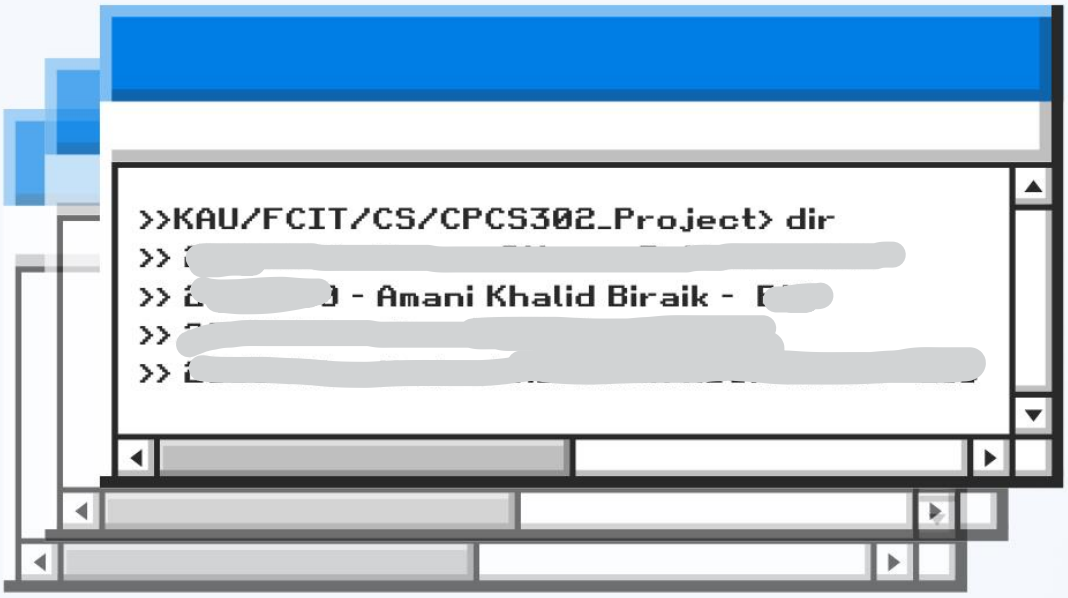
King Abdulaziz University
Faculty of Computing and Information Technology
Computer Science Department
Spring 2023
CPCS302 - Compiler Construction
Final Project
May, 27th 2023

1631 hira



Instructors:
Dr.

Table of Contents

Table of Figures

Work Distribution

Name	Work
Hasna Othman Bukhari	*All Team Members Contributed Equally
Amani Khalid Biraik	
Rima Ali Alghamdi	
Hadeel Abdullah Abuhamous	

Introduction

When it comes to constructing a language using a new idea or theme, there are a lot of things to keep in mind, whether for programmers or just users without any programming background. In the first step, the language should be simple to use and easy to understand. However, in Hausa, Hira means conversation, which describes our language. Hira is a programming language composed of several statements developed by CS20 students using the JavaCC language and the Eclipse IDE. As developers, we were considering simplicity and usability. So that our language would be more like a natural language that could be used in conversations. There are special symbols used in Hira, such as @ before identifiers, a colon (:) after some statements to present dialogue, and after every statement a dot(.) as an indicator of the end of a statement.

Phase I

I.I: Regular Expressions of the Tokens

Token Type	TOKEN NAME	Regular Expression
Comments	SINGLE_LINE_COMMENT	“cc:” (~[\n])* “.”
Words	COMMENT	“cc”
	IS	“is”
	TO	“to”
	LOOP	“loop”
	USE	“using”
	ASSIGN	“assign”
	DONE	“done”
Conditional Operations	IF_CONDITION|	“if”
	OR_CONDITION	“orIf”
	ELSE	“else”
	DO	“do:”
Relational Operations	LESS_THAN	“lt”
	GREATER_THAN	“gt”
	EQUAL	“eq”
	NOT_EQUAL	“nq”
	LESS_EQUAL	“lq”
	GREATER_EQUAL	“gq”
Arithmetic Operations	SUM	“sum”
	SUB	“sub”
	DIV	“div”
	MULT	“mult”
	REM	“rem”
Logical Operators	AND_OP	“and”
	OR_OP	“or”
	NOT_OP	“not”

Data Types	BOOLEAN	“bool”
	NUMBER	“num”
	STRING	“str”
	CONSTANT	“const”
	REGISTRY	“reg”
	STACK	“stc”
Value of Data Type	BOOL	TRUE | FALSE
	NUM	“-”? (DIGIT)+ (“.” (DIGIT)+)?
	STR	(LETTER)+ (FULL_STOP)?
	REG	NUM (COMMA (“ “)? NUM)*
	STC	STC_CONTENT (COMMA (“ “)? STC_CONTENT)*
	STC_CONTENT	NUM | STR | BOOL | REG | IDENTIFIER
White Spaces	NEWLINE	“\\n” | “\\r”
	TAB	“\\t”
Identifiers	IDENTIFIER	“@” (LETTER)+
Digit	DIGIT	[“0” – “9”]
Letter	LETTER	[“a” – “z”] | [“A” – “Z”]
Boolean values	TRUE	“true”
	FALSE	“false”
Punctuation Marks & Symbols	DOT	“.”
	AT	“@”
	COLON	“:”
	COMMA	“,”
	FULL_STOP	“\\.”
	LEFT_BRACKET	“[”
	RIGHT_BRACKET	“]”

I.II: Statements

Statement Type		Example	Code
Arithmetic Statements		Example #1: 10 + 10. Example #2: @a + @b.	(NUM | IDENTIFIER) ARITHMETIC_OP (NUM | IDENTIFIER)
Relational Statements		Example #1: @a lt @b. Example #2: 5 nq @nine.	(NUM | IDENTIFIER) RELATIONAL_OP (NUM | IDENTIFIER)
Logical Statements		Example #1: @a and @b. Example #2: not @c.	(IDENTIFIER LOGICAL_OP IDENTIFIER) | (NOT_OP IDENTIFIER)
Boolean Variable Declaration		@b is bool.	< IDENTIFIER > < IS > < BOOLEAN > < DOT >
Numeric Variable Declaration		@i is num.	< IDENTIFIER > < IS > < NUMBER > < DOT >
String Variable Declaration		@a is bool. @b is str.	< IDENTIFIER > < IS > < STRING > < DOT >
Constant Variable Declaration		@a is const str.	< IDENTIFIER > < IS > < CONSTANT > < STRING > < DOT >
Conditional Statements		if @a lt 2 do: assign 5 to @a. orif @a gq 2 do: assign -5 to @a else do: assign 1 to @a end	IF_CONDITION CONDITION DO COLON (STMTS)+ END (OR_CONDITION CONDITION DO COLON (STMTS)+ END) * (ELSE DO COLON (STMTS)+ END)?
Iterative Statement		loop @a using @b: assign 0 to @b. end.	< LOOP > < IDENTIFIER > < USE > < IDENTIFIER > < COLON > (Statements())+ < END >
Data Structures	Registry	@a is reg. assign [1, 2, 3] to @a.	< IDENTIFIER > < IS > <REGISTRY> < DOT > < ASSIGN > < REG > < TO > < IDENTIFIER > < DOT >
	Stack	@a is stc. assign [1, Hi, false] to @a.	< IDENTIFIER > < IS > < STACK > < DOT > < ASSIGN > < STC > < TO > < IDENTIFIER > < DOT >

Phase II: BNF notation

II.I BNF Notation

1. **START** → STMTS | DONE
2. **STMTS** → (STMT DOT)
3. **STMT** → DECLARATION_STATEMENT | ARITHMETIC_STATEMENT | ITERATIVE_STATEMENT | ASSIGNMENT_STATEMENT | CONDITIONAL_STATEMENT | LOGICAL_STATEMENT | RELATIONAL_STATEMENT
4. **SINGLE_LINE_COMMENT**→ COMMENT COLON (~["\n"])* DOT
5. **DECLARATION_STATEMENT** → IDENTIFIER IS (CONSTANT)? DATATYPE
6. **ARITHMETIC_STATEMENT** → (NUM | IDENTIFIER) ARITHMETIC_OP (NUM | IDENTIFIER)
7. **ITERATIVE_STATEMENT** → LOOP IDENTIFIER USE IDENTIFIER COLON (STMTS)+ END
8. **ASSIGNMENT_STATEMENT** → ASSIGN (ASSIGN_VALUE) TO IDENTIFIER
9. **CONDITIONAL_STATEMENT** →
   IF_CONDITION CONDITION DO COLON (STMTS)+ END
   (OR_CONDITION CONDITION DO COLON (STMTS)+ END) *
   (ELSE DO COLON (STMTS)+ END)?
10. **LOGICAL_STATEMENT** → (IDENTIFIER LOGICAL_OP IDENTIFIER) | (NOT_OP IDENTIFIER)
11. **RELATIONAL_STATEMENT** → (NUM | IDENTIFIER) RELATIONAL_OP (NUM | IDENTIFIER)
12. **IDENTIFIER** → AT (LETTER)+
13. **DATATYPE** → BOOLEAN | NUMBER | STRING | CONSTANT | REGISTERY | STACK
14. **ARITHMETIC_OP** → SUM | SUB | MULT | DIV | REM
15. **ASSIGN_VALUE** → STR+ | NUM | BOOL | (LEFT_BRACKET (REG | STC) RIGHT_BRACKET) | (ARITHMETIC_STATEMENT DOT) | (RELATIONAL STATEMENT DOT) | (LOGICAL_STATEMENT DOT)
16. **CONDITION** → (NUM RELATIONAL_OP (NUM | (IDENTIFIER)) | (IDENTIFIER (LOGICAL_OP IDENTIFIER | RELATIONAL_OP (NUM | IDENTIFIER))) | BOOL
17. **CONDITIONAL_OP** → IF_CONDITION | OR_CONDITION | ELSE | DO
18. **LOGICAL_OP** → AND_OP | NOT_OP | OR_OP
19. **RELATIONAL_OP** → LESS_THAN | GREATER_THAN | EQUAL | NOT_EQUAL | LESS_EQUAL | GREATER_EQUAL
20. **VALUES_OF_DATATYPE** → BOOL | NUM | STR | REG | STC
21. **BOOL** → TRUE | FALSE

22. **NUM** → (“-“)? (DIGIT)+ ((DOT (DIGIT)+) ?)
23. **STR** → (LETTER)+ (FULL_STOP)?
24. **REG** → NUM (COMMA (" ")? NUM) *
25. **STC** → STC_CONTENT (COMMA (" ")? STC_CONTENT) *
26. **SYMBOLS** → COLON | DOT | AT | LEFT_BRACKET | RIGHT_BRACKET | COMMA | FULL_STOP
27. **KEYWORDS** → COMMENT | IS | TO | LOOP | USE | ASSIGN
28. **WHITE_SPACE** → NEWLINE | TAB
29. **BOOLEAN** → “bool”
30. **NUMBER** → “num”
31. **STRING** → “str”
32. **CONSTANT** → “const”
33. **REGISTERY** → “reg”
34. **STACK** → “stc”
35. **DIGIT** → [“0” – “9”]
36. **LETTER** → ["A" – "Z"] | ["a" – "z"]
37. **TRUE** → “true”
38. **FALSE** → “false”
39. **STC_CONTENT** → (NUM | STR | BOOL | REG | IDENTIFIER)
40. **SUM** → “sum”
41. **SUB** → “sub”
42. **MULT** → “mult”
43. **DIV** → “div”
44. **REM** → “rem”
45. **LOOP** → “loop”
46. **USE** → “using”
47. **ASSIGN** → “assign”
48. **TO** → “to”
49. **IF_CONDITION** → “if”
50. **DO** → “do”
51. **OR_CONDITION** → “orIf”
52. **ELSE** → “else”
53. **LESS_THAN** → "lt"
54. **GREATER_THAN** → "gt"

55. **EQUAL** → "eq"
56. **NOT_EQUAL** → "nq"
57. **LESS_EQUAL** → "lq"
58. **GREATER_EQUAL** → "gq"
59. **COMMENT** → "cc"
60. **COLON** → ":"
61. **DOT** → "."
62. **AT** → "@"
63. **RIGHT_BRACKET** → "]"
64. **LEFT_BRACKET** → "["
65. **COMMA** → ","
66. **FULL_STOP** → "\\."
67. **IS** → "is"
68. **AND_OP** → "and"
69. **NOT_OP** → "not"
70. **OR_OP** → "or"
71. **STACK** → "stack"
72. **TO** → "to"
73. **SPACE** → " "
74. **TAB** → "\\t"
75. **NEWLINE** → \\n | "\\r"
76. **DONE** → "done"
77. **END** → "end"

II.II Commented BNF notation

/*
* HIRA language starts with statements, or the keyword “done”.
*/
START → STMTS | DONE

/**
* All language statements must end with a dot.
*/
STMTS → (STMT DOT)

/**
* There are 7 types of statements in HIRA language.
*/
STMT → DECLARATION_STATEMENT | ARITHMETIC_STATEMENT | ITERATIVE_STATEMENT | ASSIGNMENT_STATEMENT | CONDITIONAL_STATEMENT | LOGICAL_STATEMENT | RELATIONAL_STATEMENT

/**
* A single-line comment must begin with the keyword “cc” followed by a colon “:”, and end
* with a dot, it can contain a combination of letters, digits, and punctuation marks.
*/
SINGLE_LINE_COMMENT→ COMMENT COLON (~["\n"])* DOT

/**
* A declaration statement must start with an identifier followed by the keyword “is” and the data
* type. The constant is optional.
*/
DECLARATION_STATEMENT → IDENTIFIER IS (CONSTANT)? DATATYPE

/**
* An arithmetic statement must begin and end with a number or an identifier and an
* arithmetic operator in the middle.
*/
ARITHMETIC_STATEMENT → (NUM | IDENTIFIER) ARITHMETIC_OP (NUM | IDENTIFIER)

/**
* An iterative statement must begin with the keyword "loop" followed by an identifier, the
* keyword "using", another identifier, a colon, and a sequence of statements that end with the
* keyword "end".
*/
ITERATIVE_STATEMENT → LOOP IDENTIFIER USE IDENTIFIER COLON (STMTS)+ END

/**
* An assignment statement begins with the keyword “assign” followed by a certain assign value,
* the keyword “to”, and an identifier.
*/
ASSIGNMENT_STATEMENT → ASSIGN (ASSIGN_VALUE) TO IDENTIFIER
/**
* To define a conditional statement, it must start with the keyword “if” followed by the
* condition, then the keyword “do”, colon “:”, and a block of statements and end with the keyword “end”.
* The (if) statement can be followed by zero or more (or If) statements, which start with the
* keyword “orIf” followed by the condition, the keyword “do”, colon “:” and a block of
* statements and end with the keyword “end”.
* Else statement is optional, and it must start with the keyword “else” followed by the keyword
* “do”, colon “:”, and a block of statements and end with the keyword “end”.
*/
CONDITIONAL_STATEMENT →
IF_CONDITION CONDITION DO COLON (STMTS)+ END
(OR_CONDITION CONDITION DO COLON (STMTS)+ END) *
(ELSE DO COLON (STMTS)+ END)?

/**
* A logical statement can be written in two ways:
* 1. Begin with an identifier followed by a logical operator and another identifier.
* 2. Begin with the “not” logical operator followed by an identifier.
*/
LOGICAL_STATEMENT → (IDENTIFIER LOGICAL_OP IDENTIFIER) | (NOT_OP IDENTIFIER)

/**
* A relational statement must begin and end with a number or an identifier, and a
* relational operator in the middle.
*/
RELATIONAL_STATEMENT → (NUM | IDENTIFIER) RELATIONAL_OP (NUM | IDENTIFIER)

/**
* An identifier must start with the “@” symbol followed by one or more letters.
*/
IDENTIFIER → AT (LETTER)+

/**
* A data type can be boolean, number, string, constant, registry, or stack.
*/
DATATYPE → BOOLEAN | NUMBER | STRING | CONSTANT | REGISTERY | STACK

/**
* There are 5 arithmetic operators: summation, subtraction, multiplication, division, or
* remainder.
*/
ARITHMETIC_OP → SUM | SUB | MULT | DIV | REM

/**
* The assigned value can be: a string, number, Boolean value (true or false), an assigned value
* for a registry or a stack ex: [1, 2, 3], arithmetic statement ex: 2 + 2., relational statement
* ex: 2 eq 2, or logical statement ex: @a and @b
*/
ASSIGN_VALUE → STR+ | NUM | BOOL | (LEFT_BRACKET (REG | STC) RIGHT_BRACKET) | (ARITHMETIC_STATEMENT DOT) | (RELATIONAL STATEMENT DOT) | (LOGICAL_STATEMENT DOT)

/**
* Hira`s Conditions can 1. start with Numbers and end with numbers or identifiers, ei.1 lt 2 or
* 2. Start with and identifiers followed by a logical operator and an identifier, ei, @a and @b
* or 3. Starts with an identifier followed by a relational operator and a number or an identifier ei,
* @a eq 3 or 4. A boolean value such as true or false
*/
CONDITION → (NUM RELATIONAL_OP (NUM | IDENTIFIER))
| (IDENTIFIER (LOGICAL_OP IDENTIFIER | RELATIONAL_OP (NUM | IDENTIFIER)))
| BOOL

/**
* Hira`s Conditional operators consist of “if”, “orif”, “else”, and “do”
*/
CONDITIONAL_OP → IF_CONDITION | OR_CONDITION | ELSE | DO

/**
* Hira`s logical operators consist of “and”, “not”, and “or”
*/
LOGICAL_OP → AND_OP | NOT_OP | OR_OP

/**
* Hira`s relational operations consist of “lt”, “gt”, “eq”, “nq”, “lq”, and “gq”
*/
RELATIONAL_OP → LESS_THAN | GREATER_THAN | EQUAL | NOT_EQUAL | LESS_EQUAL | GREATER_EQUAL

/**
* Hira`s data types can have the following values: boolean, number, string, registry values, or stack values
*/
VALUES_OF_DATATYPE → BOOL | NUM | STR | REG | STC

/**
* Hira`s boolean value can be either true or false
*/
BOOL → TRUE | FALSE

/**
* Hira`s numeric values can start with a negative sign “-“, followed by one or more digit
* the number can optionally end with a dot followed by a digit or more ei, -1, -1.2, and 4.
*/
NUM → (“-“)? (DIGIT)+ ((DOT (DIGIT)+) ?)

/**
* Hira`s string values are all letters, and can end with \\. Representing the full stop
*/
STR → (LETTER)+ (FULL_STOP)?

/**
* Hira`s registry value is one or more numbers separated by a comma ei, 3, 4, 5
*/
REG → NUM (COMMA (" ")? NUM) *

/**
* Hira`s stack value is one ore more stack content separated by a comma
*/
STC → STC_CONTENT (COMMA (" ")? STC_CONTENT) *

/**
* Hira's stack content can be any of numbers, strings, boolean, registry, or identifiers
*/
STC_CONTENT → (NUM | STR | BOOL | REG | IDENTIFIER)

/**
* Hira`s symbols are ":", ".", "@", "[", "]", ",", "."
*/
SYMBOLS → COLON | DOT | AT | LEFT_BRACKET | RIGHT_BRACKET | COMMA | FULL_STOP

/**
* Hira`s white spaces are the following ""\\n"" , "\\r" as a new line, and \\t as a tab
*/
WHITE_SPACE → NEWLINE | TAB

/**
* Hira`s alphabetical digits can be any single digit from 0 to 9
*/
DIGIT → ["0" – "9"]

/**
* Hira`s alphabetical letters can be any single letter, as an uppercase or lowercase.
*/
LETTER → ["A" – "Z"] | ["a" – "z"]

/**
* Hira`s Keywords
*/
KEYWORDS → COMMENT | IS | TO | LOOP | USE | ASSIGN
TRUE → "true"
FALSE → "false"
BOOLEAN → "bool"
NUMBER → "num"
STRING → "str"

CONSTANT → “const”
REGISTERY → “reg”
STACK → “stc”
SUM → “sum”
SUB → “sub”
MULT → “mult”
DIV → “div”
REM → “rem”
LOOP → “loop”
USE → “using”
ASSIGN → “assign”
TO → “to”
IF_CONDITION → “if”
DO → “do”
OR_CONDITION → “orIf”
ELSE → “else”
LESS_THAN → "lt"
GREATER_THAN → "gt"
EQUAL → "eq"
NOT_EQUAL → "nq"
LESS_EQUAL → "lq"
GREATER_EQUAL → "gq"
COMMENT → "cc"
COLON → “:”
DOT → “.”
AT → “@”
RIGHT_BRACKET → “]”
LEFT_BRACKET → “[”
COMMA → “,”
FULL_STOP → "\\."
IS → “is”
AND_OP → “and”
NOT_OP → “not”
OR_OP → “or”
STACK → “stack”
TO → “to”
TAB → "\\t"
NEWLINE → \\n | “\\r”
DONE → “done”
END → “end”

Phasse lll

III.I jj File Screen shots examples

```
                ~~ Welcome to HIRA ~~
  To write a comment, start your comment with "cc:" and end with "."
  Make sure to end each statement with "."

Enter your line: cc: HI
Error!! Syntactically not correct statement!

Encountered " "cc" "cc "" at line 1, column 1.
Was expecting one of:
    "done" ...
    <IDENTIFIER> ...
    "not" ...
    "if" ...
    "loop" ...
    "assign" ...
    <NUM> ...
```

```
                ~~ Welcome to HIRA ~~
  To write a comment, start your comment with "cc:" and end with "."
  Make sure to end each statement with "."

Enter your line: cc: Arithmetic_Statements example:.
@a sum @b.
Syntactically correct statement.

Enter your line: 3 mult @c.
Syntactically correct statement.

Enter your line: cc: Relational_Statements example:.
@a lt @b.
Syntactically correct statement.

Enter your line: 5 nq 10.
Syntactically correct statement.

Enter your line: cc: Logical_Statements example:.
@a and @b.
 and is a logical operator
Syntactically correct statement.

Enter your line: not @c.
 not is a logical operator
Syntactically correct statement.

Enter your line: cc: Boolean_Statements example:.
@a is bool.
Syntactically correct statement.

Enter your line: assign true to @a.
Syntactically correct statement.

Enter your line:

Enter your line: cc: Conditional_Statements example:.
if false do: assign 5 to @a. end orIf true do: assign -5 to @a. @s sum 4. end else do: assign 0 to @a. end.
Syntactically correct statement.

Enter your line: cc: Conditional_Statement example:.
if 10 eq @id do: assign 20 to @id. end orIf 10 lt @id do: @w is num. assign 10 to @w. end.
Syntactically correct statement.
Enter your line: cc: Iterative_Statements example:.
loop @a using @b:  assign 0 to @b. end.
Syntactically correct statement.

Enter your line: cc: Variable_Declaration example:.
@a is bool.
Syntactically correct statement.

Enter your line: @b is str.
Syntactically correct statement.

Enter your line: cc: Constant_Variable_Declaration example:.
@a is const str.
Syntactically correct statement.

Enter your line: cc: Data_Structures example:.
@a is reg.
Syntactically correct statement.

Enter your line: assign [1, 2, 3] to @a.
Syntactically correct statement.

Enter your line: @a is stc.
Syntactically correct statement.

Enter your line: assign [1, Hi, false] to @a.
Syntactically correct statement.

Enter your line: done.
Good Bye ~~
```

Figure 1:jj File - Screenshot

III.II jjt File Screen shots examples

```
              ~~ Welcome to HIRA ~~
  To write a comment, start your comment with "cc:" and end with "."
  Make sure to end each statement with "."

Enter your line: cc: Conditional_Statement example:.
if 10 eq @id do: assign 20 to @id. end orIf 10 lt @id do: @w is num. assign 10 to @w. end.
>Statements
> Statement
>  Conditional_Statement
>   IF_CONDITION : if
>   Condition
>    NUM : 10
>    Relational_Op
>     EQUAL : eq
>    IDENTIFIER : @id
>   DO : do
>   COLON : :
>   Statements
>    Statement
>     Assignment_Statement
>      ASSIGN : assign
>      Assign_Value
>       NUM : 20
>      TO : to
>      IDENTIFIER : @id
>    DOT : .
>   END : end
>   OR_CONDITION : orIf
>   Condition
>    NUM : 10
>    Relational_Op
>     LESS_THAN : lt
>    IDENTIFIER : @id
>   DO : do
>   COLON : :
>   Statements
>    Statement
>     Declaration_Statement
>      IDENTIFIER : @w
>      IS : is
>      Data_Type
>       NUMBER : num
>    DOT : .
>   Statements
>    Statement
>     Assignment_Statement
>      ASSIGN : assign
>      Assign_Value
>       NUM : 10
>      TO : to
>      IDENTIFIER : @w
>    DOT : .
>   END : end
> DOT : .
 correct statement.

Enter your line: cc: Iterative_Statement example:.
loop @reg using @index: assign -1 to @index. end.
>Statements
> Statement
>  Iterative_Statement
>   LOOP : loop
>   IDENTIFIER : @reg
>   USE : using
>   IDENTIFIER : @index
>   COLON : :
>   Statements
>    Statement
>     Assignment_Statement
>      ASSIGN : assign
>      Assign_Value
>       NUM : -1
>      TO : to
>      IDENTIFIER : @index
>    DOT : .
>   END : end
> DOT : .
 correct statement.

Enter your line: done
Good Bye ~~
```

```
              ~~ Welcome to HIRA ~~
  To write a comment, start your comment with "cc:" and end with "."
  Make sure to end each statement with "."

Enter your line: cc: Arithmetic_Statement example:.
5 mult @id.
>Statements
> Statement
>  Arithmetic_Statement
>   NUM : 5
>   Arthmetic_Op
>    MULT : mult
>   IDENTIFIER : @id
> DOT : .
 correct statement.

Enter your line: cc: Relational_Statement example:.
@minute gt 50.
>Statements
> Statement
>  Relational_Statement
>   IDENTIFIER : @minute
>   Relational_Op
>    GREATER_THAN : gt
>   NUM : 50
> DOT : .
 correct statement.

Enter your line: cc: Declaration_Statement example:.
@a is const reg.
>Statements
> Statement
>  Declaration_Statement
>   IDENTIFIER : @a
>   IS : is
>   CONSTANT : const
>   Data_Type
>    REGISTRY : reg
> DOT : .
 correct statement.

Enter your line: cc: Assignment_Statemet example:.
assign [1, 2, 3, 4] to @a.
>Statements
> Statement
>  Assignment_Statement
>   ASSIGN : assign
>   Assign_Value
>    LEFT_BRACKET : [
>    REG : 1, 2, 3, 4
>    RIGHT_BRACKET : ]
>   TO : to
>   IDENTIFIER : @a
> DOT : .
 correct statement.

Enter your line: cc: Conditional_Statement example:.
if 3 lt @id do: assign 3 to @id. end.
>Statements
> Statement
>  Conditional_Statement
>   IF_CONDITION : if
>   Condition
>    NUM : 3
>    Relational_Op
>     LESS_THAN : lt
>    IDENTIFIER : @id
>   DO : do
>   COLON : :
>   Statements
>    Statement
>     Assignment_Statement
>      ASSIGN : assign
>      Assign_Value
>       NUM : 3
>      TO : to
>      IDENTIFIER : @id
>    DOT : .
>   END : end
> DOT : .
 correct statement.

Enter your line: cc: HI
incorrect statement!

Encountered " "cc" "cc "" at line 1, column 1.
Was expecting one of:
    "done" ...
    <IDENTIFIER> ...
    "not" ...
    "if" ...
    "loop" ...
    "assign" ...
    <NUM> ...
```

Figure 2:jjt File - Screenshot

Appendix A

jj Code

```
/**
 * CPCS 302 group project --
 */
options
{
  static = true;
}

PARSER_BEGIN(MyNewGrammar)
package Project;

public class MyNewGrammar
{
  public static void main(String args []) throws ParseException
  {
    MyNewGrammar parser = new MyNewGrammar(System.in);

       System.out.println("\t\t~~ Welcome to HIRA ~~\n" +
      "  To write a comment, start your comment with \"cc:\" and end with \".\"\n" +
      "  Make sure to end each statement with \".\"\n");

    while (true)
    {
      System.out.print("Enter your line: ");
      try
      {
        MyNewGrammar.Statements();
        System.out.println("Syntactically correct statement.\n");
      }
      catch (Exception e)
      {
        System.out.println("Error!! Syntactically not correct statement!\n");
        System.out.println(e.getMessage());
        System.exit(1);
      }
    }
  }
}

PARSER_END(MyNewGrammar)

SKIP :
{
  " "
| "\n"
| "\r" }
```

```
/**
* Hira's white spaces are the following "\\n" or "\\r" as a new line, and "\\t" as a
tab
*/

TOKEN : /* White Space */
{
  < NEWLINE : "\\n" | "\\r" > { System.out.println();      }
| < TAB     : "\\t"         > { System.out.print("     "); } }

/**
* A single-line comment must begin with the keyword "cc" followed by a colon ":", and
end
* with a dot, it can contain a combination of letters, digits, and punctuation marks.
*/

SPECIAL_TOKEN : /* Comment */
{
  < SINGLE_LINE_COMMENT : < COMMENT > < COLON > (~["\n"])* < DOT > > }

/**
* Hira's symbols are ":", ".", "@", "[", "]", ",", "."
*/

TOKEN : /* Symbols */
{
  < DOT           : "."    >
| < AT            : "@"    >
| < COLON         : ":"    >
| < COMMA         : ","    >
| < FULL_STOP     : "\\."  >
| < LEFT_BRACKET  : "["    >
| < RIGHT_BRACKET : "]"    > }

/**
* An identifier must start with the "@" symbol followed by one or more letters.
*/

TOKEN : /* Identifiers */
{
  < IDENTIFIER : < AT > (< LETTER >)+ > }

/**
* There are 5 arithmetic operators: summation, subtraction, multiplication, division,
or
* remainder.
*/

TOKEN : /* Arithmetic Operations */
{
  < SUM    : "sum"    >
| < SUB    : "sub"    >
| < MULT   : "mult"   >
| < DIV    : "div"    >
| < REM    : "rem"    > }
```

```
/**
* Hira's relational operations consist of "less than", "greater than", "equal", "not
equal"
* "less or equal", and "greater or equal".
*/

TOKEN : /* Relational Operations */
{
  < LESS_THAN     : "lt" >
| < GREATER_THAN  : "gt" >
| < EQUAL         : "eq" >
| < NOT_EQUAL     : "nq" >
| < LESS_EQUAL    : "lq" >
| < GREATER_EQUAL : "gq" > }

/**
* Hira's logical operators consist of "and", "not", and "or".
*/

TOKEN : /* Logical Operators */
{
  < AND_OP : "and" >  { System.out.println(" and is a logical operator");  }
| < OR_OP  : "or"  >  { System.out.println(" or is a logical operator");   }
| < NOT_OP : "not" >  { System.out.println(" not is a logical operator "); } }

/**
* Hira's Conditional operators consist of "if", "orIf", "else", and "do".
*/
TOKEN : /* If Statements */
{
  < IF_CONDITION : "if"   >
| < DO           : "do"   >
| < OR_CONDITION : "orIf" >
| < ELSE         : "else" > }

/**
* Hira's Keywords
*/
TOKEN : /* Keywords */
{
  < COMMENT : "cc"     >
| < IS      : "is"     >
| < TO      : "to"     >
| < LOOP    : "loop"   >
| < END     : "end"    >
| < USE     : "using"  >
| < ASSIGN  : "assign" >
| < DONE    : "done"   > { System.out.println("Good Bye ~~"); System.exit(0); } }
/**
* A data type can be boolean, number, string, constant, registry, or stack.
*/
TOKEN : /* Data Types */
{
  < BOOLEAN  : "bool"  >
| < NUMBER   : "num"   >
| < STRING   : "str"   >
| < CONSTANT : "const" >
| < REGISTRY : "reg"   >
| < STACK    : "stc"   > }
```

```
/**
* Hira's data types can have the following values: boolean, number, string, registry
values, or stack values
*/

TOKEN : /* VALUES OF DATA TYPES */
{
/**
* Hira's boolean value can be either true or false
*/
  < BOOL    : (< TRUE > | < FALSE >) >

/**
* Hira's numeric values can start with a negative sign "-", followed by one or more
digit
* the number can optionally end with a dot followed by a digit or more ei, -1, -1.2,
and 4.
*/
| < NUM     : ("-")? (< DIGIT >)+ ((< DOT >(< DIGIT >)+)?) >

/**
* Hira's string values are all letters, and can end with \. Representing the full
stop
*/
| < STR     : (< LETTER >)+ (< FULL_STOP >)? >

/**
* Hira's registry value is one or more numbers separated by a comma ei, 3, 4, 5
*/
| < REG     :  < NUM > (< COMMA > (" ")? < NUM >)* >

/**
* Hira's stack value is one ore more stack content separated by a comma
*/
| < STC     :  < STC_CONTENT > (< COMMA > (" ")? < STC_CONTENT >)* >

/**
* Hira's alphabetical letters can be any single letter, as an uppercase or lowercase.
*/
       /* LETTERS AND DIGITS */
| < #LETTER : ["a"-"z"] | ["A"-"Z"] >

/**
* Hira's alphabetical digits can be any single digit from 0 to 9
*/
| < #DIGIT  : ["0"-"9"]
>

       /* BOOLEAN VALUES */
| < #TRUE   : "true">
| < #FALSE  : "false" >
```

```
/**
* Hira's stack content can be any of numbers, strings, boolean, registry, or
identifiers
*/
      /* CONTENT OF STACK */
| < #STC_CONTENT : (< NUM > | < STR > | < BOOL > | < REG > | < IDENTIFIER >) > }

//-----------------------------------
//---------------BNF-----------------
//-----------------------------------
/**
* HIRA language starts with statements, single-line comment, or the keyword "done".
*/
void Start() :
{}
{
 Statements() | < DONE > }
/**
* All language statements must end with a dot.
*/
void Statements() :
{}
{
 (Statement() < DOT >) }
/**
* There are 7 types of statements in HIRA language.
*/

void Statement() :
{}
{
  LOOKAHEAD(3)Arithmetic_Statement()
| LOOKAHEAD(2)Relational_Statement()
| LOOKAHEAD(3)Logical_Statement()
| Assignment_Statement()
| Conditional_Statement()
| Declaration_Statement()
| Iterative_Statement()
| < NUM > }

/**
* An arithmetic statement must begin and end with a number or an identifier and an
* arithmetic operator in the middle.
*/

void Arithmetic_Statement() :
{}
{
  ( < NUM > | < IDENTIFIER > ) Arthmetic_Op() ( < NUM > | < IDENTIFIER > ) {
System.out.println("Found arithmetic statement."); } }

void Arthmetic_Op():
{}
{
  < SUM  >
| < SUB  >
| < MULT >
| < DIV  >
| < REM  > }
```

```
/**
* A relational statement must begin and end with a number or an identifier, and a
* relational operator in the middle.
*/

void Relational_Statement() :
{}
{
  ( < NUM > | < IDENTIFIER > ) Relational_Op() ( < NUM > | < IDENTIFIER > ) {
System.out.println("Found relational statement."); } }

void Relational_Op() :
{}
{
  < LESS_THAN     >
| < GREATER_THAN  >
| < EQUAL         >
| < NOT_EQUAL     >
| < LESS_EQUAL    >
| < GREATER_EQUAL > }

/**
* A logical statement can be written in two ways:
* 1. Begin with an identifier followed by a logical operator and another identifier.
* 2. Begin with the "not" logical operator followed by an identifier.
*/
void Logical_Statement() :
{}
{
  < IDENTIFIER > Logical_Op() < IDENTIFIER > { System.out.println("Found logical
statement."); }
| < NOT_OP > < IDENTIFIER > { System.out.println("Found logical statement."); } }

void Logical_Op() :
{}
{
  < AND_OP >
| < OR_OP  > }
/**
* A declaration statement must start with an identifier followed by the keyword "is"
and the data
* type. The constant is optional.
*/
void Declaration_Statement() :
{}
{
   < IDENTIFIER > < IS > (< CONSTANT >)? Data_Type() { System.out.println("Found
declaration statement."); } }

void Data_Type() :
{}
{
  < BOOLEAN  >
| < NUMBER   >
| < STRING   >
| < REGISTRY >
| < STACK    > }
```

```
/**
* An assignment statement begins with the keyword "assign" followed by a certain
assign value,
* the keyword “to”, and an identifier.
*/

void Assignment_Statement() :
{}
{
  < ASSIGN > (Assign_Value()) < TO > < IDENTIFIER > { System.out.println("Found
assignment statement."); } }

/**
* The assigned value can be: a string, number, Boolean value (true or false), an
assigned value
* for a registry or a stack ex: [1, 2, 3], arithmetic statement ex: 2 + 2.,
relational statement
* ex: 2 eq 2, or logical statement ex: @a and @b.
*/

void Assign_Value():
{}
{
  (< STR >)+
| < NUM  > ((Arthmetic_Op() | Relational_Op()) < NUM > )?
| < IDENTIFIER > ( Arthmetic_Op() | Relational_Op() | Logical_Op()) < IDENTIFIER >
| < NOT_OP > < IDENTIFIER >
| < BOOL >
| < LEFT_BRACKET > (< REG > | < STC >) < RIGHT_BRACKET >
}


/**
* To define a conditional statement, it must start with the keyword “if” followed by
the
* condition, then the keyword “do”, colon “:”, and a block of statements and end with
the keyword “end”.
* The (if) statement can be followed by zero or more (or If) statements, which start
with the
* keyword “orIf” followed by the condition, the keyword “do”, colon “:” and a block
of
* statements and end with the keyword “end”.
* Else statement is optional, and it must start with the keyword “else” followed by
the keyword
* “do”, colon “:”, and a block of statements and end with the keyword “end”.
*/
void Conditional_Statement():
{}
{
  < IF_CONDITION > Condition() < DO > < COLON > (Statements())+ < END > {
System.out.println("Found if statement."); }
 (< OR_CONDITION > Condition() < DO > < COLON > (Statements())+ < END > {
System.out.println("Found orIf statement."); })*
 (< ELSE > < DO > < COLON > (Statements())+ < END > { System.out.println("Found else
statement."); })? }
```

```
/**
* Hira`s Conditions can 1. start with Numbers and end with numbers or identifiers,
ei.1 lt 2 or
* 2. Start with and identifiers followed by a logical operator and an identifier, ei,
@a and @b
* or 3. Starts with an identifier followed by a relational operator and a number or
an identifier ei,
* @a eq 3 or 4. A boolean value such as true or false
*/

void Condition():
{}
{
  (< NUM > Relational_Op() (< NUM > | < IDENTIFIER >))
| (< IDENTIFIER > (Logical_Op() < IDENTIFIER > | Relational_Op() (< NUM > | <
IDENTIFIER >)))
|  < BOOL > }

/**
* An iterative statement must begin with the keyword "loop" followed by an
identifier, the
* keyword "using", another identifier, a colon, and a sequence of statements that end
with the
* keyword "end".
*/

void Iterative_Statement():
{}
{
  < LOOP > < IDENTIFIER > < USE > < IDENTIFIER > < COLON > (Statements())+ < END > {
System.out.println("Found iterative statement."); } }
```

Appendix B

jjt Code

```
/**
 * CPCS 302 group project
 */

options
{
  static = true;
}

PARSER_BEGIN(MyNewGrammar)
package TREE;

public class MyNewGrammar
{
  public static void main(String args [])
  {
     MyNewGrammar parser = new MyNewGrammar(System.in);

      System.out.println("\t\t~~ Welcome to HIRA ~~\n" +
      "  To write a comment, start your comment with \"cc:\" and end with \".\"\n" +
      "  Make sure to end each statement with \".\"\n");
    while (true)
    {
      System.out.print("Enter your line: ");

      try
      {
       SimpleNode n = MyNewGrammar.Start();
        n.dump(">");

        System.out.println(" correct statement.\n");
      }
      catch (Exception e)
      {
        System.out.println("incorrect statement!\n");
        System.out.println(e.getMessage());
        System.exit(1);
      }
    }
  }
}

PARSER_END(MyNewGrammar)

SKIP :
{
  " "
| "\n"
| "\r"
}
```

```
/**
* Hira's white spaces are the following "\\n" or "\\r" as a new line, and "\\t" as a
tab
SPECIAL_TOKEN : /* White Space */
{
  < NEWLINE : "\\n" | "\\r" > { System.out.println();    }
| < TAB     : "\\t"         > { System.out.print("     "); } }


/**
* A single-line comment must begin with the keyword "cc" followed by a colon ":", and
end
* with a dot, it can contain a combination of letters, digits, and punctuation marks.
*/
SPECIAL_TOKEN : /* Comment */
{
  < SINGLE_LINE_COMMENT : < COMMENT > < COLON > (~["\n"])* < DOT > > }

/**
* Hira's symbols are ":", ".", "@", "[", "]", ",", "."
*/
TOKEN : /* Symbols */
{
  < DOT           : "."    >
| < AT            : "@"    >
| < COLON         : ":"    >
| < COMMA         : ","    >
| < FULL_STOP     : "\\."  >
| < LEFT_BRACKET  : "["    >
| < RIGHT_BRACKET : "]"    >
| < DONE          : "done" > { System.out.println("Good Bye ~~"); System.exit(0); } }

/**
* An identifier must start with the "@" symbol followed by one or more letters.
*/
TOKEN : /* Identifiers */
{
  < IDENTIFIER : < AT > (< LETTER >)+ > }

/**
* There are 5 arithmetic operators: summation, subtraction, multiplication, division,
or
* remainder.
*/
TOKEN : /* Arithmetic Operations */
{
  < SUM    : "sum"    >
| < SUB    : "sub"    >
| < MULT   : "mult"   >
| < DIV    : "div"    >
| < REM    : "rem"    > }
```

```
/**
* Hira's relational operations consist of "less than", "greater than", "equal", "not
equal"
* "less or equal", and "greater or equal".
*/
TOKEN : /* Relational Operations */
{
  < LESS_THAN     : "lt" >
| < GREATER_THAN  : "gt" >
| < EQUAL         : "eq" >
| < NOT_EQUAL     : "nq" >
| < LESS_EQUAL    : "lq" >
| < GREATER_EQUAL : "gq" > }

/**
* Hira's logical operators consist of "and", "not", and "or".
*/
TOKEN : /* Logical Operators */
{
  < AND_OP : "and" >  { System.out.println(" and is a logical operator");  }
| < OR_OP  : "or"  >  { System.out.println(" or is a logical operator");   }
| < NOT_OP : "not" >  { System.out.println(" not is a logical operator "); } }

/**
* Hira's Conditional operators consist of "if", "orIf", "else", and "do".
*/
TOKEN : /* If Statements */
{
  < IF_CONDITION : "if"   >
| < DO           : "do"   >
| < OR_CONDITION : "orIf" >
| < ELSE         : "else" > }

/**
* Hira's Keywords
*/
TOKEN : /* Keywords */
{
  < COMMENT : "cc" >
| < IS      : "is"         >
| < TO      : "to"         >
| < LOOP    : "loop"       >
| < END     : "end"        >
| < USE     : "using"      >
| < ASSIGN  : "assign"     > }

/**
* A data type can be boolean, number, string, constant, registry, or stack.
*/
TOKEN : /* Data Types */
{
  < BOOLEAN  : "bool"  >
| < NUMBER   : "num"   >
| < STRING   : "str"   >
| < CONSTANT : "const" >
| < REGISTRY : "reg"   >
| < STACK    : "stc"   > }
```

```
/**
* Hira's data types can have the following values: boolean, number, string, registry
values, or stack values
*/
TOKEN : /* VALUES OF DATA TYPES */
{

/**
* Hira's boolean value can be either true or false
*/
  < BOOL          : (< TRUE > | < FALSE >) >
/**
* Hira's numeric values can start with a negative sign "-", followed by one or more
digit
* the number can optionally end with a dot followed by a digit or more ei, -1, -1.2,
and 4.
*/

| < NUM           : ("-")? (< DIGIT >)+ ((< DOT >(< DIGIT >)+)?)  >
/**
* Hira's string values are all letters, and can end with \. Representing the full
stop
*/
| < STR           : (< LETTER >)+ (< FULL_STOP >)? >
/**
* Hira's registry value is one or more numbers separated by a comma ei, 3, 4, 5
*/
| < REG           :  < NUM > (< COMMA > (" ")? < NUM >)* >
/**
* Hira's stack value is one ore more stack content separated by a comma
*/
| < STC           :  < STC_CONTENT > (< COMMA > (" ")? < STC_CONTENT >)* >
/**
* Hira's alphabetical letters can be any single letter, as an uppercase or lowercase.
*/
       /* LETTERS AND DIGITS */
| < #LETTER       : ["a"-"z"] | ["A"-"Z"] >

/**
* Hira's alphabetical digits can be any single digit from 0 to 9
*/
| < #DIGIT        : ["0"-"9"]
>
       /* BOOLEAN VALUES */
| < #TRUE         : "true" >
| < #FALSE        : "false" >
/**
* Hira's stack content can be any of numbers, strings, boolean, registry, or
identifiers
*/
      /* CONTENT OF STACK */
| < #STC_CONTENT : (< NUM > | < STR > | < BOOL > | < REG > | < IDENTIFIER >) > }
```

```
/*
* HIRA language starts with statements, or the keyword “done”.
*/
SimpleNode Start() :
{}
{
 Statements()
{
   return jjtThis;
 }
 | DONE()
 {
   return jjtThis;
 } }
/**
* One or more statements can be written. All language statements must end with a dot.
*/
void Statements() :
{}
{
 (Statement() DOT()) }
// DOTis one of Hira`s symbols
void  DOT():
{Token t;}
{ t = < DOT > {jjtThis.jjtSetValue(t.image);} }
// DONE is one of Hira`s symbols
void  DONE():
{Token t;}
{ t = < DONE > {jjtThis.jjtSetValue(t.image);} }
/**
* There are 7 types of statements in HIRA language.
*/
void Statement() :
{Token t;}
{
  LOOKAHEAD(3)Arithmetic_Statement()
| LOOKAHEAD(2)Relational_Statement()
| LOOKAHEAD(3)Logical_Statement()
| Assignment_Statement()
| Conditional_Statement()
| Declaration_Statement()
| Iterative_Statement()
| NUM() }
```

```
/**
* An arithmetic statement must begin and end with a number or an identifier and an
* arithmetic operator in the middle.
*/
void Arithmetic_Statement() :
{}
{
  ( NUM() | IDENTIFIER() ) Arthmetic_Op() (  NUM()  | IDENTIFIER() ) }

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

void Arthmetic_Op():
{}
{
   SUM()
|  SUB()
|  MULT()
|  DIV()
|  REM() }
void SUM():
{Token t;}
{ t = < SUM > {jjtThis.jjtSetValue(t.image);} }
void SUB():
{Token t;}
{ t = < SUB > {jjtThis.jjtSetValue(t.image);} }
void MULT():
{Token t;}
{t = < MULT > {jjtThis.jjtSetValue(t.image);} }
void DIV():
{Token t;}
{t = < DIV > {jjtThis.jjtSetValue(t.image);} }
void REM():
{Token t;}
{ t = < REM > {jjtThis.jjtSetValue(t.image);} }

//----------------------------------
/**
* A relational statement must begin and end with a number or an identifier, and a
* relational operator in the middle.
*/
void Relational_Statement() :
{}
{
  ( NUM() | IDENTIFIER() ) Relational_Op() ( NUM() | IDENTIFIER() ) }
```

```
/**
* Hira's relational operations consist of "less than", "greater than", "equal", "not
equal"
* "less or equal", and "greater or equal".
*/
void Relational_Op() :
{}
{
   LESS_THAN()
|  GREATER_THAN()
|  EQUAL()
|  NOT_EQUAL()
|  LESS_EQUAL()
|  GREATER_EQUAL() }

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

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

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

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

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

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

/**
* A logical statement can be written in two ways:
* 1. Begin with an identifier followed by a logical operator and another identifier.
* 2. Begin with the "not" logical operator followed by an identifier.
*/
void Logical_Statement() :
{}
{
  IDENTIFIER() Logical_Op() IDENTIFIER()
|  NOT_OP() IDENTIFIER() }
```

```
/**
* Hira's logical operators consist of "and", "not", and "or".
*/
void Logical_Op() :
{}
{
   AND_OP()
|  OR_OP() }
void AND_OP() :
{Token t;}
{ t = < AND_OP > {jjtThis.jjtSetValue(t.image);} }
void OR_OP() :
{Token t;}
{ t = < OR_OP  > {jjtThis.jjtSetValue(t.image);} }
void NOT_OP():
{Token t;}
{ t = < NOT_OP > {jjtThis.jjtSetValue(t.image);} }
/**
* A declaration statement must start with an identifier followed by the keyword "is"
and the data
* type. The constant is optional.
*/
void Declaration_Statement() :
{}
{
   IDENTIFIER() IS() (CONSTANT())? Data_Type() }
void IS():
{Token t;}
{ t = < IS > {jjtThis.jjtSetValue(t.image);} }
// CONSTANT is one of Hira`s Keuwords
void CONSTANT():
{Token t;}
{ t = < CONSTANT > {jjtThis.jjtSetValue(t.image);} }
/**
* A data type can be boolean, number, string, constant, registry, or stack.
*/
void Data_Type() :
{}
{
   BOOLEAN()
|  NUMBER()
|  STRING()
|  REGISTRY()
|  STACK() }
// BOOLEAN is one of Hira`s Keuwords
void BOOLEAN() :
{Token t;}
{ t =  < BOOLEAN  > {jjtThis.jjtSetValue(t.image);} }
// NUMNBER is one of Hira`s Keuwords
void NUMBER() :
{Token t;}
{ t = < NUMBER   > {jjtThis.jjtSetValue(t.image);} }
// STRING is one of Hira`s Keuwords
void STRING() :
{Token t;}
{ t = < STRING > {jjtThis.jjtSetValue(t.image);} }
```

```
// REGISTRY is one of Hira`s Keuwords
void REGISTRY() :
{Token t;}
{ t = < REGISTRY > {jjtThis.jjtSetValue(t.image);} }

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

//----------------------------------
/**
* An assignment statement begins with the keyword "assign" followed by a certain
assign value,
* the keyword “to”, and an identifier.
*/
void Assignment_Statement() :
{}
{ ASSIGN()(Assign_Value()) TO() IDENTIFIER() }
// Assign is one of Hira`s Keuwords
void ASSIGN() :
{Token t;}
{ t = <ASSIGN > {jjtThis.jjtSetValue(t.image);} }
// To is one of Hira`s Keuwords
void TO() :
{Token t;}
{ t = <TO> {jjtThis.jjtSetValue(t.image);} }


/**
* The assigned value can be: a string, number, Boolean value (true or false), an
assigned value
* for a registry or a stack ex: [1, 2, 3], arithmetic statement ex: 2 + 2.,
relational statement
* ex: 2 eq 2, or logical statement ex: @a and @b.
*/
void Assign_Value():
{}
{
   STR()( STR())*
|  NUM()   ((Arthmetic_Op() | Relational_Op()) NUM() )?
| IDENTIFIER() ( Arthmetic_Op() | Relational_Op() | Logical_Op()) IDENTIFIER()
| NOT_OP() IDENTIFIER ()
| BOOL()
| LEFT_BRACKET() ( REG() | STC()) RIGHT_BRACKET() }
/**
* Hira's string values are all letters, and can end with \. Representing the full
stop
*/
void STR() :
{Token t;}
{ t = < STR > {jjtThis.jjtSetValue(t.image);} }
/**
* Hira's boolean value can be either true or false
*/
void BOOL() :
{Token t;}
{ t = < BOOL > {jjtThis.jjtSetValue(t.image);} }
```

```
/**
* "[" is one of Eqtinaa`s symbols
*/
void LEFT_BRACKET() :
{Token t;}
{ t = < LEFT_BRACKET > {jjtThis.jjtSetValue(t.image);} }
/**
* Hira's registry value is one or more numbers separated by a comma ei, 3, 4, 5
*/
void REG() :
{Token t;}
{ t = < REG > {jjtThis.jjtSetValue(t.image);} }
/**
* Hira's stack value is one ore more stack content separated by a comma
*/
void STC() :
{Token t;}
{ t = < STC > {jjtThis.jjtSetValue(t.image);} }
// “]” is one of Hira`s symbols
void RIGHT_BRACKET():
{Token t;}
{ t = < RIGHT_BRACKET > {jjtThis.jjtSetValue(t.image);} }

/**
/**
* To define a conditional statement, it must start with the keyword “if” followed by
the
* condition, then the keyword “do”, colon “:”, and a block of statements and end with
the keyword “end”.
* The (if) statement can be followed by zero or more (or If) statements, which start
with the
* keyword “orIf” followed by the condition, the keyword “do”, colon “:” and a block
of
* statements and end with the keyword “end”.
* Else statement is optional, and it must start with the keyword “else” followed by
the keyword
* “do”, colon “:”, and a block of statements and end with the keyword “end”.
*/void Conditional_Statement():
{}
{    IF_CONDITION() Condition() DO() COLON() (Statements())+ END()
    (OR_CONDITION() Condition() DO() COLON() (Statements())+ END())*
    (ELSE() DO() COLON() (Statements())+ END())? }
/**
* Hira`s Conditions can 1. start with Numbers and end with numbers or identifiers,
ei.1 lt 2 or
* 2. Start with and identifiers followed by a logical operator and an identifier, ei,
@a and @b
* or 3. Starts with an identifier followed by a relational operator and a number or
an identifier ei,
* @a eq 3 or 4. A boolean value such as true or false
*/
void Condition():
{}
{
  ( NUM() Relational_Op() (NUM() | IDENTIFIER()))
| ( IDENTIFIER() (Logical_Op() IDENTIFIER() | Relational_Op() ( NUM() |
IDENTIFIER())))
|  BOOL() }
```

```
/**
* Hira's Conditional operators consist of "if", "orIf", "else", and "do".
*/
void IF_CONDITION():
{Token t;}
{ t = < IF_CONDITION > {jjtThis.jjtSetValue(t.image);} }
void DO():
{Token t;}
{ t = < DO > {jjtThis.jjtSetValue(t.image);} }
void COLON():
{Token t;}
{ t = < COLON > {jjtThis.jjtSetValue(t.image);} }
void OR_CONDITION():
{Token t;}
{ t =  < OR_CONDITION > {jjtThis.jjtSetValue(t.image);} }
void ELSE():
{Token t;}
{ t = < ELSE > {jjtThis.jjtSetValue(t.image);} }
void END():
{Token t;}
{ t = < END > {jjtThis.jjtSetValue(t.image);} }


/**
* An iterative statement must begin with the keyword "loop" followed by an
identifier, the
* keyword "using", another identifier, a colon, and a sequence of statements that end
with the
* keyword "end".
*/
void Iterative_Statement():
{}
{
  LOOP() IDENTIFIER() USE()  IDENTIFIER() COLON() (Statements())+ END() }

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

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