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

CPCS 302

COMPILER CONSTRUCTION

NARS Programming Language

Student name	ID	Section
Noor [redacted]	[redacted]	B9
Nsreen Hujjatullah Asadullah	[redacted]	B9
Asma[redacted]	[redacted]	B9
Ren[redacted]	[redacted]	B9
Sara[redacted]	[redacted]	B9

Table of Contents

Introduction:

"NARS" programming language is a programming language designed using JavaCC, the main purpose of NARS is to conduct simple lines of code for various operations (e.g. arithmetic operations, relational operations, if statements, etc.).

Why did we call our language Nars? Nars is a compilation of the first letter of our names:

N: for Nsreen & Noor

A: for Asma

R: for Renad

S: for Sara

1 Phase 1: Lexical Analysis

1.1. Tokens:

Token Type	Token Name	Regular Expression
Keywords	IF | THEN | ELSE | INT_TYPE | BOOL_TYPE | FLOAT_TYPE | STR_TYPE | LOOP | NARSOUT | RETURN |LIST | ARRAY | TRUE | FALSE | ENDIF | ENDLOOP | EXIT	"if" | "then" | "else" | "INT" | "BOOL" |"FLOAT" | "STR" | "loop" | "NARSout" | "return" |"list" | "array" | "true" | "false" | "endif" | "endloop" | "exit"
Data Types	NUM_VAL	(DIGIT) + (("."(DIGIT)+)?)
	BOOLEAN_VAL	"true" | "false"
	STRING_VAL	(LETTER| DIGIT| SYMBOL)+
Identifiers	IDENTIFIER	(TILDE_SYMBOL) (LETTER | DIGIT)+
	CONSTANT	"final"
Alphabets	DIGIT	["0" – "9"]
	LETTER	["A" – "Z"] | ["a" – "z"]
	SYMBOL	"#"| "$" | "?" | "_"
Comments	SINGLE_LINE_COMMENT	"@"(~["\n"])*"@"
Spaces	WHIT_SPACE	" "

	TAB	"\t"
	NEW_LINE	“\n”
	NEW_LINE2	"\r"
Binary Arithmetic Operators	ADD	"+"
	SUB	"-"
	MULTIPLY	"*"
	DIVIDE	"/"
	POWER	"^^"
	REMINDER	"%"
Unary Arithmetic Operators	DECREMENT	"--"
	INCREMENT	"++"
Assignment operator	ASSIGNMENT	"="
Logical Operators	AND	"&&"
	OR	"||"
	NOT	"!"
Relational Operators	GREATER_THAN	".>"
	LESS_THAN	".<"
	LESS_THAN_OR_EQUAL	"<="
	GREATER THAN_OR_ EQUAL	">="
	IS_EQUAL	"=="
	NOT_EQUAL	"!="
Punctuation Marks	SEMICOLON	";"
	COLON	":"
	LEFT_BRACKETS	"("
	RIGHT_BRACKETS	")"
	LEFT_ANG_BRACKETS	"<"
	RIGHT_ANG_BRACKETS	">"
	DOUBLE_QUTATION	"""
	SEPARATOR	"|"
	COMMA	","
	TILDE_SYMBOL	"~"

1.2. Statements:

Language Statement	Regular Expression	Example(code)
Logical Statements	LogicalOperand LogicalOP LogicalOperand COMMA	true && false,
Arithmetic Statements	operand binaryOP operand COMMA	~sum + ~total,
		5 * ~sum,
	unaryOP operand COMMA	--~sumN,
		++5,
Comparison Statements	operand relationalOP operand COMMA	9 >= 4,
		~flag1 .< ~flag2,
		~X != ~Y,
Conditional statements	IF LEFT_ANG_BRACKETS conditionalExpression RIGHT_ANG_BRACKETS THEN (Stmts)+ ENDIF COMMA	if < ~flag1 .< ~flag2, > then ~sum= ~flag2, endif,
	IF LEFT_ANG_BRACKETS conditionalExpression RIGHT_ANG_BRACKETS THEN (Stmts)+ ELSE (Stmts)+ ENDIF COMMA	if < ~flag1 .< ~flag2,> then ~sum= ~flag2, else ~sum= ~flag1, endif,
Iterative Statements	LOOP LEFT_ANG_BRACKETS conditionalExpression RIGHT_ANG_BRACKETS (Stmts)+ ENDLOOP COMMA	loop < ~Flag .< 5, > ++~Flag , endloop,
Declaration Statements	INT_TYPE IDENTIFIER COMMA	INT ~dal,
	FLOAT_TYPE IDENTIFIER COMMA	FLOAT ~Cost,
	BOOL_TYPE IDENTIFIER COMMA	BOOL ~Flag,
	STR_TYPE IDENTIFIER COMMA	STR ~name,
Print Statements	NARSOUT LEFT_BRACKETS DOUBLE_QUTATION	NARSout ("Welcome To NARS Language "),

	(STRING_VAL)+ DOUBLE_QUTATION RIGHT_BRACKETS COMMA	
CONSTANT Declaration Statements	CONSTANT DataType IDENTIFIER COMMA	final FLOAT ~pi,
Assignment Statement	IDENTIFIER ASSIGNMENT (STRING_VAL| BOOLEAN_VAL| NUM_VAL | IDENTIFIER) COMMA	~COST = 15,
List Statements	LIST IDENTIFIER ASSIGNMENT LEFT_BRACKETS ((NUM_VAL) SEMICOLON)+ RIGHT_BRACKETS COMMA	list ~C = (2;4;8;) ,
Array statements	ARRAY LEFT_ANG_BRACKETS (INT_TYPE | BOOL_TYPE | FLOAT_TYPE| STR_TYPE) RIGHT_ANG_BRACKETS IDENTIFIER ASSIGNMENT LEFT_BRACKETS ((NUM_VAL | STRING_VAL| BOOLEAN_VAL) SEMICOLON) + RIGHT_BRACKETS COMMA	array <STR> ~s = (Noor; Nesreen; Asma; Renad;Sara;),
Comment	AT_SIGN (~["\n"])* AT_SIGN	@--Declare Variable--@

2 Phase 2: Syntactic Analysis:

2.1. BNF Form:

1. Start → Stmts | Exit
2. Stmts → declerationStmt | arithmeticStmt | assignmentStmt | relationalStmt | logicalStmt | iterativeStmt | conditionalStmt | listStmt | arrayStmt | printStmt
3. declerationStmt → (CONSTANT)? DataType IDENTIFIER COMMA
4. assignmentStmt → IDENTIFIER ASSIGNMENT (STRING_VAL | BOOLEAN_VAL | NUM_VAL | IDENTIFIER) COMMA
5. printStmt → NARSOUT LEFT_BRACKETS DOUBLE_QUTATION (STRING_VAL)+ DOUBLE_QUTATION RIGHT_BRACKETS COMMA
6. listStmt → LIST IDENTIFIER ASSIGNMENT LEFT_BRACKETS ((NUM_VAL) SEMICOLON)+ RIGHT_BRACKETS COMMA
7. arrayStmt → ARRAY LEFT_ANG_BRACKETS (INT_TYPE | BOOL_TYPE | FLOAT_TYPE| STR_TYPE) RIGHT_ANG_BRACKETS IDENTIFIER ASSIGNMENT LEFT_BRACKETS ((NUM_VAL | STRING_VAL| BOOLEAN_VAL) SEMICOLON)+ RIGHT_BRACKETS COMMA
8. iterativeStmt → LOOP LEFT_ANG_BRACKETS conditionalExpression RIGHT_ANG_BRACKETS (Stmts)+ ENDLOOP COMMA
9. conditionalStmt → IF LEFT_ANG_BRACKETS conditionalExpression RIGHT_ANG_BRACKETS THEN (Stmts)+ (ELSE (Stmts)+)? ENDIF COMMA
10. conditionalExpression → relationalStmt | logicalStmt | BOOLEAN_VAL COMMA
11. arithmeticStmt → (unaryOP | operand binaryOP) operand COMMA
12. relationalStmt → operand relationalOP operand COMMA
13. logicalStmt → logical_operand logicalOP logical_operand COMMA
14. operand → IDENTIFIER | NUM_VAL | STRING_VAL
15. logical_operand → IDENTIFIER | BOOLEAN_VAL
16. unaryOP → INCREMENT | DECREMENT
17. binaryOP → ADD | SUB | MULTIPLY| DIVID| REMINDER | POWER
18. logicalOP → AND | OR | NOT
19. relationalOP → GREATER_THAN | LESS_THAN | LESS_THAN_OR_EQUAL | GREATER THAN_OR_ EQUAL|IS_EQUAL|NOT_EQUAL
20. DataType → INT_TYPE | BOOL_TYPE | FLOAT_TYPE | STR_TYPE
21. IDENTIFIER → TILDE_SYMBOL (LETTER | DIGIT)+
22. NUM_VAL → DIGIT+ (. DIGIT+) ?
23. BOOLEAN_VAL → "true" | "false"
24. STRING_VAL → (LETTER | SYMBOL | DIGIT)+
25. LETTER → ["A"-"Z", "a"- "z"]
26. DIGIT → ["0"-"9"]
27. SYMBOL → "#"| "$" | "?" | "_"
28. ADD → "+"
29. SUB → "-"
30. MULTIPLY → "*"
31. DIVIDE →"/"
32. REMINDER →"%"
33. POWER→"^^"

34. INCREMENT →"++"
35. DECREMENT→"--"
36. ASSIGNMENT→"="
37. AND →"&&"
38. OR →"||"
39. NOT → "!"
40. IS_EQUAL → "=="
41. NOT_EQUAL → "!="
42. GREATER_THAN → ".>"
43. LESS_THAN → ".<"
44. GREATER_THAN_OR_EQUAL → ">="
45. LESS_THAN_OR_EQUAL → "<="
46. TILDE_SYMBOL → "~"
47. COMMA → ","
48. LEFT_ANG_BRACKETS → "<"
49. RIGHT_ANG_BRACKETS→ ">"
50. LEFT_BRACKETS → "("
51. RIGHT_BRACKETS → ")"
52. SEPRATOR → "|"
53. COLON → ":"
54. SEMICOLON → ";"
55. IF → "if"
56. THEN → "then"
57. INT_TYPE → "INT"
58. BOOL_TYPE → "BOOL"
59. FLOAT_TYPE → "FLOAT"
60. STR_TYPE → "STR"
61. ELSE → "else"
62. LOOP → "loop"
63. NARSOUT → "NARSout"
64. RETURN → "return"
65. ARRAY→ "array"
66. LIST → "list"
67. ENDIF → "endif"
68. ENDLOOP → "endloop"
69. CONSTANT → "final"
70. EXIT →"exit"

2.2. BNF Explanation:

1. Start → Stmts | Exit
   Nars' language starts with a statement type, or the keyword "exit" to quit.

2. Stmts → declerationStmt | arithmeticStmt | assignmentStmt | relationalStmt | logicalStmt | iterativeStmt | conditionalStmt | listStmt | arrayStmt | printStmt
   A Nars' statements can be an declaration, arithmetic, assignment, relational, logical, iterative, conditional, array, list, or print statement.

3. declerationStmt → (CONSTANT)? DataType IDENTIFIER COMMA
   The declaration statement starts with the data type followed by an identifier name in a basic way. You may define a constant identifier by using the keyword "final" before the data type.

4. assignmentStmt → IDENTIFIER ASSIGNMENT (STRING_VAL | BOOLEAN_VAL | NUM_VAL | IDENTIFIER) COMMA
   The assignment statement starts with an identifier, an assignment operator, followed by the assigned string, number, Boolean value, or another identifier.

5. printStmt → NARSOUT LEFT_BRACKETS DOUBLE_QUTATION (STRING_VAL)+ DOUBLE_QUTATION RIGHT_BRACKETS COMMA
   A print statement starts with the keyword " NARSout" followed by opening brackets and the string between double quotation marks. Ends with closing brackets and comma.

6. listStmt → LIST IDENTIFIER ASSIGNMENT LEFT_BRACKETS ((NUM_VAL) SEMICOLON) + RIGHT_BRACKETS COMMA

   A List statement starts with the keyword "list", followed by an identifier, an assignment operator, and one or more numbers separated by a semicolon, all numbers enclosed between brackets, ending with a comma.

7. arrayStmt → ARRAY LEFT_ANG_BRACKETS (INT_TYPE | BOOL_TYPE | FLOAT_TYPE| STR_TYPE) RIGHT_ANG_BRACKETS IDENTIFIER ASSIGNMENT LEFT_BRACKETS ((NUM_VAL | STRING_VAL| BOOLEAN_VAL) SEMICOLON)+ RIGHT_BRACKETS COMMA
   An Array statement starts with the keyword "array", followed by data type between two angle brackets"<" and ">". After that identifier, an assignment operator, and one or more elements separated by a semicolon, all data is enclosed between brackets, ending with a comma.

8. iterativeStmt → LOOP LEFT_ANG_BRACKETS conditionalExpression RIGHT_ANG_BRACKETS (Stmts)+ ENDLOOP COMMA
   The definition of an iterative statement starts with the keyword “loop” and writes a condition between the symbols "<" and ">" And at least one statement to iterate. It ends with the keyword "endloop." And comma.

9. conditionalStmt → IF LEFT_ANG_BRACKETS conditionalExpression RIGHT_ANG_BRACKETS THEN (Stmts)+ (ELSE (Stmts)+)? ENDIF COMMA
   if -then-endif: To define an if statement, start with the keyword “if” and write a condition between the symbols "<" and ">" Follow it with "then" and at least one statement and end with "endif " . And comma.
   if -then-else-endif: If the condition is not satisfied, you can add the "else" part and write the statement that is done if the condition is not satisfied.

10. conditionalExpression → relationalStmt | logicalStmt | BOOLEAN_VAL COMMA
    The conditional expression can be a relational, logical statement, or Boolean literal end with a comma.

11. arithmeticStmt → (unaryOP | operand binaryOP) operand COMMA
    An arithmetic statement can be binary with two operands and one operator. It can be a unary operator and one operand, ending with a comma.

12. relationalStmt → operand relationalOP operand COMMA
    A relational statement consists of two operands with a relational operator between them and ends with a comma.

13. logicalStmt → logical_operand logicalOP logical_operand COMMA
    A logical statement consists of two logical operands and a logical operator between them and ends with a comma.

14. operand → IDENTIFIER | NUM_VAL | STRING_VAL
    An operand can be either an identifier or a number value or string value.

15. logical_operand → IDENTIFIER | BOOLEAN_VAL
    A logical operand can be either an identifier or a Boolean value.

16. unaryOP → INCREMENT | DECREMENT
    Numerous unary operators that are used in arithmetic statements.

17. binaryOP → ADD | SUB | MULTIPLY| DIVID| REMINDER | POWER
    Numerous binary operators are used in arithmetic statements.

18. logicalOP → AND | OR | NOT
    Numerous logical operators are used in logical statements.

19. relationalOP → IS_EQUAL | NOT_EQUAL | GREATER_THAN | LESS_THAN | GREATER_THAN_OR_EQUAL | LESS_THAN_OR_EQUAL
    Numerous relational operators are used in comparison statements.

20. DataType→ STR_TYPE | BOOL_TYPE | INT_TYPE | FLOAT_TYPE
    The type of an identifier can be a string, boolean, integer, or float.

21. IDENTIFIER→ TILDE_SYMBOL (LETTER | DIGIT)+
    Any identifier must start with a tilde symbol followed by one or more letters and digits.

22. NUM_VAL → DIGIT+ (. DIGIT+)?
    A number value can be an integer or double. It must start with one or more digit, and its optional to use the fractional part.

23. BOOLEAN_VAL → "true" | "false"
    A boolean value can be either a true or false.

24. STRING_VAL → (LETTER | SYMBOL | DIGIT)+
    A string can be any one or more combination of letters, symbols, or digit.

The NARS' BNF keywords and symbols are in the following section:

25. LETTER → ["A"-"Z", "a"- "z"]
26. DIGIT → ["0"-"9"]
27. SYMBOL → "#" |"$" | "?" | "_"
28. ADD → "+"
29. SUB → "-"
30. MULTIPLY → "*"
31. DIVIDE →"/"
32. REMINDER →"%"
33. POWER→"^^"
34. INCREMENT →"++"
35. DECREMENT→"--"
36. ASSIGNMENT→"="
37. AND →"&&"
38. OR →"||"
39. NOT → "!"
40. IS_EQUAL → "=="
41. NOT_EQUAL → "!="
42. GREATER_THAN → ".>"
43. LESS_THAN → ".<"
44. GREATER_THAN_OR_EQUAL → ">="
45. LESS_THAN_OR_EQUAL → "<="
46. TILDE_SYMBOL → "~"
47. COMMA → ","
48. LEFT_ANG_BRACKETS → "<"
49. RIGHT_ANG_BRACKETS→ ">"
50. LEFT_BRACKETS → "("
51. RIGHT_BRACKETS → ")"
52. SEPRATOR → "|"
53. COLON → ":"
54. SEMICOLON → ";"
55. IF → "if"
56. THEN → "then"
57. INT_TYPE → "INT"
58. BOOL_TYPE → "BOOL"
59. FLOAT_TYPE → "FLOAT"
60. STR_TYPE → "STR"
61. ELSE → "else"
62. LOOP → "loop"
63. NARSOUT → "Narsout"
64. RETURN → "return"
65. ARRAY→ "array"
66. LIST→"list"
67. ENDIF → "endif"
68. ENDLOOP → "endloop"
69. CONSTANT → "final"
70. EXIT →"exit"

2.3. Production Rules in JavaCC

1. Start → Stmts | Exit

```
void Start() :

{}
{
  Stmts() | <EXIT>
}
```

2. Stmts → declerationStmt | arithmeticStmt | assignmentStmt | relationalStmt | logicalStmt | iterativeStmt | conditionalStmt | listStmt | arrayStmt | printStmt

```
void Stmts() :
{}
{
     declerationStmt()
   | LOOKAHEAD(2) arithmeticStmt()
   | LOOKAHEAD(3) assignmentStmt ()
   | LOOKAHEAD(2) relationalStmt ()
   | LOOKAHEAD(3) logicalStmt()
   | iterativeStmt()
   | conditionalStmt()
   | listStmt()
   | arrayStmt()
   | printStmt ()

}
```

3. declerationStmt → (CONSTANT)? DataType IDENTIFIER COMMA

```
void declerationStmt() :
{}
{
  (<CONSTANT>)? DataType() (< IDENTIFIER >) <COMMA>
}
```

4. assignmentStmt → IDENTIFIER ASSIGNMENT (STRING_VAL | BOOLEAN_VAL | NUM_VAL | IDENTIFIER) COMMA

```
void assignmentStmt ():
{}
{
   < IDENTIFIER > <ASSIGNMENT>
  (< STRING_VAL >|<BOOLEAN_VAL>|< NUM_VAL >|< IDENTIFIER> ) <COMMA>
}
```

5. printStmt → NARSOUT LEFT_BRACKETS DOUBLE_QUTATION (STRING_VAL)+ DOUBLE_QUTATION RIGHT_BRACKETS COMMA

```
void printStmt ():
{}
{
< NARSOUT > <LEFT_BRACKETS > < DOUBLE_QUTATION>
(< STRING_VAL >)+ < DOUBLE_QUTATION><RIGHT_BRACKETS >
<COMMA>
}
```

6. listStmt → LIST IDENTIFIER ASSIGNMENT LEFT_BRACKETS ((NUM_VAL) SEMICOLON)+ RIGHT_BRACKETS COMMA

```
void listStmt():
{}
{
  <LIST> < IDENTIFIER > <ASSIGNMENT> <LEFT_BRACKETS >
  ((< NUM_VAL >) < SEMICOLON >)+<RIGHT_BRACKETS >
  <COMMA>
  }
```

7. arrayStmt → ARRAY LEFT_ANG_BRACKETS (INT_TYPE | BOOL_TYPE | FLOAT_TYPE| STR_TYPE) RIGHT_ANG_BRACKETS IDENTIFIER ASSIGNMENT LEFT_BRACKETS ((NUM_VAL | STRING_VAL| BOOLEAN_VAL) SEMICOLON)+ RIGHT_BRACKETS COMMA

```
void arrayStmt():
{}
{
   <ARRAY> <LEFT_ANG_BRACKETS >
  (< INT_TYPE > | < BOOL_TYPE >| < FLOAT_TYPE >| < STR_TYPE >)
  < RIGHT_ANG_BRACKETS >
  < IDENTIFIER > <ASSIGNMENT> <LEFT_BRACKETS >
  ((< NUM_VAL > | < STRING_VAL >|< BOOLEAN_VAL>)< SEMICOLON >)+
  <RIGHT_BRACKETS >
  <COMMA>
 }
```

8. iterativeStmt → LOOP LEFT_ANG_BRACKETS conditionalExpression RIGHT_ANG_BRACKETS (Stmts)+ ENDLOOP COMMA

```
void iterativeStmt():
{}
{
  < LOOP > < LEFT_ANG_BRACKETS > conditionalExpression()
  < RIGHT_ANG_BRACKETS > (Stmts())+ < ENDLOOP > <COMMA>
}
```

9. conditionalStmt → IF LEFT_ANG_BRACKETS conditionalExpression RIGHT_ANG_BRACKETS THEN (Stmts)+ (ELSE (Stmts)+)? ENDIF COMMA

```
void conditionalStmt():
{}
{
  <IF><LEFT_ANG_BRACKETS> conditionalExpression()
  < RIGHT_ANG_BRACKETS > <THEN>
  (Stmts())+(< ELSE > (Stmts())+)? < ENDIF > <COMMA>
}
```

10. conditionalExpression → relationalStmt | logicalStmt | BOOLEAN_VAL COMMA

```
void conditionalExpression():

{}
{
        LOOKAHEAD(2) relationalStmt ()
      | LOOKAHEAD(2) logicalStmt ()
      | < BOOLEAN_VAL >
      <COMMA>
}
```

11. arithmeticStmt → (unaryOP | operand binaryOP) operand COMMA

```
void arithmeticStmt():
{}
{
  (unaryOP ()| operand() binaryOP()) operand() <COMMA>
}
```

12. relationalStmt → operand relationalOP operand COMMA

```
void relationalStmt ():
{}
{
   operand() relationalOP() operand() <COMMA>
}
```

13. logicalStmt → logical_operand logicalOP logical_operand COMMA

```
void logicalStmt ():
{}
{
   logical_operand() logicalOP() logical_operand() <COMMA>

}
```

14. operand → IDENTIFIER | NUM_VAL | STRING_VAL

```
void operand():
{}
{
   < IDENTIFIER > | < NUM_VAL > | <STRING_VAL>
}
```

15. logical_operand → IDENTIFIER | BOOLEAN_VAL

```
void logical_operand():
{}
{
 < IDENTIFIER > | < BOOLEAN_VAL >
}
```

16. unaryOP → INCREMENT | DECREMENT

```
void unaryOP():
{}
{
 < INCREMENT > | < DECREMENT >
}
```

17. binaryOP → ADD | SUB | MULTIPLY| DIVID| REMINDER | POWER

```
void binaryOP():
{}
{
 < ADD > | < SUB > | < MULTIPLY > |
 < DIVID > | < REMINDER > | < POWER >
}
```

18. logicalOP → AND | OR | NOT

```
void logicalOP():
{}
{
< AND >|< OR >|< NOT >
}
```

19. relationalOP → IS_EQUAL | NOT_EQUAL | GREATER_THAN | LESS_THAN | GREATER_THAN_OR_EQUAL | LESS_THAN_OR_EQUAL

```
void relationalOP():
{}
{
   < GREATER_THAN > | < LESS_THAN > |
   < LESS_THAN_OR_EQUAL >|< GREATER_THAN_OR_EQUAL > |
   < IS_EQUAL > | <NOT_EQUAL >
}
```

20. DataType→ STR_TYPE | BOOL_TYPE | INT_TYPE | FLOAT_TYPE

```
void DataType ():
{}
{
   < INT_TYPE >|< BOOL_TYPE >|<FLOAT_TYPE >|< STR_TYPE >
}
```

21. IDENTIFIER→ TILDE_SYMBOL (LETTER | DIGIT)+

```
TOKEN: /*Identifiers*/
{
 < IDENTIFIER : < TILDE_SYMBOL >(<LETTER>|< DIGIT >)+ >
}
```

22. NUM_VAL → DIGIT+ (. DIGIT+)?
23. BOOLEAN_VAL → "true" | "false"
24. STRING_VAL → (LETTER | SYMBOL | DIGIT)+

```
TOKEN: /*Data types */
{

   < NUM_VAL : (< DIGIT >)+ (("."(< DIGIT >)+)?) >
 | < BOOLEAN_VAL : "true"| "false" >
 | < STRING_VAL : (<LETTER >|< SYMBOL >|< DIGIT >)+ >
}
```

25. LETTER → ["A"-"Z", "a"- "z"]
26. DIGIT → ["0"-"9"]
27. SYMBOL → "#" |"$" | "?" | "_"

```
TOKEN: /*alphabets*/
{
 < LETTER :(["A"-"Z","a"-"z"])>
| < DIGIT : ["0"-"9"] >
| < SYMBOL : "#" | "$" | "?" | "_" >
}
```

28. ADD → "+"
29. SUB → "-"
30. MULTIPLY → "*"
31. DIVIDE →"/"
32. REMINDER →"%"
33. POWER→"^^"
34. INCREMENT →"++"
35. DECREMENT→"--"
36. ASSIGNMENT→"="

```
TOKEN : /* OPERATORS*/
{
  < ADD : "+" >
| < SUB : "-" >
| < MULTIPLY : "*" >
| < DIVID : "/" >
| < REMINDER : "%" >
| < POWER : "^^" >
| < INCREMENT : "++" >
| < DECREMENT : "--" >
| < ASSIGNMENT : "=" >
}
```

37. AND →"&&"
38. OR →"||"
39. NOT → "!"

```
TOKEN: /* Logical Operations */
{
  < AND : "&&" >
| < OR : "||" >
| < NOT : "!" >
}
```

40. IS_EQUAL → "=="
41. NOT_EQUAL → "!="
42. GREATER_THAN → ".>"
43. LESS_THAN → ".<"
44. GREATER_THAN_OR_EQUAL → ">="
45. LESS_THAN_OR_EQUAL → "<="

```
TOKEN :/* Relational Operations */
{
  < IS_EQUAL :"==" >
| < NOT_EQUAL :"!=" >
| < GREATER_THAN : ".>" >
| < LESS_THAN : ".<" >
| < GREATER_THAN_OR_EQUAL :">=" >
| < LESS_THAN_OR_EQUAL :"<=" >
}
```

46. TILDE_SYMBOL → "~"
47. COMMA → ","
48. LEFT_ANG_BRACKETS → "<"
49. RIGHT_ANG_BRACKETS→ ">"
50. LEFT_BRACKETS → "("
51. RIGHT_BRACKETS → ")"
52. SEPRATOR → "|"
53. COLON → ":"
54. SEMICOLON → ";"

```
TOKEN: /*Punctuation Marks*/
{
  < TILDE_SYMBOL : "~" > | < COMMA : "," > | < DOUBLE_QUTATION : "\"" >
| < LEFT_ANG_BRACKETS : "<" > | < RIGHT_ANG_BRACKETS : ">" >
| < LEFT_BRACKETS :"(" > | < RIGHT_BRACKETS:")" > | < SEPARATOR : "|" >
| < COLON : ":" >| < SEMICOLON : ";" >
}
```

55. IF → "if"
56. THEN → "then"
57. INT_TYPE → "INT"
58. BOOL_TYPE → "BOOL"
59. FLOAT_TYPE → "FLOAT"
60. STR_TYPE → "STR"
61. ELSE → "else"
62. LOOP → "loop"
63. NARSOUT → "Narsout"
64. RETURN → "return"
65. ARRAY→ "array"
66. LIST→"list"
67. ENDIF → "endif"
68. ENDLOOP → "endloop"
69. CONSTANT → "final"
70. EXIT →"exit"

```
TOKEN: /*Keywords*/
{
  < IF           : "if" >
| < THEN         : "then" >
| < INT_TYPE     : "INT" >
| < BOOL_TYPE    : "BOOL" >
| < FLOAT_TYPE   : "FLOAT" >
| < STR_TYPE     : "STR" >
| < ELSE         : "else" >
| < LOOP         : "loop" >
| < NARSOUT      : "NARSout" >
| < RETURN       : "return" >
| < ARRAY        : "array" >
| < LIST         : "list" >
| < ENDIF        : "endif" >
| < ENDLOOP      : "endloop" >
| < CONSTANT     : "final" >
| < EXIT         : "exit" > { System.out.println("Thank you for using
NARS! hope to see you soon."); System.exit(0); }
}
```

3 Phase 3: Semantic Analysis

3.1. Grammar and Build Using JJTree

```
/**
 * JJTree template file created by SF JavaCC plugin 1.5.28+ wizard for
JavaCC 1.5.0+
 * N-> NOOR & NSREEN, A-> ASMA, R-> RENAD, S-> SARA SO ( NARS )
 */
options
{
  static = true;
}

PARSER_BEGIN(NARSt)
package NARS_T;

import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class NARSt
{
  public static void main(String args [])
  {

    try {
        new NARSt(new FileInputStream("NARS_SAMPLE.txt"));
    }
    catch (FileNotFoundException e)
    {
            System.out.println("There is no available file of sample
code.");
            System.out.println(e.getMessage());
    }
      System.out.println("****** Wlecome to NARS Programming Lanaguage
******");
            System.out.println("Reading from input file...\n\n");

    while (true) {
            try {
                  System.out.println("-----------------------------------
------");
                  SimpleNode n = NARSt.Start();
                  n.dump(" >>");
                  System.out.print("Syntctically correct statements.\n");
        }// end try
        catch(Exception e)
                   {
                         System.out.println("Error encountered");
                         System.out.println(e.getMessage());
                         break;
                  }// end catch
            }// end while
  } //end main
}// end class

PARSER_END(NARSt)

SKIP :
```

```
{
  < WHIT_SPACE      : " "  >
| < AT_SIGN  : "@"  >
| < TAP                    : "\t" >
| < NEW_LINE : "\n" >
| < NEW_LINE2: "\r" >
}

TOKEN : /* OPERATORS*/
{
  < ADD : "+" >
| < SUB : "-" >
| < MULTIPLY : "*" >
| < DIVID : "/" >
| < REMINDER : "%" >
| < POWER : "^^" >
| < INCREMENT : "++" >
| < DECREMENT : "--" >
| < ASSIGNMENT : "=" >
}

TOKEN: /* Logical Operations */
{
  < AND : "&&" >
| < OR : "||" >
| < NOT : "!" >
}

TOKEN :/* Relational Operations */
{
  < IS_EQUAL :"==" >
| < NOT_EQUAL :"!=" >
| < GREATER_THAN : ".>" >
| < LESS_THAN : ".<" >
| < GREATER_THAN_OR_EQUAL :">=" >
| < LESS_THAN_OR_EQUAL :"<=" >
}

TOKEN: /*Punctuation Marks*/
{
  < TILDE_SYMBOL : "~" > | < COMMA : "," > | < DOUBLE_QUTATION : "\"" >
| < LEFT_ANG_BRACKETS : "<" > | < RIGHT_ANG_BRACKETS : ">" >
| < LEFT_BRACKETS :"(" > | < RIGHT_BRACKETS:")" > | < SEPARATOR : "|" >
| < COLON : ":" >| < SEMICOLON : ";" >
}


TOKEN: /*Keywords*/
{
  < IF              : "if" >
| < THEN            : "then" >
| < INT_TYPE : "INT" >
| < BOOL_TYPE       : "BOOL" >
| < FLOAT_TYPE      : "FLOAT" >
| < STR_TYPE : "STR" >
| < ELSE            : "else" >
| < LOOP            : "loop" >
| < NARSOUT   : "NARSout" >
| < RETURN          : "return" >
```

```
| < ARRAY           : "array" >
| < LIST            : "list" >
| < ENDIF           : "endif" >
| < ENDLOOP   : "endloop" >
| < CONSTANT : "final" >
| < EXIT            : "exit" > { System.out.println("Thank you for using
NARS! hope to see you soon."); System.exit(0); }
}

TOKEN: /*Identifiers*/
{
 < IDENTIFIER : < TILDE_SYMBOL >(<LETTER>|< DIGIT >)+ >
}

TOKEN: /*Data types */
{

   < NUM_VAL : (< DIGIT >)+ (("."(< DIGIT >)+)?) >
 | < BOOLEAN_VAL : "true"| "false" >
 | < STRING_VAL : (<LETTER >|< SYMBOL >|< DIGIT >)+ >
}

TOKEN: /*alphabets*/
{
 < LETTER :(["A"-"Z","a"-"z"])>
| < DIGIT : ["0"-"9"] >
| < SYMBOL : "#" | "$" | "?" | "_" >
}


SPECIAL_TOKEN:/*Comments*/
{
  < SINGLE_LINE: "@" (~["\n"])* "@" >
}

SimpleNode Start():{Token tt; }/*Start*/
      {
        Stmts()
            {return jjtThis; }
      | EXIT ()
            {return jjtThis; }
      }

void Stmts() : /*Stmts*/
{}
{
     declerationStmt()
   | LOOKAHEAD(2) arithmeticStmt()
   | LOOKAHEAD(3) assignmentStmt ()
   | LOOKAHEAD(2) relationalStmt ()
   | LOOKAHEAD(3) logicalStmt()
   | iterativeStmt()
   | conditionalStmt()
   | listStmt()
   | arrayStmt()
   | printStmt ()

}
```

```
void declerationStmt() :/*declerationStmt*/
{}
{
  (CONSTANT())? DataType() (IDENTIFIER()) COMMA()
}

void assignmentStmt ():/*assignmentStmt*/
{}
{
   IDENTIFIER() ASSIGNMENT()
  (STRING_VAL() | BOOLEAN_VAL()| NUM_VAL()| IDENTIFIER()) COMMA()
  }

void printStmt (): /*printStmt*/
{}
{
NARSOUT() LEFT_BRACKETS() DOUBLE_QUTATION()
(STRING_VAL())+ DOUBLE_QUTATION() RIGHT_BRACKETS() COMMA()
}

void listStmt():/*listStmt*/
{}
{
  LIST() IDENTIFIER() ASSIGNMENT() LEFT_BRACKETS()
  ((NUM_VAL()) SEMICOLON())+ RIGHT_BRACKETS()
  COMMA()
  }


void arrayStmt():/*arrayStmt*/
{}
{
   ARRAY() LEFT_ANG_BRACKETS()
  ( INT_TYPE() | BOOL_TYPE()| FLOAT_TYPE() |  STR_TYPE())
  RIGHT_ANG_BRACKETS()
  IDENTIFIER() ASSIGNMENT() LEFT_BRACKETS()
  ((NUM_VAL()| STRING_VAL()|BOOLEAN_VAL()) SEMICOLON())+
  RIGHT_BRACKETS()
  COMMA()
 }


void iterativeStmt():/*iterativeStmt*/
{}
{
  LOOP() LEFT_ANG_BRACKETS() conditionalExpression()
  RIGHT_ANG_BRACKETS() (Stmts())+ ENDLOOP() COMMA()

}

void conditionalStmt():/*conditionalStmt*/
{}
{
  IF() LEFT_ANG_BRACKETS() conditionalExpression()
  RIGHT_ANG_BRACKETS() THEN()
  (Stmts())+ (ELSE()(Stmts())+)? ENDIF() COMMA()
}

void conditionalExpression():/*conditionalExpression*/
```

```
{}
{
       LOOKAHEAD(2) relationalStmt ()
     | LOOKAHEAD(2) logicalStmt ()
     | BOOLEAN_VAL()
     COMMA()
}

/* arithmetic stmt both unary and binary */
void arithmeticStmt():/*arithmeticStmt*/
{}
{
  (unaryOP ()| operand() binaryOP()) operand() COMMA()

}

void relationalStmt ():/*relationalStmt*/
{}
{
   operand() relationalOP() operand() COMMA()
}

void logicalStmt ():/*logicalStmt*/
{}
{
   logical_operand() logicalOP() logical_operand() COMMA()
}

void operand():/*operand*/
{}
{
   IDENTIFIER()| NUM_VAL() | STRING_VAL()
}

void logical_operand():/*logical_operand*/
{}
{
 IDENTIFIER() | BOOLEAN_VAL()
}

void unaryOP():/*unaryOP*/
{}
{
 INCREMENT() | DECREMENT()
}

void binaryOP():/*binaryOP*/
{}
{
 ADD() | SUB() | MULTIPLY() |
 DIVID() | REMINDER() | POWER()
}

void logicalOP():/*logicalOP*/
{}
{
AND()|OR()|NOT()
}
```

```
void relationalOP():/*relationalOP*/
{}
{
   GREATER_THAN() | LESS_THAN()|
   LESS_THAN_OR_EQUAL()|GREATER_THAN_OR_EQUAL() |
   IS_EQUAL()| NOT_EQUAL()
}

/* data type of identifer */
void DataType () :
{}
{
   INT_TYPE() |BOOL_TYPE()|FLOAT_TYPE()|STR_TYPE()
}

/*OPERATORS*/
void ADD(): { Token tt;} {
tt= < ADD > { jjtThis.jjtSetValue(tt.image); }
}

void SUB(): { Token tt;} {
tt= < SUB > { jjtThis.jjtSetValue(tt.image); }
}

void MULTIPLY(): { Token tt;} {
tt= < MULTIPLY > { jjtThis.jjtSetValue(tt.image); }
}

void DIVID(): { Token tt;} {
tt= < DIVID > { jjtThis.jjtSetValue(tt.image); }
}

void REMINDER(): { Token tt;} {
tt= < REMINDER > { jjtThis.jjtSetValue(tt.image); }
}

void POWER(): { Token tt;} {
tt= < POWER > { jjtThis.jjtSetValue(tt.image); }
}

void INCREMENT(): { Token tt;} {
tt= < INCREMENT > { jjtThis.jjtSetValue(tt.image); }
}

void DECREMENT(): { Token tt;} {
tt= < DECREMENT > { jjtThis.jjtSetValue(tt.image); }
}

void ASSIGNMENT(): { Token tt;} {
tt= < ASSIGNMENT > { jjtThis.jjtSetValue(tt.image); }
}

/*Logical Operations*/
void AND(): { Token tt;} {
tt= < AND > { jjtThis.jjtSetValue(tt.image); }
}

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

```
}

void NOT(): { Token tt;} {
tt= < NOT > { jjtThis.jjtSetValue(tt.image); }
}

/*rational Operations */
void IS_EQUAL(): { Token tt;} {
tt= < IS_EQUAL > { jjtThis.jjtSetValue(tt.image); }
}
void NOT_EQUAL(): { Token tt;} {
tt= < NOT_EQUAL > { jjtThis.jjtSetValue(tt.image); }
}
void GREATER_THAN(): { Token tt;} {
tt= < GREATER_THAN > { jjtThis.jjtSetValue(tt.image); }
}
void LESS_THAN(): { Token tt;} {
tt= < LESS_THAN > { jjtThis.jjtSetValue(tt.image); }
}
void GREATER_THAN_OR_EQUAL(): { Token tt;} {
tt= < GREATER_THAN_OR_EQUAL > { jjtThis.jjtSetValue(tt.image); }
}
void LESS_THAN_OR_EQUAL(): { Token tt;} {
tt= < LESS_THAN_OR_EQUAL > { jjtThis.jjtSetValue(tt.image); }
}

/*Punctuation Marks*/

void TILDE_SYMBOL(): { Token tt;} {
tt= < TILDE_SYMBOL > { jjtThis.jjtSetValue(tt.image); }
}

void  COMMA(): { Token tt;} {
tt= <  COMMA > { jjtThis.jjtSetValue(tt.image); }
}

void  SEMICOLON(): { Token tt;} {
tt= <  SEMICOLON > { jjtThis.jjtSetValue(tt.image); }
}

void  COLON(): { Token tt;} {
tt= <  COLON > { jjtThis.jjtSetValue(tt.image); }
}

void  LEFT_BRACKETS(): { Token tt;} {
tt= <  LEFT_BRACKETS > { jjtThis.jjtSetValue(tt.image); }
}

void  RIGHT_BRACKETS(): { Token tt;} {
tt= <  RIGHT_BRACKETS > { jjtThis.jjtSetValue(tt.image); }
}
void  LEFT_ANG_BRACKETS(): { Token tt;} {
tt= <  LEFT_ANG_BRACKETS > { jjtThis.jjtSetValue(tt.image); }
}
void  RIGHT_ANG_BRACKETS (): { Token tt;} {
tt= <  RIGHT_ANG_BRACKETS  > { jjtThis.jjtSetValue(tt.image); }
}

void  DOUBLE_QUTATION (): { Token tt;} {
```

```
tt= <  DOUBLE_QUTATION  > { jjtThis.jjtSetValue(tt.image); }
}

void  SEPARATOR (): { Token tt;} {
tt= <  SEPARATOR  > { jjtThis.jjtSetValue(tt.image); }
}


/*Keywords*/

void IF(): { Token tt;} {
tt= < IF > { jjtThis.jjtSetValue(tt.image); }
}

void THEN(): { Token tt;} {
tt= < THEN > { jjtThis.jjtSetValue(tt.image); }
}

void INT_TYPE(): { Token tt;} {
tt= < INT_TYPE > { jjtThis.jjtSetValue(tt.image); }
}

void BOOL_TYPE(): { Token tt;} {
tt= < BOOL_TYPE > { jjtThis.jjtSetValue(tt.image); }
}

void ELSE(): { Token tt;} {
tt= < ELSE > { jjtThis.jjtSetValue(tt.image); }
}

void FLOAT_TYPE(): { Token tt;} {
tt= < FLOAT_TYPE > { jjtThis.jjtSetValue(tt.image); }
}

void STR_TYPE(): { Token tt;} {
tt= < STR_TYPE > { jjtThis.jjtSetValue(tt.image); }
}

void LOOP(): { Token tt;} {
tt= < LOOP > { jjtThis.jjtSetValue(tt.image); }
}

void NARSOUT(): { Token tt;} {
tt= < NARSOUT > { jjtThis.jjtSetValue(tt.image); }
}

void RETURN(): { Token tt;} {
tt= < RETURN > { jjtThis.jjtSetValue(tt.image); }
}

void ARRAY(): { Token tt;} {
tt= < ARRAY > { jjtThis.jjtSetValue(tt.image); }
}

void ENDIF(): { Token tt;} {
tt= < ENDIF > { jjtThis.jjtSetValue(tt.image); }
}

void ENDLOOP(): { Token tt;} {
```

```
tt= < ENDLOOP > { jjtThis.jjtSetValue(tt.image); }
}

void CONSTANT(): { Token tt;} {
tt= < CONSTANT > { jjtThis.jjtSetValue(tt.image); }
}

void LIST(): { Token tt;} {
tt= < LIST > { jjtThis.jjtSetValue(tt.image); }
}

void EXIT(): { Token tt;} {
tt= < EXIT > { jjtThis.jjtSetValue(tt.image); }
}


/*Identifiers*/
void IDENTIFIER(): { Token tt;} {
tt= < IDENTIFIER > { jjtThis.jjtSetValue(tt.image); }
}

/*Data types*/
void NUM_VAL(): { Token tt;} {
tt= < NUM_VAL > { jjtThis.jjtSetValue(tt.image); }
}
void BOOLEAN_VAL(): { Token tt;} {
tt= < BOOLEAN_VAL > { jjtThis.jjtSetValue(tt.image); }
}
void STRING_VAL(): { Token tt;} {
tt= < STRING_VAL > { jjtThis.jjtSetValue(tt.image); }
}



/*alphabets*/
void LETTER(): { Token tt;} {
tt= < LETTER > { jjtThis.jjtSetValue(tt.image); }
}
void DIGIT(): { Token tt;} {
tt= < DIGIT > { jjtThis.jjtSetValue(tt.image); }
}
void SYMBOL(): { Token tt;} {
tt= < SYMBOL > { jjtThis.jjtSetValue(tt.image); }
}
```

3.2. Screenshots of Output

The output shows the code result of a sample NARS_SAMPLE.txt file that can be found in— [Appendix C].

3.2.1. jj file Screenshots:

NARSGrammar.jj | Console × | NARS_SAMPLE.txt | NARSt.jjt | *NARS_SAMPLE.txt

<terminated> NARSGrammar [Java Application] C:\Users\nsree\.p2\pool\plugins\org.eclipse.justj.openjdk.hotsp

```
****** Wlecome to NARS Programming Lanaguage ******
Reading from input file...


Found a Comment Statement

Found a Declaration Statement
**Syntctically correct statments**

Found a Declaration Statement
**Syntctically correct statments**

Found a Declaration Statement
**Syntctically correct statments**

Found an Arithmetic Statement
**Syntctically correct statments**

Found a Logical Statement
**Syntctically correct statments**

Found a Relational Statement
**Syntctically correct statments**

Found a Relational Statement
Found an Assignment Statement
Found a conditional Statement
**Syntctically correct statments**

Found a Relational Statement
Found an Assignment Statement
Found an Assignment Statement
Found a conditional Statement
**Syntctically correct statments**

Found a Relational Statement
Found an Arithmetic Statement
Found an iterative Statement
**Syntctically correct statments**

Found a Comment Statement

Found a print Statement
**Syntctically correct statments**

Found a Declaration Statement
**Syntctically correct statments**

Found an Assignment Statement
**Syntctically correct statments**

Found a list Statement
**Syntctically correct statments**

Found an Array Statement
**Syntctically correct statments**

Thank you for using NARS! hope to see you soon.
```

3.2.2. jjt file Screenshots:

Input	Output
@N-> NOOR & NSREEN, A-> ASMA, R-> RENAD, S-> SARA SO (NARS)@	
STR ~name,	>>Start >> Stmts >> declerationStmt >> DataType >> STR_TYPE:STR >> IDENTIFIER:~name >> COMMA:, Syntctically correct statements. ----------
final INT ~x,	>>Start >> Stmts >> declerationStmt >> CONSTANT:final >> DataType >> INT_TYPE:INT >> IDENTIFIER:~x >> COMMA:, Syntctically correct statements. ----------
BOOL ~NARS,	>>Start >> Stmts >> declerationStmt >> DataType >> BOOL_TYPE:BOOL >> IDENTIFIER:~NARS >> COMMA:, Syntctically correct statements. ----------
++10,	>>Start >> Stmts >> arithmeticStmt >> unaryOP >> INCREMENT:++ >> operand >> NUM_VAL:10 >> COMMA:, Syntctically correct statements. ----------

true && true,	>>Start >> Stmts >> logicalStmt >> logical_operand >> BOOLEAN_VAL:true >> logicalOP >> AND:&& >> logical_operand >> BOOLEAN_VAL:true >> COMMA:, Syntctically correct statements. ----------
~flag1 .< ~flag2,	>>Start >> Stmts >> relationalStmt >> operand >> IDENTIFIER:~flag1 >> relationalOP >> LESS_THAN:.< >> operand >> IDENTIFIER:~flag2 >> COMMA:, Syntctically correct statements. ----------
if < ~flag1 .< ~flag2, > then ~sum= ~flag2, endif,	>>Start >> Stmts >> conditionalStmt >> IF:if >> LEFT_ANG_BRACKETS:< >> conditionalExpression >> relationalStmt >> operand >> IDENTIFIER:~flag1 >> relationalOP >> LESS_THAN:.< >> operand >> IDENTIFIER:~flag2 >> COMMA:, >> RIGHT_ANG_BRACKETS:> >> THEN:then >> Stmts >> assignmentStmt >> IDENTIFIER:~sum >> ASSIGNMENT:= >> IDENTIFIER:~flag2 >> COMMA:, >> ENDIF:endif >> COMMA:, Syntctically correct statements. ----------

if < ~flag1 .< ~flag2,> then ~sum= ~flag2, else ~sum= ~flag1, endif,	>>Start >> Stmts >> conditionalStmt >> IF:if >> LEFT_ANG_BRACKETS:< >> conditionalExpression >> relationalStmt >> operand >> IDENTIFIER:~flag1 >> relationalOP >> LESS_THAN:.< >> operand >> IDENTIFIER:~flag2 >> COMMA:, >> RIGHT_ANG_BRACKETS:> >> THEN:then >> Stmts >> assignmentStmt >> IDENTIFIER:~sum >> ASSIGNMENT:= >> IDENTIFIER:~flag2 >> COMMA:, >> ELSE:else >> Stmts >> assignmentStmt >> IDENTIFIER:~sum >> ASSIGNMENT:= >> IDENTIFIER:~flag1 >> COMMA:, >> ENDIF:endif >> COMMA:, Syntctically correct statements. ----------
loop < ~Flag .< 5, > ++~Flag , endloop,	>>Start >> Stmts >> iterativeStmt >> LOOP:loop >> LEFT_ANG_BRACKETS:< >> conditionalExpression >> relationalStmt >> operand >> IDENTIFIER:~Flag >> relationalOP >> LESS_THAN:.< >> operand >> NUM_VAL:5 >> COMMA:, >> RIGHT_ANG_BRACKETS:> >> Stmts >> arithmeticStmt >> unaryOP >> INCREMENT:++ >> operand >> IDENTIFIER:~Flag >> COMMA:, >> ENDLOOP:endloop >> COMMA:, Syntctically correct statements. ----------

@NARS@	
NARSout ("Welcome To NARS Language "),	>>Start >> Stmts >> printStmt >> NARSOUT:NARSout >> LEFT_BRACKETS:(>> DOUBLE_QUTATION:" >> STRING_VAL:Welcome >> STRING_VAL:To >> STRING_VAL:NARS >> STRING_VAL:Language >> DOUBLE_QUTATION:" >> RIGHT_BRACKETS:) >> COMMA:, Syntctically correct statements. ----------
INT ~dal,	>>Start >> Stmts >> declerationStmt >> DataType >> INT_TYPE:INT >> IDENTIFIER:~dal >> COMMA:, Syntctically correct statements. ----------
~COST = 15,	>>Start >> Stmts >> assignmentStmt >> IDENTIFIER:~COST >> ASSIGNMENT:= >> NUM_VAL:15 >> COMMA:, Syntctically correct statements. ----------
list ~C = (2;4;8;),	>>Start >> Stmts >> listStmt >> LIST:list >> IDENTIFIER:~C >> ASSIGNMENT:= >> LEFT_BRACKETS:(>> NUM_VAL:2 >> SEMICOLON:; >> NUM_VAL:4 >> SEMICOLON:; >> NUM_VAL:8 >> SEMICOLON:; >> RIGHT_BRACKETS:) >> COMMA:, Syntctically correct statements. ----------

array <STR> ~s = (noor;nsreen;asma;renad;sara;),	>>Start >> Stmts >> arrayStmt >> ARRAY:array >> LEFT_ANG_BRACKETS:< >> STR_TYPE:STR >> RIGHT_ANG_BRACKETS:> >> IDENTIFIER:~s >> ASSIGNMENT:= >> LEFT_BRACKETS:(>> STRING_VAL:noor >> SEMICOLON:; >> STRING_VAL:nsreen >> SEMICOLON:; >> STRING_VAL:asma >> SEMICOLON:; >> STRING_VAL:renad >> SEMICOLON:; >> STRING_VAL:sara >> SEMICOLON:; >> RIGHT_BRACKETS:) >> COMMA:, Syntctically correct statements. ----------
\n	
exit,	Thank you for using NARS! hope to see you soo

Appendix A: JJ Grammar

```
/**
 * JavaCC template file created by SF JavaCC plugin 1.5.28+ wizard for
JavaCC 1.5.0+
 * N-> NOOR & NSREEN, A-> ASMA, R-> RENAD, S-> SARA SO ( NARS )
 */
options
{
  static = true;
}

PARSER_BEGIN(NARSGrammar)
package NARS;

import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class NARSGrammar
{
  public static void main(String args []) throws ParseException
  {
    try {
       NARSGrammar parser=new NARSGrammar(new
FileInputStream("NARS_SAMPLE.txt"));
    }
    catch (FileNotFoundException e)
    {
              System.out.println("There is no available file of sample
code.");
              System.out.println(e.getMessage());
    }
              System.out.println("****** Wlecome to NARS Programming
Lanaguage ******");
              System.out.println("Reading from input file...\n\n");
       while (true) {
              try {
                NARSGrammar.Start();
                System.out.println("**Syntctically correct statments**\n");
              }
              catch (Exception e) {
                           System.out.println("**Something wrong**");
                           System.out.println(e.getMessage());
                           NARSGrammar.ReInit(System.in);
              }
              catch (Error e) {
                           System.out.println("**Something wrong**");
                           System.out.println(e.getMessage());
                           break;
              }
       }
  }
}

PARSER_END(NARSGrammar)

SKIP :
{
  < WHIT_SPACE      : " "  >
```

```
| < AT_SIGN  : "@"  >
| < TAP                    : "\t" >
| < NEW_LINE : "\n" >
| < NEW_LINE2: "\r" >
}


TOKEN : /* OPERATORS*/
{
  < ADD : "+" >
| < SUB : "-" >
| < MULTIPLY : "*" >
| < DIVID : "/" >
| < REMINDER : "%" >
| < POWER : "^^" >
| < INCREMENT : "++" >
| < DECREMENT : "--" >
| < ASSIGNMENT : "=" >
}

TOKEN: /* Logical Operations */
{
  < AND : "&&" >
| < OR : "||" >
| < NOT : "!" >
}

TOKEN :/* Relational Operations */
{
  < IS_EQUAL :"==" >
| < NOT_EQUAL :"!=" >
| < GREATER_THAN : ".>" >
| < LESS_THAN : ".<" >
| < GREATER_THAN_OR_EQUAL :">=" >
| < LESS_THAN_OR_EQUAL :"<=" >
}

TOKEN: /*Punctuation Marks*/
{
  < TILDE_SYMBOL : "~" > | < COMMA : "," > | < DOUBLE_QUTATION : "\"" >
| < LEFT_ANG_BRACKETS : "<" > | < RIGHT_ANG_BRACKETS : ">" >
| < LEFT_BRACKETS :"(" > | < RIGHT_BRACKETS:")" > | < SEPARATOR : "|" >
| < COLON : ":" >| < SEMICOLON : ";" >
}


TOKEN: /*Keywords*/
{
  < IF              : "if" >
| < THEN            : "then" >
| < INT_TYPE : "INT" >
| < BOOL_TYPE       : "BOOL" >
| < FLOAT_TYPE      : "FLOAT" >
| < STR_TYPE : "STR" >
| < ELSE            : "else" >
| < LOOP            : "loop" >
| < NARSOUT   : "NARSout" >
| < RETURN          : "return" >
| < ARRAY           : "array" >
```

```
| < LIST            : "list" >
| < ENDIF           : "endif" >
| < ENDLOOP   : "endloop" >
| < CONSTANT : "final" >
| < EXIT            : "exit" > { System.out.println("Thank you for using
NARS! hope to see you soon."); System.exit(0); }
}

TOKEN: /*Identifiers*/
{
 < IDENTIFIER : < TILDE_SYMBOL >(<LETTER>|< DIGIT >)+ >
}

TOKEN: /*Data types */
{

   < NUM_VAL : (< DIGIT >)+ (("."(< DIGIT >)+)?) >
 | < BOOLEAN_VAL : "true"| "false" >
 | < STRING_VAL : (<LETTER >|< SYMBOL >|< DIGIT >)+ >
}

TOKEN: /*alphabets*/
{
 < LETTER :(["A"-"Z","a"-"z"])>
| < DIGIT : ["0"-"9"] >
| < SYMBOL : "#" | "$" | "?" | "_" >
}


SPECIAL_TOKEN:/*Comments*/
{
  < SINGLE_LINE: "@" (~["\n"])* "@" >
  {
   System.out.println("Found a Comment Statement \n");
  }
}


void Start() :
{}
{
  Stmts() | <EXIT>
}


void Stmts() :
{}
{
     declerationStmt()
    | LOOKAHEAD(2) arithmeticStmt()
    | LOOKAHEAD(3) assignmentStmt ()
    | LOOKAHEAD(2) relationalStmt ()
    | LOOKAHEAD(3) logicalStmt()
    | iterativeStmt()
    | conditionalStmt()
    | listStmt()
    | arrayStmt()
    | printStmt ()
```

```
}

/* to declar the type of the var */
void declerationStmt() :
{}
{
  (<CONSTANT>)? DataType() (< IDENTIFIER >) <COMMA>
 {
 System.out.println("Found a Declaration Statement");
 }
}
void assignmentStmt ():
{}
{
   < IDENTIFIER > <ASSIGNMENT>
  (< STRING_VAL >|<BOOLEAN_VAL>|< NUM_VAL >|< IDENTIFIER> ) <COMMA>
  {
  System.out.println("Found an Assignment Statement");
  }
  }
void printStmt ():
{}
{
< NARSOUT > <LEFT_BRACKETS > < DOUBLE_QUTATION>
(< STRING_VAL >)+ < DOUBLE_QUTATION><RIGHT_BRACKETS >
<COMMA>
 {
 System.out.println("Found a print Statement");
 }
 }
void listStmt():
{}
{
  <LIST> < IDENTIFIER > <ASSIGNMENT> <LEFT_BRACKETS >
  ((< NUM_VAL >) < SEMICOLON >)+<RIGHT_BRACKETS >
  <COMMA>
 {
 System.out.println("Found a list Statement");
 }
  }

void arrayStmt():
{}
{
   <ARRAY> <LEFT_ANG_BRACKETS >
  (< INT_TYPE > | < BOOL_TYPE >| < FLOAT_TYPE >| < STR_TYPE >)
  < RIGHT_ANG_BRACKETS >
  < IDENTIFIER > <ASSIGNMENT> <LEFT_BRACKETS >
  ((< NUM_VAL > | < STRING_VAL >|< BOOLEAN_VAL>)< SEMICOLON >)+
  <RIGHT_BRACKETS >
  <COMMA>
 {
 System.out.println("Found an Array Statement");
 }
 }
void iterativeStmt():
{}
{
  < LOOP > < LEFT_ANG_BRACKETS > conditionalExpression()
```

```
  < RIGHT_ANG_BRACKETS > (Stmts())+ < ENDLOOP > <COMMA>
 {
 System.out.println("Found an iterative Statement");
 }
}

void conditionalStmt():
{}
{
  <IF><LEFT_ANG_BRACKETS> conditionalExpression()
  < RIGHT_ANG_BRACKETS > <THEN>
  (Stmts())+(< ELSE > (Stmts())+)? < ENDIF > <COMMA>
 {
 System.out.println("Found a conditional Statement");
 }
}

void conditionalExpression():
{}
{
       LOOKAHEAD(2) relationalStmt ()
     | LOOKAHEAD(2) logicalStmt ()
     | < BOOLEAN_VAL >
     <COMMA>
}
/* arithmetic stmt both unary and binary */
void arithmeticStmt():
{}
{
  (unaryOP ()| operand() binaryOP()) operand() <COMMA>
 {
 System.out.println("Found an Arithmetic Statement");
 }
}
void relationalStmt ():
{}
{
   operand() relationalOP() operand() <COMMA>
 {
 System.out.println("Found a Relational Statement");
 }
}

void logicalStmt ():
{}
{
   logical_operand() logicalOP() logical_operand() <COMMA>
 {
 System.out.println("Found a Logical Statement");
 }
}

void operand():
{}
{
   < IDENTIFIER > | < NUM_VAL > | <STRING_VAL>
}

void logical_operand():
```

```
{}
{
 < IDENTIFIER > | < BOOLEAN_VAL >
}
void unaryOP():
{}
{
 < INCREMENT > | < DECREMENT >
}
void binaryOP():
{}
{
 < ADD > | < SUB > | < MULTIPLY > |
 < DIVID > | < REMINDER > | < POWER >
}
void logicalOP():
{}
{
< AND >|< OR >|< NOT >
}
void relationalOP():
{}
{
   < GREATER_THAN > | < LESS_THAN > |
   < LESS_THAN_OR_EQUAL >|< GREATER_THAN_OR_EQUAL > |
   < IS_EQUAL > | <NOT_EQUAL >
}
/* data type of identifer */
void DataType () :
{}
{
   < INT_TYPE >|< BOOL_TYPE >|<FLOAT_TYPE >|< STR_TYPE >
}
```

Appendix B: JJT Grammar

```
/**
 * JJTree template file created by SF JavaCC plugin 1.5.28+ wizard for
JavaCC 1.5.0+
 * N-> NOOR & NSREEN, A-> ASMA, R-> RENAD, S-> SARA SO ( NARS )
 */
options
{
  static = true;
}

PARSER_BEGIN(NARSt)
package NARS_T;

import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class NARSt
{
  public static void main(String args [])
  {

    try {
        new NARSt(new FileInputStream("NARS_SAMPLE.txt"));
    }
    catch (FileNotFoundException e)
    {
              System.out.println("There is no available file of sample
code.");
              System.out.println(e.getMessage());
    }
       System.out.println("****** Wlecome to NARS Programming Lanaguage
******");
              System.out.println("Reading from input file...\n\n");

    while (true) {
              try {
                    System.out.println("-----------------------------------
------");
                    SimpleNode n = NARSt.Start();
                    n.dump(" >>");
                    System.out.print("Syntctically correct statements.\n");
         }// end try
         catch(Exception e)
                     {
                           System.out.println("Error encountered");
                           System.out.println(e.getMessage());
                           break;
                    }// end catch
              }// end while
  } //end main
}// end class

PARSER_END(NARSt)

SKIP :
{
  < WHIT_SPACE      : " "  >
```

```
| < AT_SIGN  : "@"  >
| < TAP                    : "\t" >
| < NEW_LINE : "\n" >
| < NEW_LINE2: "\r" >
}

TOKEN : /* OPERATORS*/
{
  < ADD : "+" >
| < SUB : "-" >
| < MULTIPLY : "*" >
| < DIVID : "/" >
| < REMINDER : "%" >
| < POWER : "^^" >
| < INCREMENT : "++" >
| < DECREMENT : "--" >
| < ASSIGNMENT : "=" >
}

TOKEN: /* Logical Operations */
{
  < AND : "&&" >
| < OR : "||" >
| < NOT : "!" >
}

TOKEN :/* Relational Operations */
{
  < IS_EQUAL :"==" >
| < NOT_EQUAL :"!=" >
| < GREATER_THAN : ".>" >
| < LESS_THAN : ".<" >
| < GREATER_THAN_OR_EQUAL :">=" >
| < LESS_THAN_OR_EQUAL :"<=" >
}

TOKEN: /*Punctuation Marks*/
{
  < TILDE_SYMBOL : "~" > | < COMMA : "," > | < DOUBLE_QUTATION : "\"" >
| < LEFT_ANG_BRACKETS : "<" > | < RIGHT_ANG_BRACKETS : ">" >
| < LEFT_BRACKETS :"(" > | < RIGHT_BRACKETS:")" > | < SEPARATOR : "|" >
| < COLON : ":" >| < SEMICOLON : ";" >
}


TOKEN: /*Keywords*/
{
  < IF              : "if" >
| < THEN            : "then" >
| < INT_TYPE : "INT" >
| < BOOL_TYPE       : "BOOL" >
| < FLOAT_TYPE      : "FLOAT" >
| < STR_TYPE : "STR" >
| < ELSE            : "else" >
| < LOOP            : "loop" >
| < NARSOUT  : "NARSout" >
| < RETURN          : "return" >
| < ARRAY           : "array" >
| < LIST            : "list" >
```

```
| < ENDIF           : "endif" >
| < ENDLOOP   : "endloop" >
| < CONSTANT : "final" >
| < EXIT            : "exit" > { System.out.println("Thank you for using
NARS! hope to see you soon."); System.exit(0); }
}

TOKEN: /*Identifiers*/
{
 < IDENTIFIER : < TILDE_SYMBOL >(<LETTER>|< DIGIT >)+ >
}

TOKEN: /*Data types */
{

   < NUM_VAL : (< DIGIT >)+ (("."(< DIGIT >)+)?) >
 | < BOOLEAN_VAL : "true"| "false" >
 | < STRING_VAL : (<LETTER >|< SYMBOL >|< DIGIT >)+ >
}

TOKEN: /*alphabets*/
{
 < LETTER :(["A"-"Z","a"-"z"])>
| < DIGIT : ["0"-"9"] >
| < SYMBOL : "#" | "$" | "?" | "_" >
}


SPECIAL_TOKEN:/*Comments*/
{
  < SINGLE_LINE: "@" (~["\n"])* "@" >
}

SimpleNode Start():{Token tt; }/*Start*/
      {
        Stmts()
            {return jjtThis; }
      | EXIT ()
            {return jjtThis; }
      }

void Stmts() : /*Stmts*/
{}
{
     declerationStmt()
   | LOOKAHEAD(2) arithmeticStmt()
   | LOOKAHEAD(3) assignmentStmt ()
   | LOOKAHEAD(2) relationalStmt ()
   | LOOKAHEAD(3) logicalStmt()
   | iterativeStmt()
   | conditionalStmt()
   | listStmt()
   | arrayStmt()
   | printStmt ()

}

void declerationStmt() :/*declerationStmt*/
{}
```

```
{
  (CONSTANT())? DataType() (IDENTIFIER()) COMMA()
}

void assignmentStmt ():/*assignmentStmt*/
{}
{
   IDENTIFIER() ASSIGNMENT()
  (STRING_VAL() | BOOLEAN_VAL()| NUM_VAL()| IDENTIFIER()) COMMA()
  }

void printStmt (): /*printStmt*/
{}
{
NARSOUT() LEFT_BRACKETS() DOUBLE_QUTATION()
(STRING_VAL())+ DOUBLE_QUTATION() RIGHT_BRACKETS() COMMA()
}

void listStmt():/*listStmt*/
{}
{
  LIST() IDENTIFIER() ASSIGNMENT() LEFT_BRACKETS()
  ((NUM_VAL()) SEMICOLON())+ RIGHT_BRACKETS()
  COMMA()
  }


void arrayStmt():/*arrayStmt*/
{}
{
   ARRAY() LEFT_ANG_BRACKETS()
  ( INT_TYPE() | BOOL_TYPE()| FLOAT_TYPE() |  STR_TYPE())
  RIGHT_ANG_BRACKETS()
  IDENTIFIER() ASSIGNMENT() LEFT_BRACKETS()
  ((NUM_VAL()| STRING_VAL()|BOOLEAN_VAL()) SEMICOLON())+
  RIGHT_BRACKETS()
  COMMA()
 }


void iterativeStmt():/*iterativeStmt*/
{}
{
  LOOP() LEFT_ANG_BRACKETS() conditionalExpression()
  RIGHT_ANG_BRACKETS() (Stmts())+ ENDLOOP() COMMA()

}

void conditionalStmt():/*conditionalStmt*/
{}
{
  IF() LEFT_ANG_BRACKETS() conditionalExpression()
  RIGHT_ANG_BRACKETS() THEN()
  (Stmts())+ (ELSE()(Stmts())+)? ENDIF() COMMA()
}

void conditionalExpression():/*conditionalExpression*/
{}
{
```

```
      LOOKAHEAD(2) relationalStmt ()
    | LOOKAHEAD(2) logicalStmt ()
    | BOOLEAN_VAL()
    COMMA()
}

/* arithmetic stmt both unary and binary */
void arithmeticStmt():/*arithmeticStmt*/
{}
{
  (unaryOP ()| operand() binaryOP()) operand() COMMA()

}

void relationalStmt ():/*relationalStmt*/
{}
{
   operand() relationalOP() operand() COMMA()
}

void logicalStmt ():/*logicalStmt*/
{}
{
   logical_operand() logicalOP() logical_operand() COMMA()
}

void operand():/*operand*/
{}
{
   IDENTIFIER()| NUM_VAL() | STRING_VAL()
}

void logical_operand():/*logical_operand*/
{}
{
 IDENTIFIER() | BOOLEAN_VAL()
}

void unaryOP():/*unaryOP*/
{}
{
 INCREMENT() | DECREMENT()
}

void binaryOP():/*binaryOP*/
{}
{
 ADD() | SUB() | MULTIPLY() |
 DIVID() | REMINDER() | POWER()
}

void logicalOP():/*logicalOP*/
{}
{
AND()|OR()|NOT()
}

void relationalOP():/*relationalOP*/
{}
```

```
{
   GREATER_THAN() | LESS_THAN()|
   LESS_THAN_OR_EQUAL()|GREATER_THAN_OR_EQUAL() |
   IS_EQUAL()| NOT_EQUAL()
}

/* data type of identifer */
void DataType () :
{}
{
   INT_TYPE() |BOOL_TYPE()|FLOAT_TYPE()|STR_TYPE()
}

/*OPERATORS*/
void ADD(): { Token tt;} {
tt= < ADD > { jjtThis.jjtSetValue(tt.image); }
}

void SUB(): { Token tt;} {
tt= < SUB > { jjtThis.jjtSetValue(tt.image); }
}

void MULTIPLY(): { Token tt;} {
tt= < MULTIPLY > { jjtThis.jjtSetValue(tt.image); }
}

void DIVID(): { Token tt;} {
tt= < DIVID > { jjtThis.jjtSetValue(tt.image); }
}

void REMINDER(): { Token tt;} {
tt= < REMINDER > { jjtThis.jjtSetValue(tt.image); }
}

void POWER(): { Token tt;} {
tt= < POWER > { jjtThis.jjtSetValue(tt.image); }
}

void INCREMENT(): { Token tt;} {
tt= < INCREMENT > { jjtThis.jjtSetValue(tt.image); }
}

void DECREMENT(): { Token tt;} {
tt= < DECREMENT > { jjtThis.jjtSetValue(tt.image); }
}

void ASSIGNMENT(): { Token tt;} {
tt= < ASSIGNMENT > { jjtThis.jjtSetValue(tt.image); }
}

/*Logical Operations*/
void AND(): { Token tt;} {
tt= < AND > { jjtThis.jjtSetValue(tt.image); }
}

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

```
void NOT(): { Token tt;} {
tt= < NOT > { jjtThis.jjtSetValue(tt.image); }
}

/*rational Operations */
void IS_EQUAL(): { Token tt;} {
tt= < IS_EQUAL > { jjtThis.jjtSetValue(tt.image); }
}
void NOT_EQUAL(): { Token tt;} {
tt= < NOT_EQUAL > { jjtThis.jjtSetValue(tt.image); }
}
void GREATER_THAN(): { Token tt;} {
tt= < GREATER_THAN > { jjtThis.jjtSetValue(tt.image); }
}
void LESS_THAN(): { Token tt;} {
tt= < LESS_THAN > { jjtThis.jjtSetValue(tt.image); }
}
void GREATER_THAN_OR_EQUAL(): { Token tt;} {
tt= < GREATER_THAN_OR_EQUAL > { jjtThis.jjtSetValue(tt.image); }
}
void LESS_THAN_OR_EQUAL(): { Token tt;} {
tt= < LESS_THAN_OR_EQUAL > { jjtThis.jjtSetValue(tt.image); }
}

/*Punctuation Marks*/

void TILDE_SYMBOL(): { Token tt;} {
tt= < TILDE_SYMBOL > { jjtThis.jjtSetValue(tt.image); }
}

void  COMMA(): { Token tt;} {
tt= <  COMMA > { jjtThis.jjtSetValue(tt.image); }
}

void  SEMICOLON(): { Token tt;} {
tt= <  SEMICOLON > { jjtThis.jjtSetValue(tt.image); }
}

void  COLON(): { Token tt;} {
tt= <  COLON > { jjtThis.jjtSetValue(tt.image); }
}

void  LEFT_BRACKETS(): { Token tt;} {
tt= <  LEFT_BRACKETS > { jjtThis.jjtSetValue(tt.image); }
}

void  RIGHT_BRACKETS(): { Token tt;} {
tt= <  RIGHT_BRACKETS > { jjtThis.jjtSetValue(tt.image); }
}
void  LEFT_ANG_BRACKETS(): { Token tt;} {
tt= <  LEFT_ANG_BRACKETS > { jjtThis.jjtSetValue(tt.image); }
}
void  RIGHT_ANG_BRACKETS (): { Token tt;} {
tt= <  RIGHT_ANG_BRACKETS  > { jjtThis.jjtSetValue(tt.image); }
}

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

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


/*Keywords*/

void IF(): { Token tt;} {
tt= < IF > { jjtThis.jjtSetValue(tt.image); }
}

void THEN(): { Token tt;} {
tt= < THEN > { jjtThis.jjtSetValue(tt.image); }
}

void INT_TYPE(): { Token tt;} {
tt= < INT_TYPE > { jjtThis.jjtSetValue(tt.image); }
}

void BOOL_TYPE(): { Token tt;} {
tt= < BOOL_TYPE > { jjtThis.jjtSetValue(tt.image); }
}

void ELSE(): { Token tt;} {
tt= < ELSE > { jjtThis.jjtSetValue(tt.image); }
}

void FLOAT_TYPE(): { Token tt;} {
tt= < FLOAT_TYPE > { jjtThis.jjtSetValue(tt.image); }
}

void STR_TYPE(): { Token tt;} {
tt= < STR_TYPE > { jjtThis.jjtSetValue(tt.image); }
}

void LOOP(): { Token tt;} {
tt= < LOOP > { jjtThis.jjtSetValue(tt.image); }
}

void NARSOUT(): { Token tt;} {
tt= < NARSOUT > { jjtThis.jjtSetValue(tt.image); }
}

void RETURN(): { Token tt;} {
tt= < RETURN > { jjtThis.jjtSetValue(tt.image); }
}

void ARRAY(): { Token tt;} {
tt= < ARRAY > { jjtThis.jjtSetValue(tt.image); }
}

void ENDIF(): { Token tt;} {
tt= < ENDIF > { jjtThis.jjtSetValue(tt.image); }
}

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

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

void LIST(): { Token tt;} {
tt= < LIST > { jjtThis.jjtSetValue(tt.image); }
}

void EXIT(): { Token tt;} {
tt= < EXIT > { jjtThis.jjtSetValue(tt.image); }
}


/*Identifiers*/
void IDENTIFIER(): { Token tt;} {
tt= < IDENTIFIER > { jjtThis.jjtSetValue(tt.image); }
}

/*Data types*/
void NUM_VAL(): { Token tt;} {
tt= < NUM_VAL > { jjtThis.jjtSetValue(tt.image); }
}
void BOOLEAN_VAL(): { Token tt;} {
tt= < BOOLEAN_VAL > { jjtThis.jjtSetValue(tt.image); }
}
void STRING_VAL(): { Token tt;} {
tt= < STRING_VAL > { jjtThis.jjtSetValue(tt.image); }
}



/*alphabets*/
void LETTER(): { Token tt;} {
tt= < LETTER > { jjtThis.jjtSetValue(tt.image); }
}
void DIGIT(): { Token tt;} {
tt= < DIGIT > { jjtThis.jjtSetValue(tt.image); }
}
void SYMBOL(): { Token tt;} {
tt= < SYMBOL > { jjtThis.jjtSetValue(tt.image); }
}
```

Appendix C: Sample Code of NARS

```
@N-> NOOR & NSREEN, A-> ASMA, R-> RENAD, S-> SARA SO ( NARS )@
STR ~name,
final INT ~x,
BOOL ~NARS,
++10,
true && true,
~flag1 .< ~flag2,
if < ~flag1 .< ~flag2, >
then
~sum= ~flag2,
endif,
if < ~flag1 .< ~flag2,>
then
~sum= ~flag2,
else
~sum= ~flag1,
endif,
loop < ~Flag .< 5, >
++~Flag ,
endloop,
@NARS@
NARSout ("Welcome To NARS Language "),
INT ~dal,
~COST = 15,
list ~C = (2;4;8;),
array <STR> ~s = (noor;nsreen;asma;renad;sara;),

exit,
```