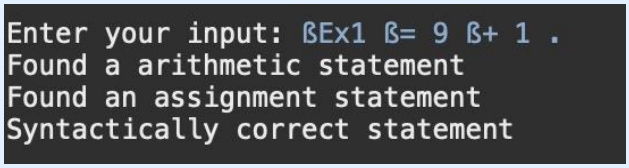
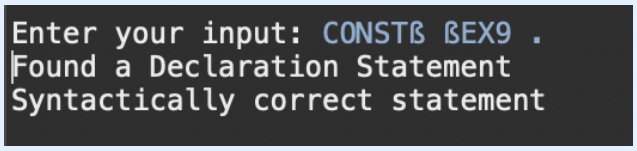
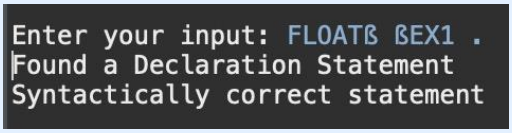
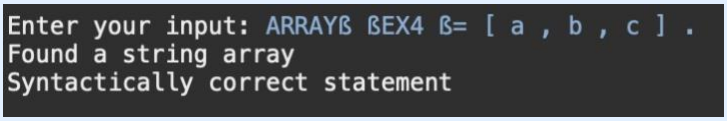
LogicalOp → ß& | ß| | ß!

ArithmeticOp → ß+ | ß- | ß* | ß/ | ß=

4. Screenshots of Output

4.1. JJ File Screenshots

Statement Type	Code
Arithmetic statement	
Relational statement	
Logical statement	
Boolean statements	
Conditional statements:	 
Iterative statement	

Variable Declaration	
Constant variable declaration	
Data type declaration	
list	
array	

4.2. Lexical or Syntactic Actions

4.2.1. Lexical Actions

```
TOKEN : /* Arithmetic Operators  */
{
  < addition: "ß+" >     {System.out.println("add is an arithmetic operation ");}
| < subtraction : "ß-" > {System.out.println("subtract is an arithmetic operation ");}
| < mult : "ß*" >        {System.out.println("multiply is an arithmetic operation ");}
| < division: "ß/" >     {System.out.println("divide is an arithmetic operation ");}
| < assign: "ß=" >       {System.out.println("assign is an arithmetic operation ");}
}
```

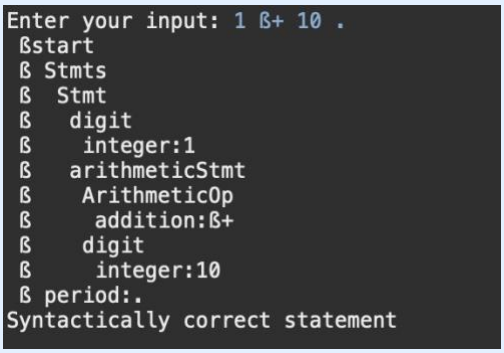
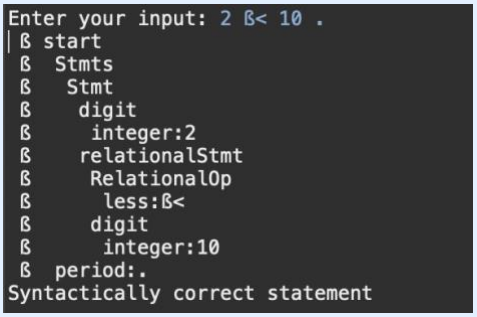
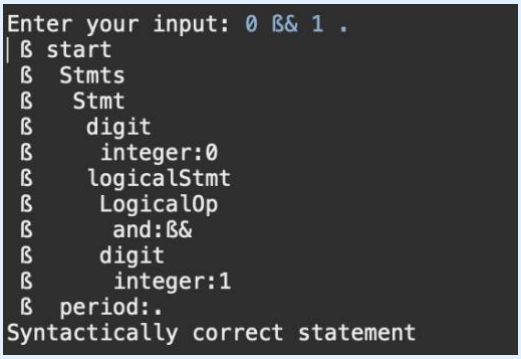
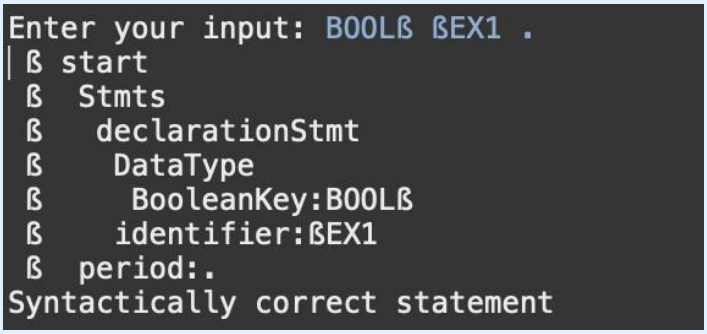
4.2.2. Syntactic Actions

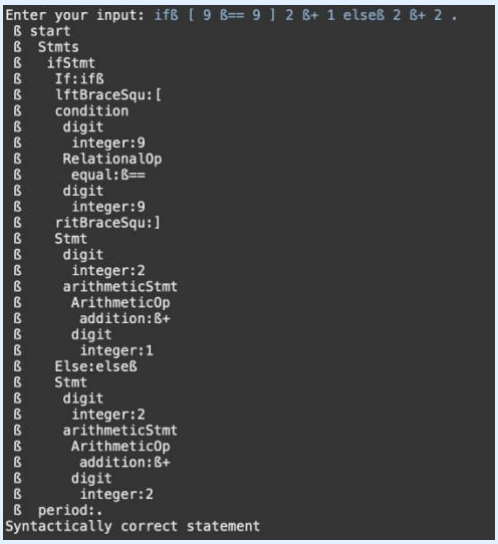
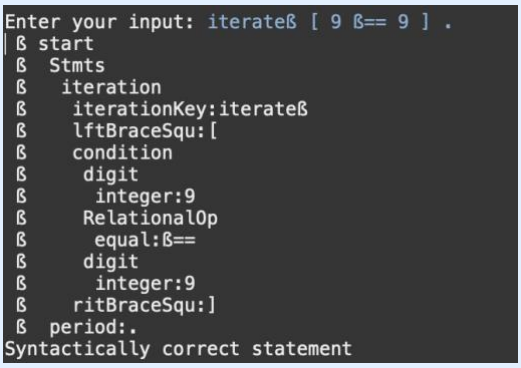
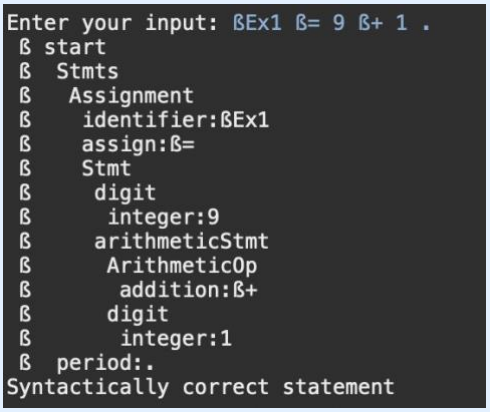
```
5.  void stringArr() :{ }
6.  {
7.  <  arrayKey > <identifier >  <assign > <lftBraceSqu > (<letter >)+  (< comma > <letter >)* <
    ritBraceSqu >
8.        {
9.       System.out.println("Found a string array");
10.    }
11. }
12.
13.
14. void List() :{ }
15. {
16.
17.   < listKey ><identifier >  <assign > <lftBraceSqu > (digit() |  < letter >)+ ((< comma >) (digit() |  <
    letter >))* < ritBraceSqu >
18.         {
19.      System.out.println("Found a list");
```

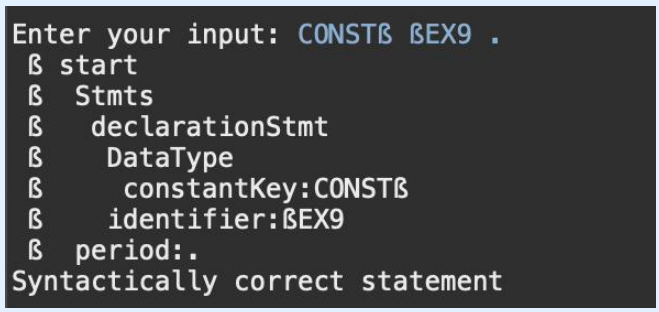
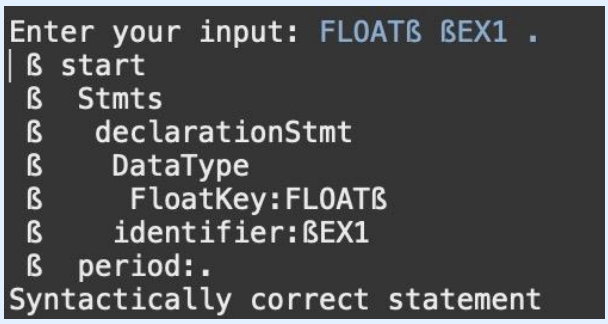
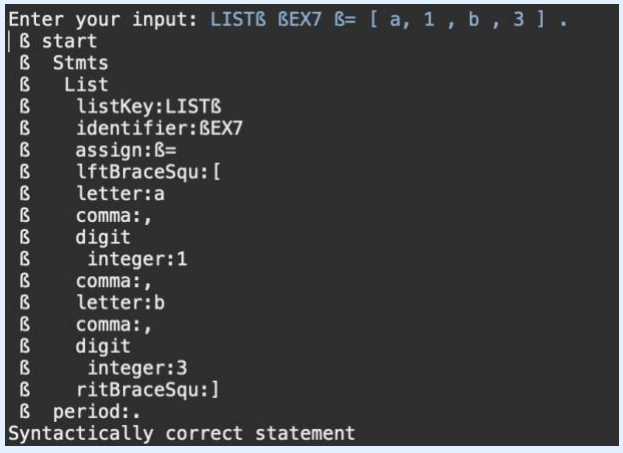
```
  }
}


void ifStmt() :{ }
{
  < If > <lftBraceSqu > condition() <ritBraceSqu >  (Stmt())+ (< Else > Stmt())*
       {
    System.out.println("Found an if statement");
  }
}

void iteration() :{ }
{

  < iterationKey > <lftBraceSqu > condition() < ritBraceSqu > (Stmt()< period >)*
       {
    System.out.println("Found an iteration statement");
  }
}


void condition() :{ }
{

  (    < identifier > | digit() )(RelationalOp() | LogicalOp()) ( < identifier > | digit())
       {
    System.out.println("Found a condition");
  }
}
```

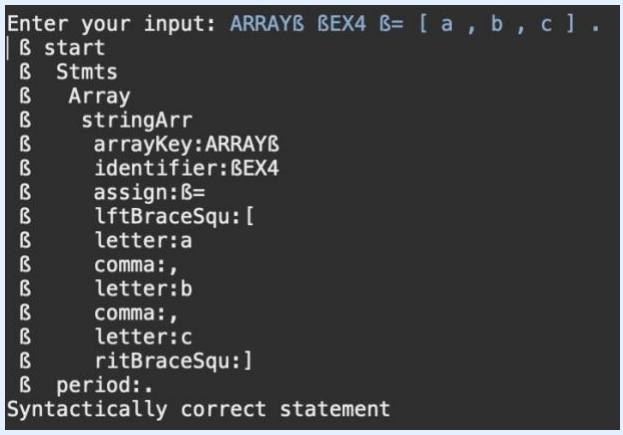
4.3. JJT File Screenshots

Statement Type	Code

Arithmetic statement	
Relational statement	
Logical statement	
Boolean statements	

Conditional statements:	
Iterative statement	
Variable Declaration	

Constant variable declaration	
Data type declaration	
list	

array	 Enter your input: ARRAYß ßEX4 ß= [a , b , c] . ß start ß Stmts ß Array ß stringArr ß arrayKey:ARRAYß ß identifier:ßEX4 ß assign:ß= ß lftBraceSqu:[ß letter:a ß comma:, ß letter:b ß comma:, ß letter:c ß ritBraceSqu:] ß period:. Syntactically correct statement

5. Appendix A: JJ Grammar

```
options
{
  static = true;
}

PARSER_BEGIN(MyNewGrammar)
package CPCS302Project;

public class MyNewGrammar
{
public static void main(String args []) throws ParseException
{
MyNewGrammar parser = new MyNewGrammar(System.in);
System.out.println("***** Welcome to Beta programming language *****");
while (true){
System.out.print("\nEnter your input: ");
try
{
MyNewGrammar.start();
System.out.println("Syntactically correct statement");
}
catch (Exception e)
{
System.out.println("Syntactically NOT correct statement");
System.out.println(e.getMessage());
break;
}
}
}
}

PARSER_END(MyNewGrammar)
//ß
SKIP :
{
  < whitespace: " " >
| < newLine1: "\r" >
| < tab: "\t" >
| < newLine2: "\n" >
}

//////////////////////////////////////

TOKEN : /* Arithmetic Operators  */
{
  < addition: "ß+" >
| < subtraction : "ß-" >
| < mult : "ß*" >
| < division: "ß/" >
| < assign: "ß=" >
}
```

```
////////////////////////////////////
TOKEN : /* Logical Operators */
{
  < and: "ß&" >
| < or : "ß|" >
| < not : "ß!" >
}
////////////////////////////////////

TOKEN : /* Relational Operators */
{
  < less: "ß<" >
| < lessOrEqual : "ß<=" >
| < greater : "ß>" >
| < greatOrEqual: "ß >=" >
| < equal : "ß==" >
| < notEqual : "ß!=" >
}
////////////////////////////////////

TOKEN : /* letter */
{
  < letter: (["a"-"z","A"-"Z"])+>
}

////////////////////////////////////

TOKEN : /* Punctuation Marks */
{
   < dblquotation: " \" " >
   | < lftBraceSqu : "[" >
   | < ritBraceSqu: "]" >
   | < period: "." >
   | < comma: "," >

}

////////////////////////////////////

TOKEN : /* Identifiers */
{
   < identifier:  (< beta >)(< letter >) (<letter >|< integer >|< Float>)+ >
}
////////////////////////////////////

TOKEN : /* Keywords */
{
   < If: "ifß" >
|  < Else: "elseß" >
|  < then: "thenß" >
|  < exit: "exitß" >
|  < arrayKey: "ARRAYß" >
|  < listKey: "LISTß" >
|  < iterationKey: "iterateß" >
|   < printKey: "printß" >
```

```
| < beta: "ß" >
}
////////////////////////////////

TOKEN : /* Data Type */
{
   < integerKey: "INTß" >
|  < FloatKey: "FLOATß" >
|  < constantKey: "CONSTß" >
|  < letterKey: "letterß" >
| < BooleanKey: "BOOLß ">

}

////////////////////////////////
TOKEN : /* digits */
{
    < #Digits: ["0"-"9"] >
|  < integer:  (< Digits >)+  >
|  < Float: (< Digits >)*(< period >)(< Digits >)+  >
}
////////////////////////////////

 void start(): {  }
{
   Stmts() < period >|   Comment()
// statements end with period.
}

void Stmts(): { }
{
  ///  statements in our program are://////
  stringArr()
| List()
| iteration()
| LOOKAHEAD(3)Assignment()
| ifStmt()
| printStmt()
| arithmeticStmt()
| logicalStmt()
| LOOKAHEAD(3)Stmt()
| relationalStmt()
| declarationStmt()
}

/* comment start and end with double " ß " , the comment contains
 combinations of digit or letter or Punctuation Marks. */
void Comment():{ }
{
   < beta >< beta > ((digit()|< letter > | PunctuationMarks() )+)  < beta >< beta
>
   {
     System.out.println("Found a Comment");
   }
```

```
}

// to assign value to an identifier ,write the identifier first then " ß= " after
that write a statement.
void Assignment():{ }
{
  < identifier > <assign > Stmt()
   {
     System.out.println("Found an assignment statement");
   }
}
/* statement can be either an identifier or letter or digit then it can be
followed
 by arithmetic, relational, or logical statement.  */
void Stmt():{ }
{
  < identifier >| < letter >| digit() (arithmeticStmt()|
relationalStmt()|logicalStmt())?
}

/* string array start with the keyword " ARRAYß" then an identifier followed by
"ß=",
combinations of letters separated by comma between the Bracket"[]".  */
void stringArr() :{ }
{
<  arrayKey > <identifier >  <assign > <lftBraceSqu > (<letter >)+  (< comma >
<letter >)* < ritBraceSqu >
       {
     System.out.println("Found a string array");
   }
}

/* list start with the keyword "LISTß" then an identifier followed by "ß=",then
combinations of letters and digits separated by comma between the Bracket"[]".
*/
void List() :{ }
{

 < listKey ><identifier >  <assign > <lftBraceSqu > (digit() |  < letter >)+ ((<
comma >) (digit() |  < letter >))* < ritBraceSqu >
        {
     System.out.println("Found a list");
   }
}

/* if statement start with the keyword "ifß" then write the conditions between
the Bracket"[]"
 followed by statement ends with period. if the conditions is not satisfied you
can write
 "elseß " then a statement.ends with period  */
void ifStmt() :{ }
{
  < If > <lftBraceSqu > condition() <ritBraceSqu >  (Stmt())+ (< Else > Stmt())*
        {
     System.out.println("Found an if statement");
```

```
    }
}
/*iteration statement start with the keyword "iterateß" then write the conditions
between the Bracket"[],
 followed by a statement ends with period.    */
void iteration() :{ }
{
  < iterationKey > <lftBraceSqu > condition() < ritBraceSqu > (Stmt()< period >)*
        {
    System.out.println("Found an iteration statement");
    }
}

/* condition starts with either an identifier , digit, relational operations,
or logical operations. ends with an identifier or digit  */
void condition() :{ }
{

  (    < identifier > | digit() )(RelationalOp() | LogicalOp()) ( < identifier >
| digit())
        {
    System.out.println("Found a condition");
    }
}


// to declare a Statement write the data type then an identifier.
void declarationStmt() : { }
{
   DataType() (< identifier >)
{

   System.out.println("Found a Declaration Statement");
 }
}

// data type can be either an integer, float, constant, or letter.
void DataType () : { }
{
< integerKey > |  < FloatKey > | < constantKey >| < letterKey > |< BooleanKey >

 }


// relational statement start with relational operations then identifier or
digit//
void relationalStmt(): { } {
 RelationalOp() (< identifier >| digit())
           {
    System.out.println("Found a relational statement");
    }
}
// logical statement start with logical operations then identifier or digit//
void logicalStmt(): { } {
```

```
 LogicalOp() (< identifier >| digit())
              {
     System.out.println("Found a logical statement");
   }
}
// arithmetic statement start with arithmetic operations then identifier or
digit//
void arithmeticStmt(): { } {
 ArithmeticOp() (< identifier >| digit())
              {
     System.out.println("Found a arithmetic statement");
   }
}
/* print statement start with keyword "printß" then "ß" ,
then combinations of letters,digits or PunctuationMarks between the double
quotations followed by "ß" */
void printStmt(): { } {
< printKey >  <beta > <dblquotation > (< letter >|digit()| PunctuationMarks()) <
dblquotation >  <beta >
            {
     System.out.println("Found a print statement");
   }
}


void RelationalOp(): { }
{
< less>
| < lessOrEqual >
| < greater >
| < greatOrEqual >
| < equal >
| < notEqual >
}

void LogicalOp(): { } {
< and >
| < or >
| < not >
}

void ArithmeticOp(): { }
{
< addition >
| < subtraction >
| < mult >
|<  division >
|<  assign >
}

void digit(): { } {
< Float > | < integer >
}
```

```
void PunctuationMarks(): { } {
 < lftBraceSqu >
| < ritBraceSqu >
| < period >
| < comma >
| < dblquotation >
}
```

6. Appendix B: JJT Grammar

```
options
{
  static = true;
}

PARSER_BEGIN(MyNewGrammar)
package CPCS302Project2;

public class MyNewGrammar
{
public static void main(String args []) throws ParseException
{
MyNewGrammar parser = new MyNewGrammar(System.in);
System.out.println("***** Welcome to Beta programming language *****");
while (true){
System.out.print("\nEnter your input: ");
try
{
SimpleNode n = MyNewGrammar.start();
n.dump(" ß ");
System.out.println("Syntactically correct statement");
}
catch (Exception e)
{
System.out.println("Syntactically NOT correct statement");
System.out.println(e.getMessage());
break;
}
}
}
}

PARSER_END(MyNewGrammar)
//ß
SKIP :
{
  < whitespace: " " >
| < newLine1: "\r" >
| < tab: "\t" >
| < newLine2: "\n" >
}

//////////////////////////////////////

TOKEN : /* Arithmetic Operators (Binary)  */
{
  < addition: "ß+">
| < subtraction : "ß-" >
| < mult : "ß*" >
```

```
| < division: "ß/" >
| < assign: "ß=" >
}
////////////////////////////////////
TOKEN : /* Logical Operators */
{
  < and: "ß&" >
| < or : "ß|" >
| < not : "ß!" >
}
////////////////////////////////////

TOKEN : /* Relational Operators */
{
  < less: "ß<" >
| < lessOrEqual : "ß<=" >
| < greater : "ß>" >
| < greatOrEqual: "ß >=" >
| < equal : "ß==" >
| < notEqual : "ß!=" >
}
////////////////////////////////////

TOKEN : /* letter */
{
  < letter: (["a"-"z","A"-"Z"])+>
}

////////////////////////////////////

TOKEN : /* Punctuation Marks */
{
      < dblquotation: "\"" >
      | < lftBraceSqu : "[" >
      | < ritBraceSqu: "]" >
      | < period: "." >
      | < comma: "," >

}

////////////////////////////////////

TOKEN : /* Identifiers */
{
      < identifier:  (< beta >) < letter > (<letter >|< Digits >)+ >
}


TOKEN : /* Keywords */
{
      < If: "ifß" >
|     < Else: "elseß" >
|     < then: "thenß" >
|     < exit: "exitß" >
|     < arrayKey: "ARRAYß" >
```

```
|      < listKey: "LISTß" >
|      < iterationKey: "iterateß" >
|    < printKey: "printß" >
|  < beta: "ß" >
}

TOKEN : /* Data Type */
{
       < integerKey: "INTß" >
|      < FloatKey: "FLOATß" >
|      < constantKey: "CONSTß" >
|      < letterKey: "letterß" >
|    < BooleanKey: "BOOLß ">

}


////////////////////////////////
TOKEN : /* digits */
{
    < #Digits: ["0"-"9"] >
|      < integer:  (< Digits >)+  >
|      < Float: (< Digits >)*(< period >)(< Digits >)+  >
}
////////////////////////////////
SimpleNode start(): { Token t; }
{
  Stmts() period()
  {
    return jjtThis;
  }
|  Comment()
{
  return jjtThis;
}
}

void Stmts(): { Token t; }
{
  stringArr()
| List()
| iteration()
| LOOKAHEAD(3)Assignment()
| ifStmt()
| printStmt()
| arithmeticStmt()
| logicalStmt()
| LOOKAHEAD(3)Stmt()
| relationalStmt()
| declarationStmt()
}

/// functions
void Comment():{ Token t; }
```

```
{
  beta()beta() ((digit()|letter()|PunctuationMarks())+)  beta()beta()
}


void Assignment():{ }
{
  (identifier()) (assign()) Stmt()
}

void Stmt():{ }
{
  ((identifier())| letter()| digit()) (arithmeticStmt()|
relationalStmt()|logicalStmt())?
}


void stringArr() :{ }
{
 arrayKey() (identifier()) (assign()) (lftBraceSqu()) (letter())+
(comma()(letter()))* (ritBraceSqu())
}


void List() :{ }
{

  listKey()(identifier())  (assign()) (lftBraceSqu()) (digit() | letter())+
((comma())(digit() | letter()))* (ritBraceSqu())
}


void ifStmt() :{ }
{
  If() lftBraceSqu() condition() ritBraceSqu()  (Stmt())+ (Else() Stmt())*
}

void iteration() :{ }
{

  iterationKey() lftBraceSqu() condition() ritBraceSqu() (Stmt()period())*
}


void condition() :{ }
{

  (identifier() | digit()) (RelationalOp() | LogicalOp()) (identifier() |digit())
}


void declarationStmt() : { }
{
   DataType() (identifier())
```

```
}


void DataType () : { }
{
 integerKey() |  FloatKey() | constantKey() | letterKey() | BooleanKey()

 }



void relationalStmt(): { } {
 RelationalOp() (identifier()| digit())
}

void logicalStmt(): { } {
 LogicalOp() (identifier()| digit())
}

void arithmeticStmt(): { } {
 ArithmeticOp() (identifier()| digit())
}

void printStmt(): { } {
printKey() beta() dblquotation() (letter()|digit()| PunctuationMarks())
dblquotation()  beta()
}


void RelationalOp(): { }
{
less()
| lessOrEqual()
| greater()
| greatOrEqual()
| equal()
| notEqual()
}

void LogicalOp(): { } {
and()
| or ()
| not()
}

void ArithmeticOp(): { }
{
addition()
| subtraction()
| mult()
| division()
| assign()
}
```

```
void digit(): { } {
Float() | integer()
}


void PunctuationMarks(): { } {
 lftBraceSqu()
| ritBraceSqu()
| period()
| comma()
| dblquotation()
}

///keywords/////
void iterationKey() :{ Token t;}
{
  t=< iterationKey > { jjtThis.jjtSetValue(t.image); }

}

void then() :{Token t; }
{
  t=< then > { jjtThis.jjtSetValue(t.image); }

}

void exit() :{Token t; }
{
  t=< exit > { jjtThis.jjtSetValue(t.image); }

}
void If():{ Token t; }
{
t=< If > { jjtThis.jjtSetValue(t.image); }
}

void Else():{ Token t; }
{
t=< Else > { jjtThis.jjtSetValue(t.image); }
}

void beta() :{ Token t; }
{
t=< beta > { jjtThis.jjtSetValue(t.image); }
}

void arrayKey() :{Token t; }
{
  t=< arrayKey > { jjtThis.jjtSetValue(t.image); }

}

void listKey() :{ Token t;}
{
```

```
  t=< listKey > { jjtThis.jjtSetValue(t.image); }
}

void printKey() :{ Token t; }
{
t=< printKey > { jjtThis.jjtSetValue(t.image); }
}

//////////DataType///////////

void integerKey() :{Token t; }
{
  t=< integerKey > { jjtThis.jjtSetValue(t.image); }

}

void FloatKey() :{Token t; }
{
  t=< FloatKey > { jjtThis.jjtSetValue(t.image); }

}

void constantKey() :{ Token t;}
{
  t=< constantKey > { jjtThis.jjtSetValue(t.image); }

}

void letterKey() :{ Token t; }
{
t=< letterKey > { jjtThis.jjtSetValue(t.image); }
}

////////Punctuation Marks///////////

void dblquotation():{ Token t; }
{
t=< dblquotation > { jjtThis.jjtSetValue(t.image); }
}

void lftBraceSqu() :{ Token t; }
{
t=< lftBraceSqu > { jjtThis.jjtSetValue(t.image); }
}

void ritBraceSqu() :{ Token t; }
{
t=< ritBraceSqu > { jjtThis.jjtSetValue(t.image); }
}

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

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

/////digits//////

void integer() :{ Token t; }
{
t=< integer > { jjtThis.jjtSetValue(t.image); }
}

void Float() :{ Token t; }
{
t=< Float > { jjtThis.jjtSetValue(t.image); }

}

/////boolean////////
void BooleanKey() :{ Token t; }

{

t=< BooleanKey > { jjtThis.jjtSetValue(t.image); }

}

/////identifier////
void identifier() :{ Token t; }
{
t=< identifier > { jjtThis.jjtSetValue(t.image); }
}

/////////Arithmetic Operators//////
void addition() :{ Token t; }
{
t=< addition > { jjtThis.jjtSetValue(t.image); }
}

void subtraction() :{ Token t; }
{
t=< subtraction > { jjtThis.jjtSetValue(t.image); }
}

void mult() :{ Token t; }
{
t=< mult > { jjtThis.jjtSetValue(t.image); }
}

void division() :{ Token t; }
{
t=< division > { jjtThis.jjtSetValue(t.image); }
}

void assign() :{ Token t; }
```

```
{
t=< assign > { jjtThis.jjtSetValue(t.image); }
}

////Logical Operators//
void and() :{ Token t; }
{
t=< and > { jjtThis.jjtSetValue(t.image); }
}

void or() :{ Token t; }
{
t=< or > { jjtThis.jjtSetValue(t.image); }
}

void not() :{ Token t; }
{
t=< not > { jjtThis.jjtSetValue(t.image); }
}

////Relational Operators///
void less() :{ Token t; }
{
t=< less > { jjtThis.jjtSetValue(t.image); }
}

void lessOrEqual() :{ Token t; }
{
t=< lessOrEqual > { jjtThis.jjtSetValue(t.image); }
}

void greater() :{ Token t; }
{
t=< greater > { jjtThis.jjtSetValue(t.image); }
}

void greatOrEqual() :{ Token t; }
{
t=< greatOrEqual > { jjtThis.jjtSetValue(t.image); }
}

void equal() :{ Token t; }
{
t=< equal > { jjtThis.jjtSetValue(t.image); }
}

void notEqual() :{ Token t; }
{
t=< notEqual > { jjtThis.jjtSetValue(t.image); }
}
////Alphabet///////
void letter() :{ Token t; }
{
t=< letter > { jjtThis.jjtSetValue(t.image); }
}
```