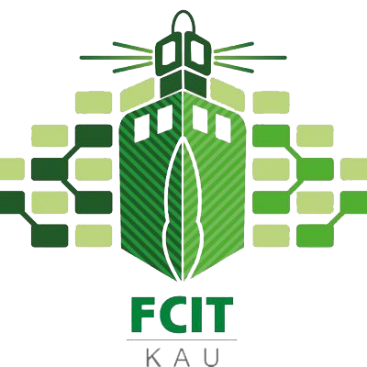
****

**King Abdul-Aziz University**

**Faculty of Computing and Information Technology**

**Computer Science Department**

**Compiler Construction**

**CPCS-304 Spring 2023 Group Project**

****

**Instructors: Dr. Sultanah M. Alshammari**

**I.Tahani S. Almoshadak**

**Section:. B10**

**Project Team:**

|  |  |
| --- | --- |
| **Name** | **ID** |
| Khlood Alsofyani |  |
|  |
|  |
|  |
|  |

**Table of Contents:**

[Phase 1: Lexical Analysis 4](#_Toc136122345)

[**1.1** **Introduction** 4](#_Toc136122346)

[**1.2** **Regular Expressions of the Tokens** 4](#_Toc136122347)

[**1.3** **Statements** 7](#_Toc136122348)

[Phase 2: Syntax Analysis 8](#_Toc136122349)

[**2.1** **BNF Form** 8](#_Toc136122350)

[**2.2** **BNF Explanation** 11](#_Toc136122351)

[Phase 3: Semantic Analysis 18](#_Toc136122352)

[Appendix A: JJ Grammar 22](#_Toc136122353)

[Appendix B: JJt Grammar 28](#_Toc136122354)

# **Phase 1: Lexical Analysis**

## **Introduction**

QMS is a programming language designed using JavaCC based on BNF rules. The main purpose of this project is to build a new high-level language that allows the user to enter several statements using different operations (e.g., arithmetic operations, relational operations, logical statements, if statements, etc.)

QMS identifier must start and end with an underscore, each statement ends with a dot, and its operations end with a question mark. More details are in the table below...

## **Regular Expressions of the Tokens**

|  |  |  |
| --- | --- | --- |
| **Token Type** | **Token Name** | **Regular Expression** |
| **Binary Arithmetic Operations** | PLUS | “+?” |
| MINUS | “-?” |
| MULTIPLICATION | “\*?” |
| DIVIDE | “/?” |
| REMINDER | “%?” |
| ASSIGN | “=?” |
| **Unary Arithmetic Operations** | INCREMENT | “++?” |
| DECREMENT | “--?” |
| **Relational Operations** | IS\_EQUAL | “==?” |
| NOT\_EQUAL | “!?” |
| GREATER\_THAN | “>?” |
| LESS\_THAN | “<?” |
| GREATER\_OR\_EQUAL | “>=?” |
| LESS\_OR\_EQUAL | “<=?” |
| **Logical Operators** | AND | “?AND?“ |
| OR | “?OR?“ |
| NEGATE | “?NOT?“ |
| **Punctuation Marks** | DOT | “.” |
| L\_BRACKET | “[“ |
| R\_BRACKET | “]” |
| L\_PARENTHESES | “(“ |
| R\_PARENTHESES | “)” |
| COLON | “:” |
| QMS | “?” |
| **Alphabets** | LETTER | [“a”-“z”] | [“A”-“Z”] |
| LETTERS | [“a”-“z”]+ | [“A”-“Z”]+ |
| DIGIT | [“0”-“9”] |
| SYMBOL | [“|”, “&” , “;” , “#” , “$” , “@” , “:” , “,” ,“?”] |
| **Identifiers** | ID | \_(LETTER | DIGIT | SYMBOL)+\_ |
| **White Spaces** | SPACE | (“ “) |
| NEWLINE | (“\n”) |
| TAB | (“\t”) |
| **Data Type** | INTEGER | (DIGIT)+ |
| DECIMAL | (DIGIT)+ (“.” )(DIGIT)+ |
| STRING | (LETTER | DIGIT | SYMBOL)+ |
| BOOLEAN | “ T “ | “F” |
| **Keywords** | IF | DO | IF\_ELSE | OTHER | INTEGER \_TYPE | DECIMAL \_TYPE | BOOLEAN\_TYPE | STRING\_TYPE |  LOOP | ENDLOOP | PRINT | CONSTANT | ARRAY |  LINKED\_LIST | EXIT | "IF" | “DO” | ”ELSEIF” |"OTHER" |"INTEGER" |”DECIMAL” | “STRING” |” BOOLEAN”| "LOOP?" | "?END" | “PRINT” | “CONST” |”?ARR” | “?LL” | “EXIT” |
| **Comments** | SINGLE\_LINE\_COMMENT | (??) STRING |
| MULTIPLE \_LINE\_COMMENT | (?\*) STRING (\*?) |
| **Constant** | CONSTANT | (?) CONST DataType ID (=?) Data |
| **Iterative Statement** | LOOP | (LOOP?) ([Condition]) (:)  (STMTS)  (?END) |
| **Conditional Statements** | IF DO | (IF) ([Condition]) (DO) (PRINT) ([STMTS]) |
| IF DO OTHER | (IF) ([Condition]) (DO) (PRINT) ([STMTS]) (OTHER) (PRINT) ([STMTS]) |

## **Statements**

|  |  |  |
| --- | --- | --- |
| **Language Statement** | **Regular Expression** | **Example (Code)** |
| **Arithmetic Statements (Binary)** | ID Multiplication ID Dot | \_length\_ \*? \_wight\_. |
| **Arithmetic Statements (Unary)** | Increment ID Dot | ++? \_Counter\_. |
| **Relational Statements** | ID Greater\_Or\_Equal ID Dot | \_i\_>=? \_j\_. |
| **Logical Statements** | BOOLEAN OR BOOLEAN Dot | T ?OR? T. |
| **Boolean Statements** | BOOLEAN ID Assign ID Greater\_Or\_Equal ID Dot | BOOLEAN \_@temp\_ =? \_i\_ >=?\_j\_. |
| **Conditional Statements** | IF L\_Bracket Condition R\_Bracket DO PRINT L\_Bracket Stmts  R\_Bracket Dot | IF [\_grade\_>? 60]  DO PRINT [\_G\_ =? 5]. |
| IF L\_Bracket Condition R\_Bracket DO PRINT L\_Bracket Stmts  R\_Bracket OTHER PRINT L\_Bracket Stmts  R\_Bracket Dot | IF [\_grade\_>? 60]  DO PRINT [passed]  OTHER PRINT [failed]. |
| **Iterative Statement** | LOOP L\_Bracket Condition R\_Bracket COLON Stmts ENDLOOP Dot | LOOP? [10 <? 5]:  \_x\_ =? 1 ?END. |
| **Data Structures** | ARRAY ID COLON L\_BRACKET  (LETTER | DIGIT )\* R\_BRACKET | ?ARR \_letters\_: [Leen|rawan|Lana]  ?ARR \_letter\_:[L|e|R|r]  ?ARR \_num\_:[2|5|7|0] |
| LINKED\_LIST ID COLON L\_BRACKET (L\_BRACKET (LETTER|DIGIT|SYMBOL)+  R\_BRACKET )+R\_BRACKET | ?LL \_NodeLL\_: [[L1,2,2] [L2,3,3]] |

# **Phase 2: Syntax Analysis**

## **BNF Form**

**1. Start à Stmts DOT | EXIT**

**2. Stmts à IFStmt |Assignment | Iteration | Constant| DataStructures | Stmt |Data Type**

**3. Assignment à ID Assign Stmt**

**4. Stmt à (ID |Data | BooleanStmt)? (ArithmeticStmt | RelationalStmt | LogicalStmt | ConditionalStmt)?**

**5. ArithmeticStmt à (Binary OP | Unary OP) (ID | Number)**

**6. RelationalStmt à RelationalOP (ID | Number)**

**7. LogicalStmt à LogicalOP BOOLEAN**

**8. BooleanStmt à BOOLEAN\_TYPE ID Assign (ID RelationalOP ID)**

**9. BOOLEAN à “ T “ | “F”**

**10. IF Stmt à IF L\_BRACKET Condition R\_BRACKET DO PRINT L\_BRACKET Stmts R\_BRACKET (OTHER PRINT L\_BRACKET Stmts R\_BRACKET)?**

**11. Condition à (ID |Number) RelationalOP( ID | Number)**

**12. Data Type à INTEGER\_TYPE | DECIMAL\_TYPE | STRING\_TYPE | BOOLEAN\_TYPE**

**14. INTEGER à (DIGIT)+**

**15. DECIMAL à (DIGIT)+ (“.” )(DIGIT)+**

**16. STRING à (LETTER | DIGIT | SYMBOL)+**

**17. Iteration à LOOP L\_Bracket Condition R\_Bracket COLON Stmt+ ENDLOOP**

**18. Constant à QMS CONSTANT DeclareVar**

**19. DeclareVar à INTEGER\_TYPE ID ASSIGN INTEGER| DECIMAL\_TYPE ID ASSIGN DECIMAL**

**| STRING ID ASSIGN STRING| BOOLEAN ID ASSIGN BOOLEAN**

**20. Data structures à ARRAY ID COLON L\_BRACKET (Data)\* R\_BRACKET | LINKED\_LIST ID COLON L\_BRACKET (Data)+ R\_BRACKET (L\_BRACKET (Data)+ R\_BRACKET)\* R\_BRACKET**

**21. LogicalOP à AND | NEGATE | OR**

**22. RelationalOP à Is\_Equal |Not\_Equal| Greater\_Than | Less\_Than | Greater\_Or\_Equal | Less\_Or\_EQUAL**

**23. Data à INTEGER | DECIMAL | STRING | BOOLEAN**

**24. Number à INTEGER | DECIMAL**

**25. AND à “?AND?“**

**26. NEGATE à “?NOT?“**

**27. OR à “?OR?“**

**28. Is\_Equal à “==?”**

**29. Not\_Equal à “!?”**

**30. Greater\_Than à “>?”**

**31. Less\_Than à “<?”**

**32. Greater\_Or\_Equal à “>=?”**

**33. Less\_Or\_Equal à “<=?”**

**34. UnaryOP à INCREMENT| DECREMENT**

**35. INCREMENT à “++?”**

**36. DECREMENT à “--?”**

**37. BinaryOP à PLUS | MINUS | MULTIPLICATION | DIVIDE| REMINDER | ASSIGN**

**38. Plus à “+?”**

**39. Minus à “-?”**

**40. Multiplication à “\*?”**

**41. Divide à “/?”**

**42. Reminder à %?”**

**43. Assign à “=?”**

**44. ID à \_(LETTER | DIGIT | SYMBOL)+\_**

**45. LETTER à [“a”-“z”] | [“A”-“Z”]**

**46. LETTERS à [“a”-“z”]+ | [“A”-“Z”]+**

**47. DIGIT à [“0”-“9”]**

**48. SYMBOL à [“|”, “&” , “;” , “#” , “$” , “@” , “:” , “,” ,“?”]**

**49. Dot à “.”**

**50. L\_Bracket à “[“**

**51. R\_Bracket à “]“**

**52. L\_ parentheses à “(“**

**53. R\_ parentheses à “)“**

**54. COLON à “:“**

**54. QMS à “?“**

## **BNF Explanation**

**1. Start à Stmts DOT | EXIT**

A statement in QMS language begins with statement type followed by dot “.”, or the keyword EXIT to quit.

**2. Stmts à IFStmt |Assignment | Iteration | Constant| DataStructures | Stmt |Data Type**

The statement in this language can be an IFStmt statement, or an assignment statement, Data Type, iteration, or constant, and also can be a Datastructures.

**3. Assignment à ID Assign Stmt**

Any assignment begins with an identifier assigned to a specific statement.

**4. Stmt à (ID |Data | BooleanStmt)? (ArithmeticStmt | RelationalStmt | LogicalStmt | ConditionalStmt)?**

The statement in QMS should begin with an ID or any Data type such as integer or decimal or string or Boolean statement followed by a question mark, or can be Boolean statement followed by any type of statement, such as an arithmetic or relational statement, logical, Boolean, or conditional statement, followed by a question mark.

**5. ArithmeticStmt à (Binary OP | Unary OP) (ID | Number)**

**Ex : ArithmeticStmt à ( MULTIPLICATION| INCREMENT) (ID | Number)**

An example of Arithmetic statements in our language here is binary multiplication statement and unary increment statement, followed by an ID or number. Here the arithmetic statement may be a binary multiplication statement or a unary increment statement.

**6. RelationalStmt à RelationalOP (ID | Number)**

**Ex : RelationalStmt à Greater\_Or\_Equal (ID | Number)**

Relational statement begins with a Relational OP (Example: Greater\_Or\_Equal ) and end with ID or number.

**7. LogicalStmt à LogicalOP BOOLEAN**

**Ex : LogicalStmt à BOOLEAN OR BOOLEAN**

Another type of statements in QMS language is a logical statement, which begins with a boolean data type ( T means true, or F means false ) followed by a logical operation such as OR in this example, and ends with the second boolean data type.

**8. BooleanStmt à BOOLEAN\_TYPE ID Assign (ID RelationalOP ID)**

**Ex: BooleanStmt à BOOLEAN\_Type ID Assign ID Greater\_Or\_Equal ID**

Boolean statement should begin with a boolean data type followed by an identifier assigned to another an identifier followed by relational operation (example: Greater\_Or\_Equal) and ends with ID.

**9. IF Stmt à IF L\_BRACKET Condition R\_BRACKET DO PRINT L\_BRACKET Stmts R\_BRACKET (OTHER PRINT L\_BRACKET Stmts R\_BRACKET)?**

The conditional IF statements in this language begin with IF keyword followed by the condition between left and right brackets, then do and print instructions refer to the statement at the end of the IF statement surrounded by brackets and end with the question mark.

**10. Condition à (ID |Number) RelationalOP( ID | Number)**

The condition in QMS language begins with an ID or number followed by a relational operation and end with another ID or Number to be compared.

**11. Data Type à INTEGER\_TYPE | DECIMAL\_TYPE | STRING\_TYPE | BOOLEAN\_TYPE**

Data type has several uses in QMS language like declarations and several types of statements. Data type may be an integer type,, decimal type,, string type, or BOOLEAN.

**12.** **INTEGER à (DIGIT)+**

An integer data type should begin with an underscore followed by one or more digit [0-9]

**13.** **DECIMAL à (DIGIT)+ (“.” )(DIGIT)+**

The decimal data type should begin with an underscore followed by one or more digit [0-9] and must followed by dot(“.”) and one or more digit [0-9].

**14.** **STRING à (LETTER | DIGIT | SYMBOL)+**

String data type begins with underscore followed by one or more letter or digits or any symbol.

**15.** **Iteration à LOOP L\_Bracket Condition R\_Bracket COLON Stmt+ ENDLOOP**

Iteration statements are iterative statements depending on a specific condition, in QMS Language iteration begins with LOOP keyword followed by a condition between two brackets, followed by a colon “:” then one or more iterative statements and ends with ENDLOOP keyword.

**16.** **Constant à QMS CONSTANT DeclareVar**

The constant data type is a special kind of data type that can not change after the declaration, It begin with QMS symbol “?” followed by the constant declaration.

**17. DeclareVar à INTEGER\_TYPE ID ASSIGN INTEGER| DECIMAL\_TYPE ID ASSIGN DECIMAL | STRING ID ASSIGN STRING| BOOLEAN ID ASSIGN BOOLEAN**

Declare a variable can be several kinds of Data types, it may be an integer assigned to integer type, or decimal or string and Boolean data type.

**18. Data structures à ARRAY ID COLON L\_BRACKET (Data)\* R\_BRACKET | LINKED\_LIST ID COLON L\_BRACKET (Data)+ R\_BRACKET (L\_BRACKET (Data)+ R\_BRACKET)\* R\_BRACKET**

Data structures are an important part of QMS language for storage and Data processing, it may be an array begins with ARRAY keyword followed by ID and colon “:”, then zero or more integer or decimal or string or Boolean between two brackets, or it may be linked list beginning with LINKED\_LIST keyword followed by ID and colon “:” that has one or more integer or decimal or string or Boolean, between two brackets, followed by one or more elements that has the same characteristics.

**19. LogicalOP à AND | NEGATE | OR**

The logical operations have some operational symbols. The symbol may be AND or OR or NEGATE.

**20. RelationalOP à Is\_Equal |Not\_Equal| Greater\_Than | Less\_Than | Greater\_Or\_Equal | Less\_Or\_EQUAL**

The relational operations are the most important part in the conditional statement and iterations, it may be Is\_Equal (“==?”), Not\_Equal (“!?”), Greater\_Than (“>?”), or Less\_Than (“<?”), Greater\_Or\_Equal (“>=?”), and Less\_Or\_Equal (“<=?).

**21. Data à INTEGER | DECIMAL | STRING | BOOLEAN**

Data refers to different data types, it can be integer or decimal, string or Boolean.

**22. Number à INTEGER | DECIMAL**

Number in QMS language can be integer or decimal with the fractional part.

**23. UnaryOP à INCREMENT| DECREMENT**

Unary operations refers to increment (“++?”) or decrement (“--?”) by one.

**24.** **BinaryOP à PLUS | MINUS | MULTIPLICATION | DIVIDE| REMINDER | ASSIGN**

Binary operations used in arithmetic operations in QMS language, it may be Plus (“+?”), Minus (“-?”), Multiplication (“\*?”), Divide (“/?”), Reminder (“%?”), Assign (“=?”).

**25.** **ID à \_(LETTER | DIGIT | SYMBOL)+\_**

The identifiers in this language begin and end with the underscore followed by one or more letters or digits or symbols.

**26. SYMBOL à [“|”, “&” , “;” , “#” , “$” , “@” , “:” , “,” ,“?”]**

There are a symbols in QMS language that accepted such as |, &, ;, #, $, @, ,, ?.

The rest of the BNF is a description of the symbols used to express QMS language tokens.

* 1. **BNF Screenshots of JJ**

1. \_length\_ \*? \_wight\_.

**A screenshot of a computer

Description automatically generated with medium confidence**

1. ++? \_Counter\_.

**A white background with black text

Description automatically generated with low confidence**

1. \_i\_>=? \_j\_.

**A picture containing text, screenshot, font, line

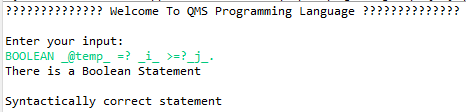
Description automatically generated**

1. T ?OR? T.

A picture containing text, screenshot, font, line

Description automatically generated

1. BOOLEAN \_@temp\_ =? \_i\_ >=?\_j\_.

****

1. IF [\_grade\_>? 60] DO PRINT [\_G\_ =? 5].

A picture containing text, screenshot, font, line

Description automatically generated

1. IF [\_grade\_>? 60] DO PRINT [passed] OTHER PRINT [failed].

**A picture containing text, screenshot, font, line

Description automatically generated**

1. LOOP? [10 <? 5]: \_x\_ =? 1 ?END.

**A screenshot of a computer screen

Description automatically generated with low confidence**

1. ?ARR \_H\_ : [ 7,8,5] .

**A picture containing text, screenshot, font, line

Description automatically generated**

1. ?LL \_NodeLL\_: [[L1,2,2] [L2,3,3]].

**A picture containing text, screenshot, font, line

Description automatically generated**

1. ? CONST STRING \_Var\_ =? Qms.

**A picture containing text, screenshot, font, line

Description automatically generated**

1. ?? Treminate the program

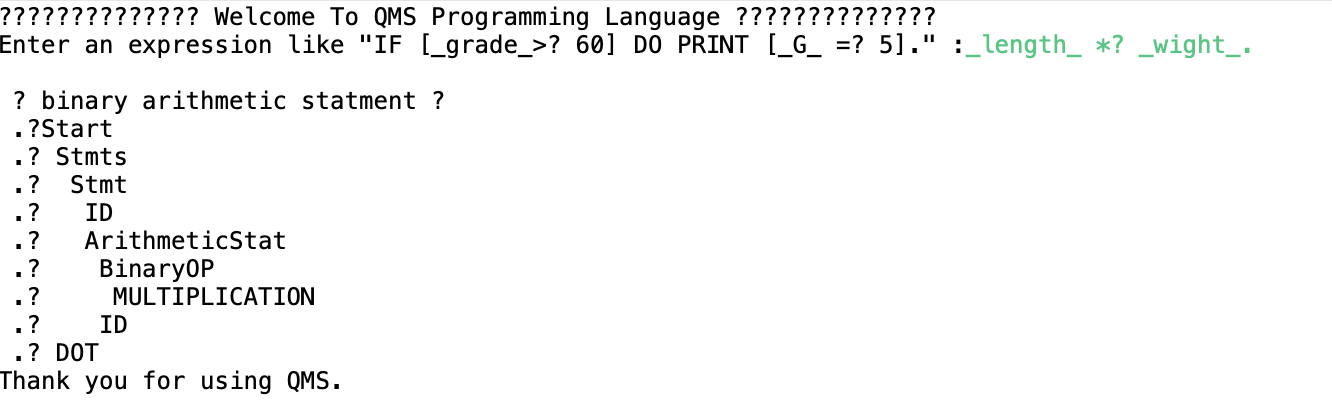
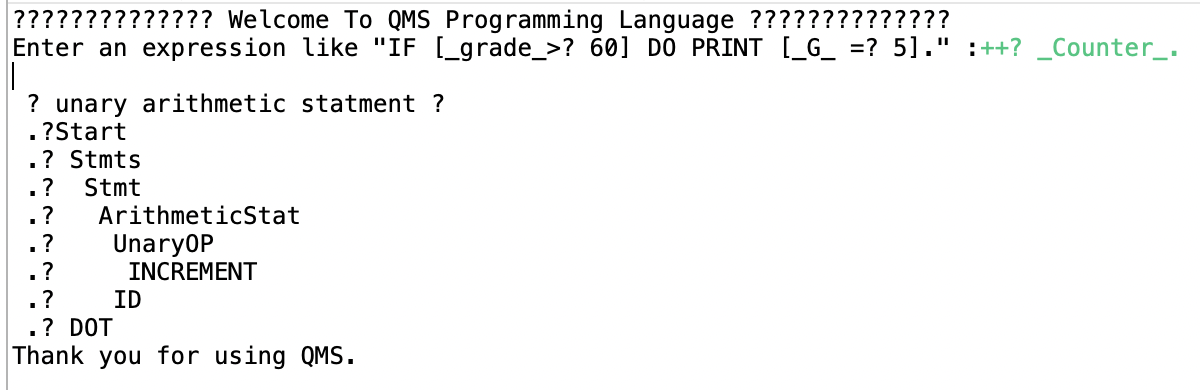
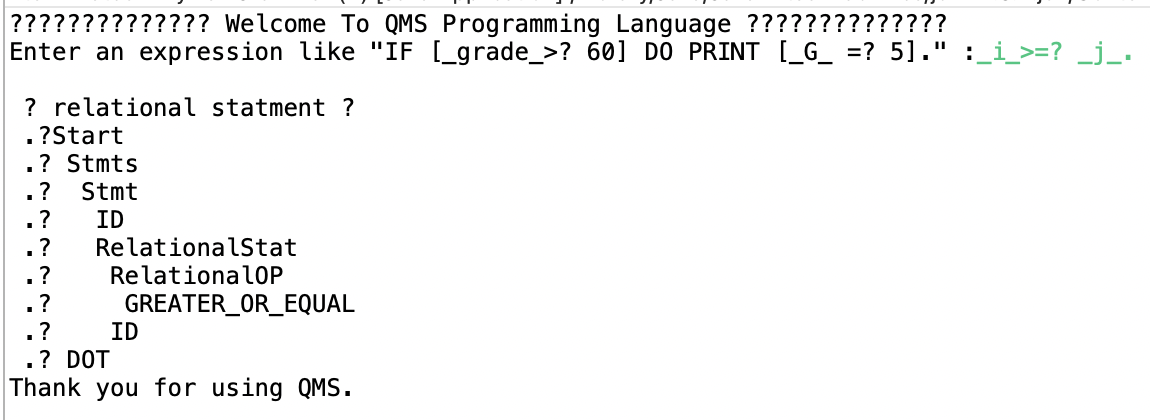
EXIT**A picture containing text, screenshot, font, line

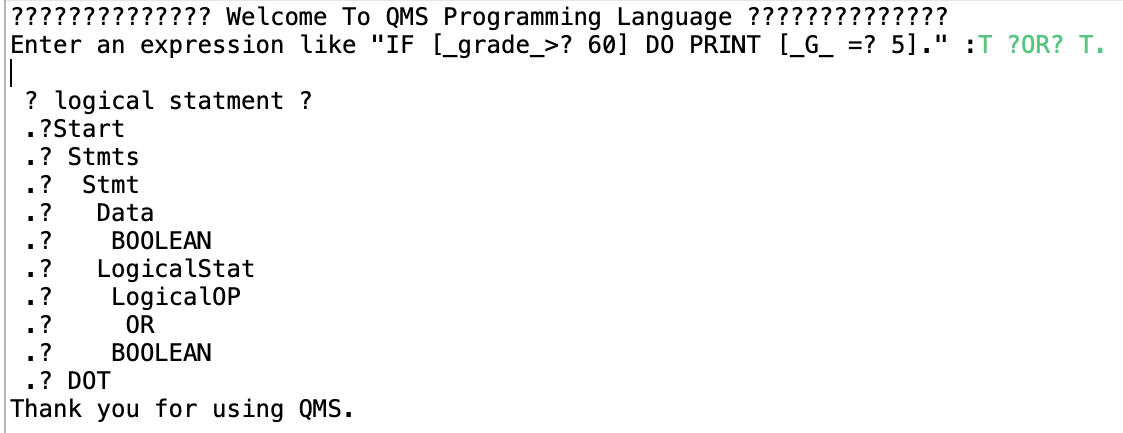
Description automatically generated**

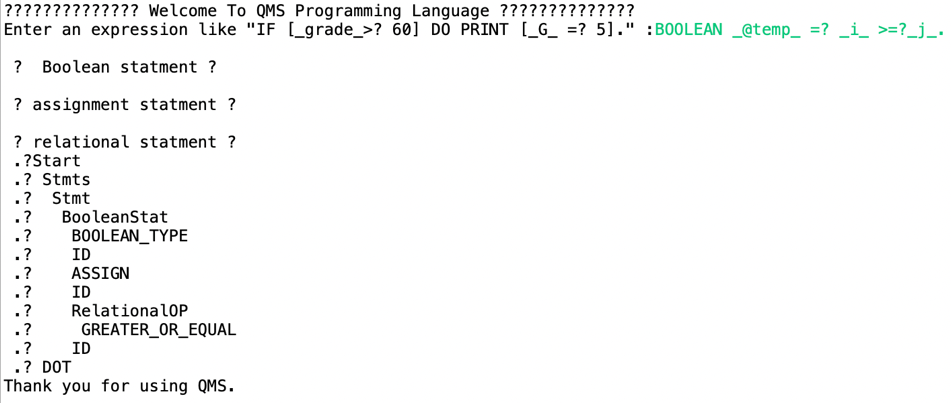
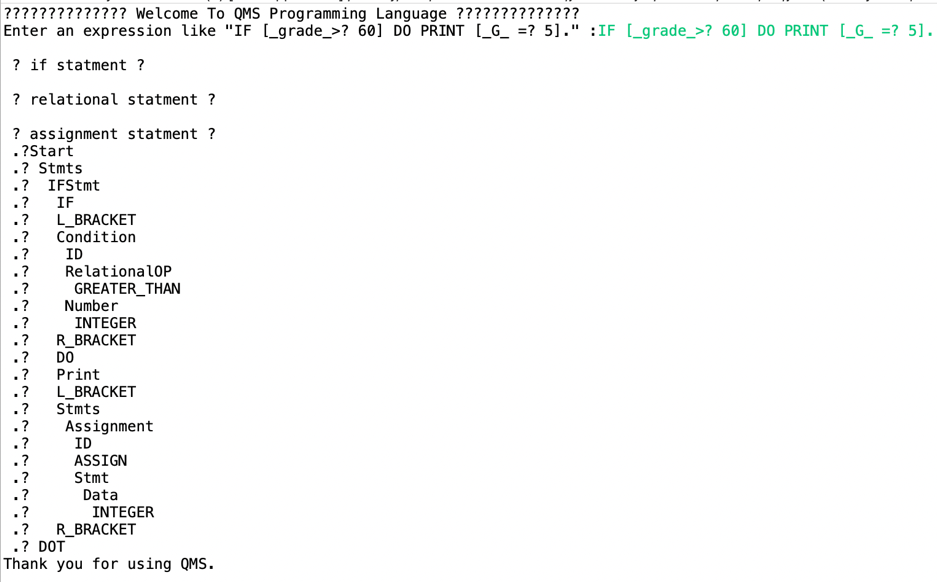
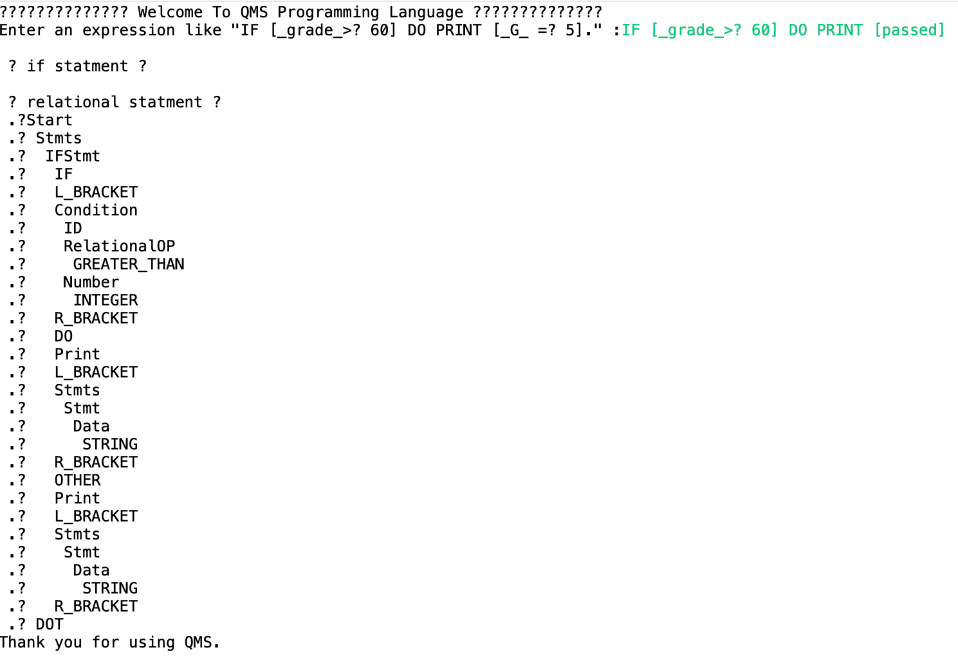
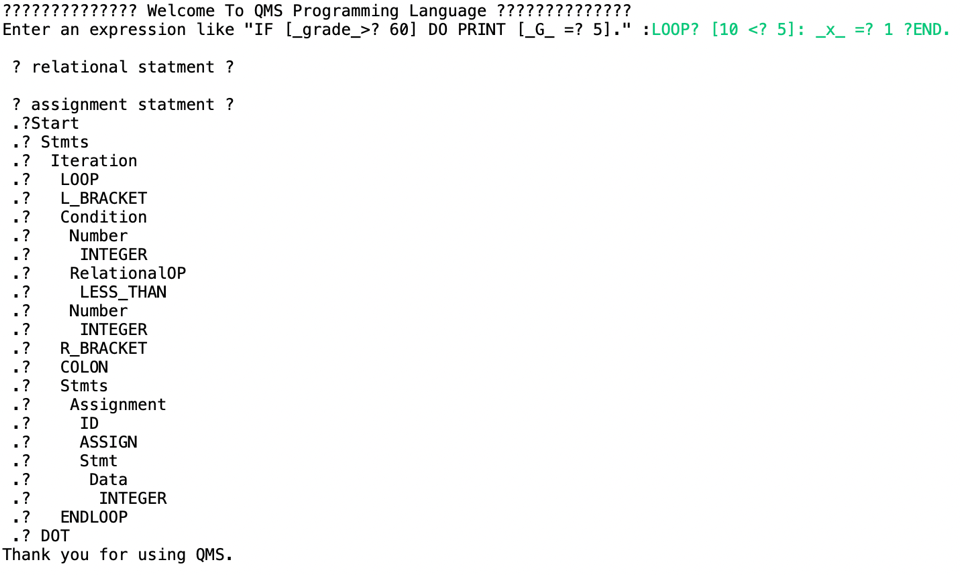
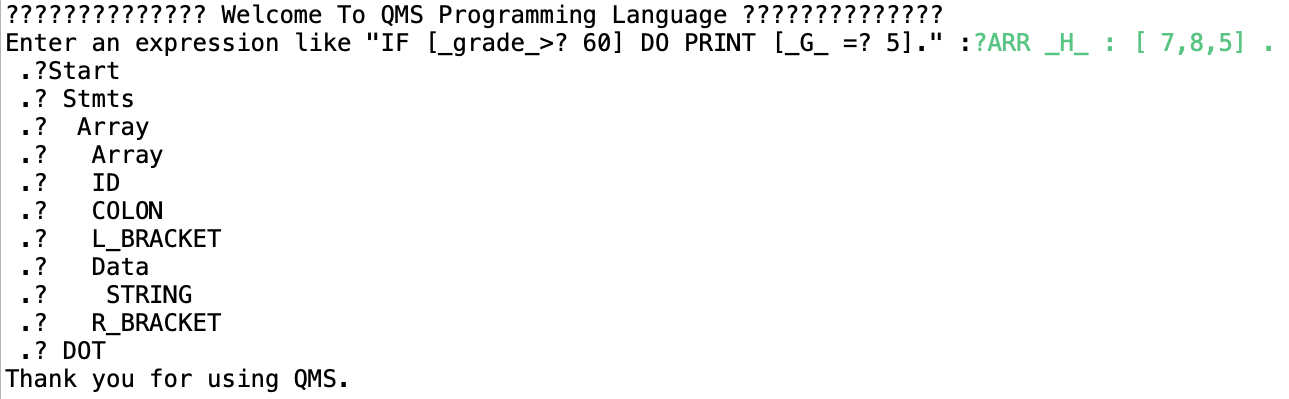
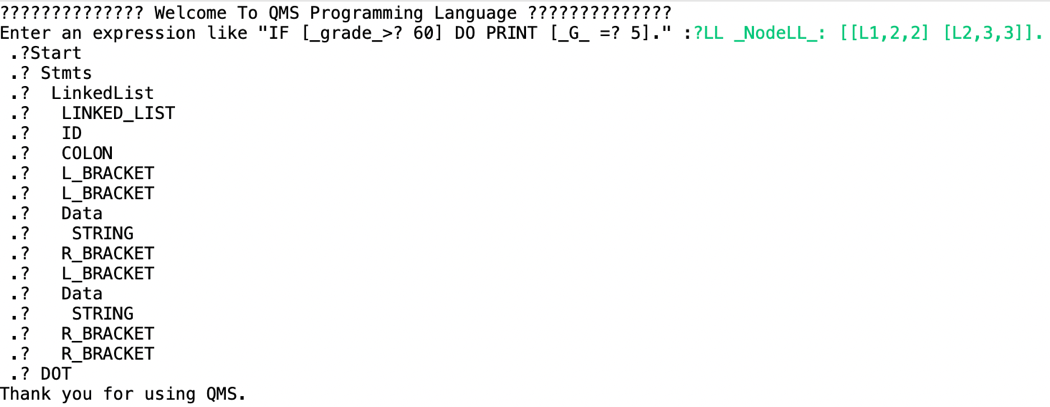
A picture containing text, screenshot, font

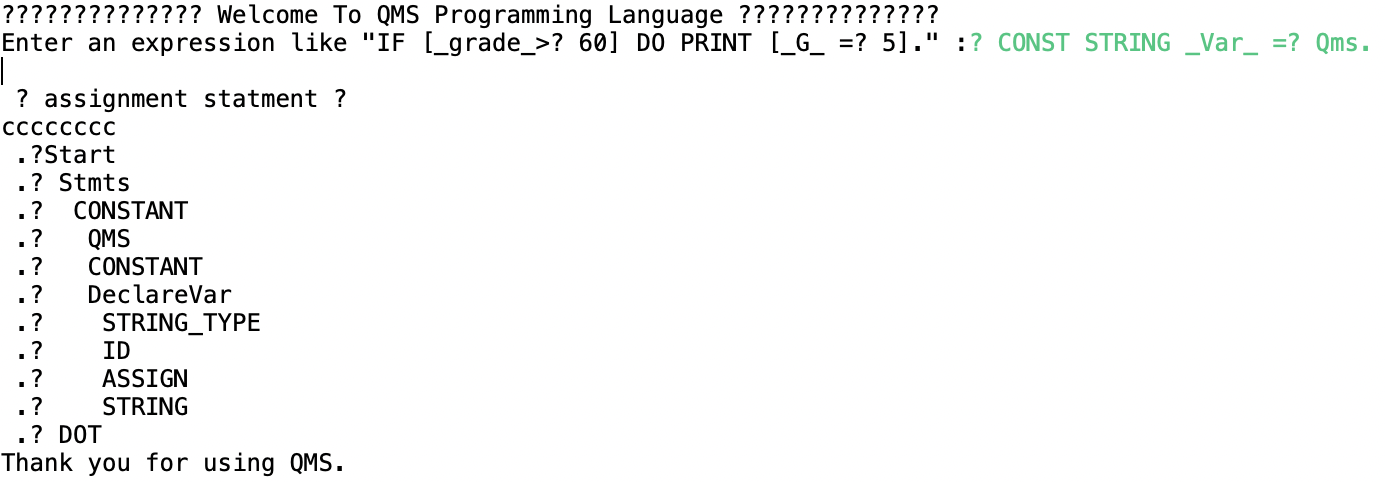
Description automatically generated**Some Lexical Actions:**

# **Phase 3: Semantic Analysis**

1. ****\_length\_ \*? \_wight\_.
2. ****++? \_Counter\_.
3. ****\_i\_>=? \_j\_.
4. T ?OR? T.

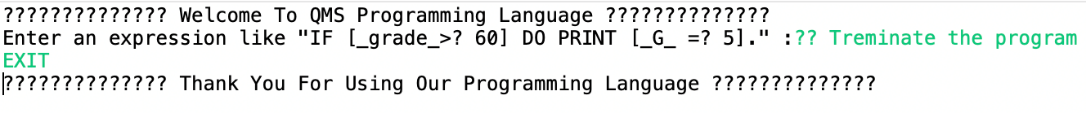


1. ****BOOLEAN \_@temp\_ =? \_i\_ >=?\_j\_.
2. IF [\_grade\_>? 60] DO PRINT [\_G\_ =? 5].
3. ****IF [\_grade\_>? 60] DO PRINT [passed] OTHER PRINT [failed].
4. ****LOOP? [10 <? 5]: \_x\_ =? 1 ?END.
5. ****?ARR \_H\_ : [ 7,8,5] .
6. ****?LL \_NodeLL\_: [[L1,2,2] [L2,3,3]].
7. ? CONST STRING \_Var\_ =? Qms.



1. ?? Treminate the program

EXIT



**Some Lexical Actions**:

# **Appendix A: JJ Grammar**

/\*\*

\* CPCS302 Project Group6

\* Razan Alamri - Leen BA QALAQL - KHLOUD ALSOFYANI - Reema Almalki -RAWAN ALGHAMDI

\*/

**options**

**{**

**static** = **true**;

**}**

**PARSER\_BEGIN(MyNewGrammar)**

**package** CPCS302Project;

**public** **class** MyNewGrammar {

**public** **static** **void** main(String args []) **throws** ParseException {

MyNewGrammar parser = **new** MyNewGrammar(System.in);

System.out.println("?????????????? Welcome To QMS Programming Language ??????????????");

**while** (**true**){

System.out.println("\nEnter your input: ");

**try**{

parser.Start();

System.out.println("\nSyntactically correct statement");

}

**catch** (Exception e){

System.out.println("Syntactically NOT correct statement");

System.out.println(e.getMessage());

**break**;

}

}

}

}

**PARSER\_END(MyNewGrammar)**

**SKIP** : **{**

/\* White spaces \*/

**<** SPACE: " " **>**

**|** **<** TAB: "\t" **>**

**|** **<** NEWLINE:"\n" **>**

**|** "\r"

**}**

/\* Comments \*/

**SPECIAL\_TOKEN** : **{**

// (~["\n"])\*

**<** SINGLE\_LINE\_COMMENT: "??"**(~[**"\n"**])\*** **>**

**|** **<** MULTIPLE\_LINE\_COMMENT: "?\*"**(**< STRING >**)\***"\*?"**>**

**}**

/\* Keywords \*/

**TOKEN** : **{**

**<** IF: "IF" **>**

**|** **<** DO: "DO" **>**

**|** **<** IF\_ELSE: "ELSEIF" **>**

**|** **<** OTHER: "OTHER" **>**

**|** **<** INTEGER\_TYPE: "INTEGER" **>**

**|** **<** DECIMAL\_TYPE: "DECIMAL" **>**

**|** **<** BOOLEAN\_TYPE: "BOOLEAN" **>**

**|** **<** STRING\_TYPE: "STRING" **>**

**|** **<** LOOP: "LOOP?" **>**

**|** **<** ENDLOOP: "?END" **>**

**|** **<** PRINT: "PRINT" **>**

**|** **<** CONSTANT: "CONST" **>**

**|** **<** ARRAY: "?ARR" **>**

**|** **<** LINKED\_LIST: "?LL" **>**

**|** **<** EXIT: "EXIT"**>**

**{**System.out.println("?????????????? Thank You For Using Our Programming Language ??????????????");

System.exit(0); **}**

**}**

/\* Binary arithmetic operations \*/

**TOKEN** : **{**

**<** PLUS: "+?" **>**

**|** **<** MINUS: "-?" **>**

**|** **<** MULTIPLICATION: "\*?" **>**

**|** **<** DIVIDE: "/?" **>**

**|** **<** REMINDER: "%?" **>**

**|** **<** ASSIGN: "=?" **>**

**}**

/\* Unary arithmetic operations \*/

**TOKEN** : **{**

**<** INCREMENT: "++?" **>**

**|** **<** DECREMENT: "--?" **>**

**}**

/\* Relational operations \*/

**TOKEN** : **{**

**<** IS\_EQUAL: "==?" **>**

**|** **<** NOT\_EQUAL: "!?" **>**

**|** **<** GREATER\_THAN: ">?" **>**

**|** **<** LESS\_THAN: "<?" **>**

**|** **<** GREATER\_OR\_EQUAL: ">=?" **>**

**|** **<** LESS\_OR\_EQUAL: "<=?" **>**

**}**

/\* Logical operators \*/

**TOKEN** : **{**

**<** AND: "?AND?" **>**

**|** **<** OR: "?OR?" **>**

**|** **<** NEGATE: "?NOT?" **>**

**}**

/\* Punctuation Marks \*/

**TOKEN** : **{**

**<** DOT: "." **>**

**|** **<** COLON: ":" **>**

**|** **<** L\_BRACKET: "[" **>**

**|** **<** R\_BRACKET: "]" **>**

**|** **<** L\_PARENTHESIS: "(" **>**

**|** **<** R\_PARENTHESIS: ")" **>**

**|** **<** QMS: "?" **>**

**}**

/\* Identifiers \*/

**TOKEN** : **{**

**<** ID: "\_" **(** <LETTER> **|** <DIGIT> **|** <SYMBOL> **)+** "\_" **>**

**}**

/\* Data type \*/

**TOKEN** : **{**

**<** INTEGER: **(** <DIGIT> **)+** **>**

**|** **<** DECIMAL: **(** <DIGIT> **)+** "." **(** <DIGIT> **)+** **>**

**|** **<** BOOLEAN: **(** "T" **|** "F"**)** **>**

**|** **<** STRING: **(** <LETTER> **|** <DIGIT> **|** <SYMBOL> **)+** **>**

**}**

/\* Alphabets \*/

**TOKEN** : **{**

**<** #LETTER: **[**"a"**-**"z", "A"**-**"Z"**]** **>**

**|** **<** LETTERS: **([**"a"**-**"z", "A"**-**"Z"**])+** **>**

**|** **<** #DIGIT: **[**"0"**-**"9"**]** **>**

**|** **<** #SYMBOL: **[**"|", "&", ";", "#", "$", "@", ":", ",", "?"**]** **>**

**}**

//???????????????????????????? BNF ????????????????????????????

/\* This method is starting point of grammar / the EXIT keyword to terminate the program. \*/

**void** Start() :

**{}**

**{**

Stmts() <DOT> **|** <EXIT>

**}**

/\* This method defines a set of rules for constructing statements in the grammar. \*/

**void** Stmts() :

**{}**

**{**

IFStmt()

**|** **LOOKAHEAD(**3**)** Assignment()

**|** Iteration()

**|** Constant()

**|** DataStructures()

**|** **LOOKAHEAD(**3**)** Stmt()

**|** DataType()

**}**

/\* This method recognizes an assignment statement,

which consists of an identifier followed by an assignment operator (=?) and a statement.\*/

**void** Assignment() :

**{}**

**{** **{** System.out.println("There is an Assignment Statement"); **}**

**(**<ID>**)** **(**<ASSIGN>**)** Stmt()

**{** System.out.println("There is an Assignment Statement"); **}**

**}**

/\* This method recognizes a basic statement. \*/

**void** Stmt(): **{}**

**{**

**((**<ID>**)** **|** Data()**|** BooleanStat()**)?**

**(**

ArithmeticStat()

**|** RelationalStat()

**|** LogicalStat()

**)?**

**}**

/\* ArithmeticStatement(): This method recognizes an arithmetic statement,

which consists of a binary or unary operator followed by an identifier or a number. \*/

**void** ArithmeticStat(): **{}**

**{**

**(**BinaryOP() **|** UnaryOP()**)** **(**<ID> **|** Number()**)**

**{** System.out.println("There is a Arithmatic Statement"); **}**

**}**

// Binary arithmetic operator.

**void** BinaryOP(): **{}**

**{**

**(**<MULTIPLICATION> **|** <DIVIDE> **|** <PLUS> **|** <MINUS> **|** <REMINDER> **|** <ASSIGN> **)**

**{** System.out.println("There is a Binary Operation"); **}**

**}**

// Unary arithmetic operator.

**void** UnaryOP(): **{}**

**{**

**(**<INCREMENT> **|** <DECREMENT>**)**

**{** System.out.println("There is a Unary Operation"); **}**

**}**

/\* This method recognizes a relational expression,

which consists of a relational operator followedby an identifier or a number. \*/

**void** RelationalStat(): **{}**

**{**

RelationalOP() **(**<ID> **|** Number()**)**

**{** System.out.println("There is a Relational Statement"); **}**

**}**

// Relational operator.

**void** RelationalOP(): **{}**

**{**

<IS\_EQUAL> **|** <NOT\_EQUAL> **|** <GREATER\_THAN> **|** <LESS\_THAN> **|** <GREATER\_OR\_EQUAL> **|** <LESS\_OR\_EQUAL>

**}**

/\* This method recognizes a logical expression,

which consists of a logical operator followed by a boolean value. \*/

**void** LogicalStat(): **{}**

**{**

LogicalOP() <BOOLEAN>

**{**System.out.println("There is a Logical Statement"); **}**

**}**

// Logical operator.

**void** LogicalOP(): **{}**

**{**

<AND> **|** <OR> **|** <NEGATE>

**}**

/\* This method recognizes a boolean statement,

which consists of a boolean type followed by an identifier and a relational expression. \*/

**void** BooleanStat(): **{}**

**{**

<BOOLEAN\_TYPE> <ID> <ASSIGN> **(**<ID> RelationalOP() <ID>**)**

**{** System.out.println("There is a Boolean Statement"); **}**

**}**

/\* This method recognizes a data value,

which can be an integer, decimal, string, or boolean. \*/

**void** Data(): **{}**

**{**

<INTEGER> **|** <DECIMAL> **|** <STRING> **|** <BOOLEAN>

**}**

// Number is INTEGER or DECIMAL.

**void** Number(): **{** **}**

**{**

<INTEGER> **|** <DECIMAL>

**}**

/\* This method recognizes a data type,

which can be an integer, decimal, string, or boolean. \*/

**void** DataType(): **{}**

**{**

<INTEGER\_TYPE> **|** <DECIMAL\_TYPE> **|** <STRING\_TYPE> **|** <BOOLEAN\_TYPE>

**{** System.out.println("It is a Data Type"); **}**

**}**

/\* This method recognizes an iteration statement,

which consists of a loop keyword followed by a condition in brackets, a colon, and a set of statements.

It also recognizes the END keyword to terminate the loop. \*/

**void** Iteration(): **{}**

**{**

<LOOP> <L\_BRACKET> Condition() <R\_BRACKET> <COLON> Stmts() <ENDLOOP>

**{** System.out.println("There is Iteration Statement"); **}**

**}**

/\* This method recognizes a constant declaration,

which consists of a constant symbol followed by a variable declaration. \*/

**void** Constant(): **{}**

**{**

<QMS> <CONSTANT> DeclareVar()

**}**

/\* This method recognizes a variable declaration,

which consists of a data type, an identifier, an assignment operator, and a value. \*/

**void** DeclareVar(): **{}**

**{**

<INTEGER\_TYPE> <ID> <ASSIGN> <INTEGER>

**{** System.out.println("There is Integer decleration"); **}**

**|** <DECIMAL\_TYPE> <ID> <ASSIGN> <DECIMAL>

**{** System.out.println("There is Decimal decleration"); **}**

**|** <STRING\_TYPE> <ID> <ASSIGN> <STRING>

**{** System.out.println("There is String decleration"); **}**

**|** <BOOLEAN\_TYPE> <ID> <ASSIGN> <BOOLEAN>

**{** System.out.println("There is Boolean decleration"); **}**

**}**

/\* This method recognizes an if statement,

which consists of an if keyword followed by a condition in brackets, a do keyword, a print statement, and a set of statements.

It can also include an ELSEIF keyword and another set of statements in case the condition is not met. \*/

**void** IFStmt(): **{}**

**{**

<IF> <L\_BRACKET> Condition() <R\_BRACKET> <DO> <PRINT> <L\_BRACKET> Stmts() <R\_BRACKET>

**(**<OTHER> <PRINT> <L\_BRACKET> Stmts() <R\_BRACKET>**)?**

**{**System.out.println("There is If Statement"); **}**

**}**

/\* This method recognizes a condition,

which consists of an identifier or a number followed by a relational operator and another identifier or number.\*/

**void** Condition(): **{}**

**{**

**(** **(**<ID> **|** Number()**)** RelationalOP() **(**<ID> **|** Number()**)** **)**

**}**

/\* This method recognizes a data structure,

which can be an array or a linked list.

For arrays, it expects to see an identifier followed by a colon, a left bracket, a list of data values, and a right bracket.

For linked lists, it expects to see an identifier followed by a colon, a left bracket, a list of data values, and a right bracket, followed by zero or more sets of left and right brackets with additional lists of data values. \*/

**void** DataStructures(): **{}**

**{**

<ARRAY> <ID> <COLON> <L\_BRACKET> **(** Data() **)\*** <R\_BRACKET>

**{** System.out.println("There is a data structure: Array");**}**

**|** <LINKED\_LIST> <ID> <COLON> <L\_BRACKET>

<L\_BRACKET> **(** Data()**)+**

<R\_BRACKET> **(**<L\_BRACKET> **(** Data()**)+** <R\_BRACKET>**)\***

<R\_BRACKET>

**{** System.out.println("There is a data structure: Linked List"); **}**

**}**

# **Appendix B: JJt Grammar**

/\*\*

\* CPCS302 Project Group6

\* Razan Alamri - Leen BA QALAQL - KHLOUD ALSOFYANI - Reema Almalki -RAWAN ALGHAMDI

\*/

**options**

**{**

**static** = **true**;

**}**

**PARSER\_BEGIN(MyNewGrammar)**

**package** JavaJJT;

**public** **class** MyNewGrammar

{

**public** **static** **void** main(String args [])

{

System.out.println("?????????????? Welcome To QMS Programming Language ??????????????");

System.out.print("Enter an expression like \"IF [\_grade\_>? 60] DO PRINT [\_G\_ =? 5].\" :");

**new** MyNewGrammar(System.in);

**try**

{

SimpleNode n = MyNewGrammar.Start();

n.dump(" .?");

System.out.println("Thank you for using QMS.");

}

**catch** (Exception e)

{

System.out.println("?Error.");

System.out.println(e.getMessage());

}}}

**PARSER\_END(MyNewGrammar)**

**SKIP** :

**{**

/\* White spaces \*/

**<** SPACE: " " **>**

**|** **<** TAB: "\t" **>**

**|** **<** NEWLINE:"\n" **>**

**|** "\r"

**}**

/\* Comments \*/

**SPECIAL\_TOKEN** : **{**

// (~["\n"])\*

**<** SINGLE\_LINE\_COMMENT: "??"**(~[**"\n"**])\*** **>**

**|** **<** MULTIPLE\_LINE\_COMMENT: "?\*"**(**< STRING >**)\***"\*?"**>**

**}**

/\* Keywords \*/

**TOKEN** : **{**

**<** IF: "IF" **>** **{** System.out.println("\n ? if statment ? "); **}**

**|** **<** DO: "DO" **>**

**|** **<** IF\_ELSE: "ELSEIF" **>{** System.out.println("\n ? else if statment ? "); **}**

**|** **<** OTHER: "OTHER" **>**

**|** **<** INTEGER\_TYPE: "INTEGER" **>{** System.out.println("\n ? Integer Number ? "); **}**

**|** **<** DECIMAL\_TYPE: "DECIMAL" **>** **{** System.out.println("\n ? DECIMAL Number ? "); **}**

**|** **<** BOOLEAN\_TYPE: "BOOLEAN" **>** **{** System.out.println("\n ? Boolean statment ? "); **}**

**|** **<** STRING\_TYPE: "STRING" **>**

**|** **<** LOOP: "LOOP?" **>**

**|** **<** ENDLOOP: "?END" **>**

**|** **<** PRINT: "PRINT" **>**

**|** **<** CONSTANT: "CONST" **>**

**|** **<** ARRAY: "?ARR" **>**

**|** **<** LINKED\_LIST: "?LL" **>**

**|** **<** EXIT: "EXIT"**>**

**{**System.out.println("?????????????? Thank You For Using Our Programming Language ??????????????");

System.exit(0); **}**

**}**

/\* Binary arithmetic operations \*/

**TOKEN** : **{**

**<** PLUS: "+?" **>** **{** System.out.println("\n ? binary arithmetic statment ? "); **}**

**|** **<** MINUS: "-?" **>** **{** System.out.println("\n ? binary arithmetic statment ? "); **}**

**|** **<** MULTIPLICATION: "\*?" **>** **{** System.out.println("\n ? binary arithmetic statment ? "); **}**

**|** **<** DIVIDE: "/?" **>** **{** System.out.println("\n ? binary arithmetic statment ? "); **}**

**|** **<** REMINDER: "%?" **>** **{** System.out.println("\n ? binary arithmetic statment ? "); **}**

**|** **<** ASSIGN: "=?" **>** **{** System.out.println("\n ? assignment statment ? "); **}**

**}**

/\* Unary arithmetic operations \*/

**TOKEN** : **{**

**<** INCREMENT: "++?" **>** **{** System.out.println("\n ? unary arithmetic statment ? "); **}**

**|** **<** DECREMENT: "--?" **>** **{** System.out.println("\n ? unary arithmetic statment ? "); **}**

**}**

/\* Relational operations \*/

**TOKEN** : **{**

**<** IS\_EQUAL: "==?" **>** **{** System.out.println("\n ? relational statment ? "); **}**

**|** **<** NOT\_EQUAL: "!?" **>** **{** System.out.println("\n ? relational statment ? "); **}**

**|** **<** GREATER\_THAN: ">?" **>** **{** System.out.println("\n ? relational statment ? "); **}**

**|** **<** LESS\_THAN: "<?" **>** **{** System.out.println("\n ? relational statment ? "); **}**

**|** **<** GREATER\_OR\_EQUAL: ">=?" **>** **{** System.out.println("\n ? relational statment ? "); **}**

**|** **<** LESS\_OR\_EQUAL: "<=?" **>** **{** System.out.println("\n ? relational statment ? "); **}**

**}**

/\* Logical operators \*/

**TOKEN** : **{**

**<** AND: "?AND?" **>** **{** System.out.println("\n ? logical statment ? "); **}**

**|** **<** OR: "?OR?" **>** **{** System.out.println("\n ? logical statment ? "); **}**

**|** **<** NEGATE: "?NOT?" **>** **{** System.out.println("\n ? logical statment ? "); **}**

**}**

/\* Punctuation Marks \*/

**TOKEN** : **{**

**<** DOT: "." **>**

**|** **<** COLON: ":" **>**

**|** **<** L\_BRACKET: "[" **>**

**|** **<** R\_BRACKET: "]" **>**

**|** **<** L\_PARENTHESIS: "(" **>**

**|** **<** R\_PARENTHESIS: ")" **>**

**|** **<** QMS: "?" **>**

**}**

/\* Identifiers \*/

**TOKEN** : **{**

**<** ID: "\_" **(** <LETTER> **|** <DIGIT> **|** <SYMBOL> **)+** "\_" **>**

**}**

/\* Data type \*/

**TOKEN** : **{**

**<** INTEGER: **(** <DIGIT> **)+** **>**

**|** **<** DECIMAL: **(** <DIGIT> **)+** "." **(** <DIGIT> **)+** **>**

**|** **<** BOOLEAN: **(** "T" **|** "F"**)** **>**

**|** **<** STRING: **(** <LETTER> **|** <DIGIT> **|** <SYMBOL> **)+** **>**

**}**

/\* Alphabets \*/

**TOKEN** : **{**

**<** #LETTER: **[**"a"**-**"z", "A"**-**"Z"**]** **>**

**|** **<** LETTERS: **([**"a"**-**"z", "A"**-**"Z"**])+** **>**

**|** **<** #DIGIT: **[**"0"**-**"9"**]** **>**

**|** **<** #SYMBOL: **[**"|", "&", ";", "#", "$", "@", ":", ",", "?"**]** **>**

**}**

/\* This method is starting point of grammar / the EXIT keyword to terminate the program. \*/

SimpleNode Start() :**{}**

**{**

Stmts() DOT()

**{**

**return** jjtThis;

**}**

**|** EXIT()

**{**

**return** jjtThis;

**}**

**}**

**void** EXIT () : **{** Token token; **}**

**{**

token = < EXIT > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

/\* This method defines a set of rules for constructing statements in the grammar. \*/

**void** Stmts() :**{}**

**{**

IFStmt()

**|** **LOOKAHEAD(**3**)** Assignment()

**|** Iteration()

**|** Array()

**|** LinkedList()

**|** CONSTANT()

**|** Print()

**|** **LOOKAHEAD(**3**)** Stmt()

**|** DataType()

**}**

**void** Assignment() : **{** **}** /\* This method recognizes an assignment statement,

which consists of an identifier followed by an assignment operator (=?) and a statement.\*/

**{**

ID () ASSIGN () Stmt()

**}**

**void** Stmt(): **{** **}**/\* This method recognizes a basic statement. \*/

**{**

**((**ID()**)** **|** Data()**|** BooleanStat()**)?**

**(**

ArithmeticStat()

**|** RelationalStat()

**|** LogicalStat()

**)?}**

**void** ArithmeticStat(): **{}** /\* recognizes an arithmetic statement,which consists of a binary or unary operator followed by an identifier or a number. \*/

**{**

**(**BinaryOP() **|** UnaryOP()**)** **(**ID() **|** Number()**)**

**}**

**void** BinaryOP(): **{}**// Binary arithmetic operator.

**{**

**(**MULTIPLICATION() **|** DIVIDE() **|** PLUS() **|** MINUS() **|** REMINDER() **|** ASSIGN() **)**

**}**

**void** PLUS () : **{** Token token; **}**

**{**

token = < PLUS > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** MINUS () : **{** Token token; **}**

**{**

token = < MINUS > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** MULTIPLICATION () : **{** Token token; **}**

**{**

token = < MULTIPLICATION > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** DIVIDE () : **{** Token token; **}**

**{**

token = < DIVIDE > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** REMINDER () : **{** Token token; **}**

**{**

token = < REMINDER > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** UnaryOP(): **{** **}**// Unary arithmetic operator.

**{**

**(**INCREMENT() **|** DECREMENT()**)**

**}**

**void** INCREMENT () : **{** Token token; **}**

**{**

token = < INCREMENT > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** DECREMENT () : **{** Token token; **}**

**{**

token = < DECREMENT > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** RelationalStat(): **{}** /\* recognizes a relational expression,which consists of a relational operator followedby an identifier or a number. \*/

**{**

RelationalOP() **(**ID() **|** Number()**)**

**}**

**void** RelationalOP(): **{}**// Relational operator.

**{**

IS\_EQUAL() **|** NOT\_EQUAL() **|** GREATER\_THAN() **|** LESS\_THAN() **|** GREATER\_OR\_EQUAL() **|** LESS\_OR\_EQUAL()

**}**

**void** CompareOP() : **{** **}**

**{**

LESS\_THAN()

**|** LESS\_OR\_EQUAL()

**|** GREATER\_THAN()

**|** GREATER\_OR\_EQUAL()

**|** IS\_EQUAL()

**|** NOT\_EQUAL()

**}**

**void** LESS\_THAN () : **{** Token token; **}**

**{**

token = < LESS\_THAN > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** LESS\_OR\_EQUAL () : **{** Token token; **}**

**{**

token = < LESS\_OR\_EQUAL > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** GREATER\_THAN () : **{** Token token; **}**

**{**

token = < GREATER\_THAN > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** GREATER\_OR\_EQUAL () : **{** Token token; **}**

**{**

token = < GREATER\_OR\_EQUAL > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** IS\_EQUAL () : **{** Token token; **}**

**{**

token = < IS\_EQUAL > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** NOT\_EQUAL () : **{** Token token; **}**

**{**

token = < NOT\_EQUAL > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** LogicalStat(): **{}** /\* recognizes a logical expression,which consists of a logical operator followed by a boolean value. \*/

**{**

LogicalOP() BOOLEAN()

**}**

**void** LogicalOP(): **{}**// Logical operator.

**{**

AND() **|** OR() **|** NEGATE()

**}**

**void** AND () : **{** Token token; **}**

**{**

token = < AND > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** NEGATE () : **{** Token token; **}**

**{**

token = < NEGATE > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** OR () : **{** Token token; **}**

**{**

token = < OR > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** BooleanStat(): **{}** /\* recognizes a boolean statement,which consists of a boolean type followed by an identifier and a relational expression. \*/

**{**

BOOLEAN\_TYPE() ID() ASSIGN() **(**ID() RelationalOP() ID() **)**

**}**

**void** Data(): **{}** /\* recognizes a data value,which can be an integer, decimal, string, or boolean. \*/

**{**

INTEGER() **|** DECIMAL() **|** STRING() **|** BOOLEAN()

**}**

**void** Number(): **{** **}**// Number is INTEGER or DECIMAL.

**{**

INTEGER() **|** DECIMAL()

**}**

**void** DataType () : **{** **}**/\* recognizes a data type,which can be an integer, decimal, string, or boolean. \*/

**{**

INTEGER\_TYPE() **|** DECIMAL\_TYPE() **|** BOOLEAN\_TYPE()**|** STRING\_TYPE()

**}**

**void** Iteration(): **{** **}** /\* This method recognizes an iteration statement,

which consists of a loop keyword followed by a condition in brackets, a colon, and a set of statements.

It also recognizes the END keyword to terminate the loop. \*/

**{**

LOOP() L\_BRACKET() Condition() R\_BRACKET() COLON() Stmts() ENDLOOP()

**}**

**void** LOOP () : **{** Token token; **}**

**{**

token = < LOOP > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** ENDLOOP () : **{**Token token; **}**

**{**

token = < ENDLOOP > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** CONSTANT(): **{** **}**

**{**

QMS() Constant() DeclareVar()

**}**

**void** Constant () : **{** Token token; **}**

**{**

token = < CONSTANT > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** IFStmt(): **{}**

**{**

IF() L\_BRACKET() Condition() R\_BRACKET() DO() PRINT() L\_BRACKET() Stmts() R\_BRACKET()

**(**OTHER() PRINT() L\_BRACKET() Stmts() R\_BRACKET()**)?**

**}**

**void** IF () : **{** Token token; **}**

**{**

token = < IF > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** OTHER () : **{** Token token; **}**

**{**

token = < OTHER > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** DO () : **{** Token token; **}**

**{**

token = < DO > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** Condition(): **{}**

**{**

**((**ID() **|** Number()**)** RelationalOP() **(**ID() **|** Number()**)** **)**

**}**

**void** Array() : **{** **}**

**{**

ARRAY() ID() COLON() L\_BRACKET() **(** Data() **)\*** R\_BRACKET()

**}**

**void** ARRAY () : **{** Token token; **}**

**{**

token = < ARRAY > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** LinkedList() : **{** **}**

**{**

LINKED\_LIST() ID() COLON() L\_BRACKET() L\_BRACKET() **(** Data()**)+**

R\_BRACKET() **(**L\_BRACKET() **(** Data()**)+** R\_BRACKET()**)\*** R\_BRACKET()

**}**

**void** LINKED\_LIST () : **{** Token token; **}**

**{**

token = < LINKED\_LIST > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** DeclareVar() : **{** **}**

**{**

INTEGER\_TYPE() ID() ASSIGN() INTEGER()

**|** DECIMAL\_TYPE() ID() ASSIGN() DECIMAL()

**|** STRING\_TYPE() ID() ASSIGN() STRING()

**|** BOOLEAN\_TYPE() ID() ASSIGN() BOOLEAN()

**}**

**void** ID () : **{** Token token; **}**

**{**

token = < ID > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** INTEGER\_TYPE () : **{** Token token; **}**

**{**

token = < INTEGER\_TYPE > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** DECIMAL\_TYPE () : **{** Token token; **}**

**{**

token = < DECIMAL\_TYPE > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** BOOLEAN\_TYPE () : **{** Token token; **}**

**{**

token = < BOOLEAN\_TYPE > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** STRING\_TYPE () : **{** Token token; **}**

**{**

token = < STRING\_TYPE > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** INTEGER () : **{**Token token; **}**

**{**

token = < INTEGER > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** DECIMAL () : **{** Token token; **}**

**{**

token = < DECIMAL > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** BOOLEAN () : **{** Token token; **}**

**{**

token = < BOOLEAN > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** STRING () : **{** Token token; **}**

**{**

token = < STRING > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** L\_BRACKET () : **{** Token token; **}**

**{**

token = < L\_BRACKET > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** R\_BRACKET () : **{** Token token; **}**

**{**

token = < R\_BRACKET > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** L\_PARENTHESIS () : **{** Token token; **}**

**{**

token = < L\_PARENTHESIS > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** R\_PARENTHESIS () : **{** Token token; **}**

**{**

token = < R\_PARENTHESIS > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** QMS () : **{**Token token; **}**

**{**

token = < QMS > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** COLON () : **{**Token token; **}**

**{**

token = < COLON > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** Print() : **{** **}**

**{**

PRINT() L\_BRACKET() **(**STRING()**)+** R\_BRACKET()

**}**

**void** PRINT () : **{** Token token; **}**

**{**

token = < PRINT > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** DOT () : **{** Token token; **}**

**{**

token = < DOT > **{** jjtThis.jjtSetValue(token.image); **}**

**}**

**void** ASSIGN () : **{**Token token; **}**

**{**

token = < ASSIGN > **{** jjtThis.jjtSetValue(token.image); **}**

**}**