International Journal of Computer Theory and Engineering, Vol.3, No.1, February, 2011
1793-8201

Relational Algebra Interpreter in Context of Query Languages

Anshu Ranjan, Ratnesh Litoriya

***Abstract*—Relational database systems have succeeded commercially because of their openness and sturdy theoretical groundwork. The contribution of this title "Relational Algebra Interpreter in context of query languages" is presentation of new implementation in such a way so that queries written in relational algebra can be compiled into SQL and executed on a relational database system. It takes a relational algebra statement as key, does syntactic and lexical parsing on it. In the event of an error in the syntax of the expression it will forward the error to user. If the syntax is correct the relational algebra expression is converted into a SQL statement and executed on an RDBMS. This work can serve up as a basis learning Relational Algebra for different class of users, as they will be given immediate feedbacks about their queries.**

***Index Terms*—Relational Algebra; structured query language; parser; interpreter**

I. Introduction

Before Dr. E. F. Codd's seminal paper on relational data banks, managing large amounts of data was a cumbersome process. Dr. Codd's work[1] suggested storing the data in a set of relations and manipulating it using relational algebra and calculus. All major database management systems in the market today are based on the relational model specified in his paper. The relational database management systems (RDBMSs) all use structured query language (SQL) to manipulate the data. SQL itself builds on relational algebra underpinning. Relational algebra is a closed algebra which takes relations as input and produces relations as output. It defines many operators such as join, project and select on the relations that it operates on.

A Relational Algebra Interpreter is presented in this paper. Implementation takes a relational algebra statement as input, does syntactic and lexical parsing on it. In the event of an error in the syntax of the expression it will forward the error to user. If the syntax is correct the relational algebra expression is converted into a SQL statement and executed on an RDBMS. JLex and JCup can be used for the purpose of lexical and syntactic analysis of the input query.

Anshu Ranjan, Computer Information Science and Engineering, University of Florida, Gainesville, USA (email: anshu_martin@yahoo.co.in).

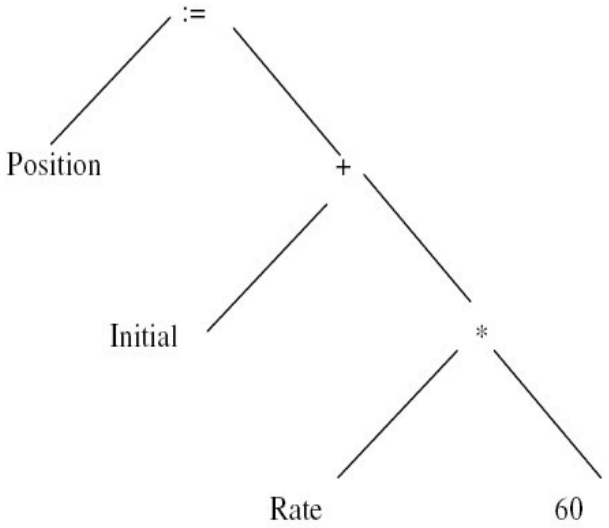
Ratnesh Litoriya , Computer Science and Engineering , Jaypee Institute of Engineering and Technology, Guna, India (email: ratneshlitoriya@yahoo.com).

II. Overview of the Interpreter

An interpreter means a computer program that executes, i.e. performs, instructions written in a programming language. An interpreter may be a program that either [I]

- Executes the source code directly
- Translates source code into some efficient intermediate representation (code) and immediately executes this.
- Explicitly executes stored precompiled code made by a compiler which is part of the interpreter system.

Unlike compilers, instead of producing a target program as a translation, an interpreter performs the operations implied by the source program. For as assignment statement, for example, an interpreter might build a tree like Fig. 1, and then carry out the operations at the nodes as it "walks" the tree. At the root it discover it had an assignment to perform, so it would call a routine to evaluate the expression on the right, and then store the resulting value in the location associated with the identifier position. At the right child of the root, the routine would discover it had to compute the sum of two expressions. It would call itself recursively to compute the value of the expression rate*60. It would then add that value to the value of the variable 'initial'. [3]



The different phases in designing an interpreter are:

- Lexical Analyzer: - in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.
- Syntax Analysis, which involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
- Code Generation, in which the interpreter generates a

target code, which can be executed directly on the machine.

- Execution, which involves executing a statement which is lexically and syntactically correct and converted into a target language.
- Error Handler, which is concerned with detecting and reporting lexical or syntactic errors.

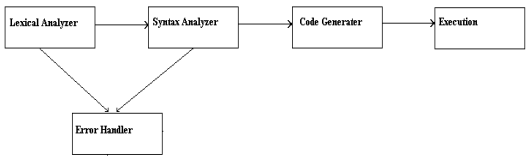


Fig.2.Phases of the interpreter

III. The Design

A. Data Flow Diagram

The figure below shows an elaborate view of the interpreter. Its explanation is provided thereafter.

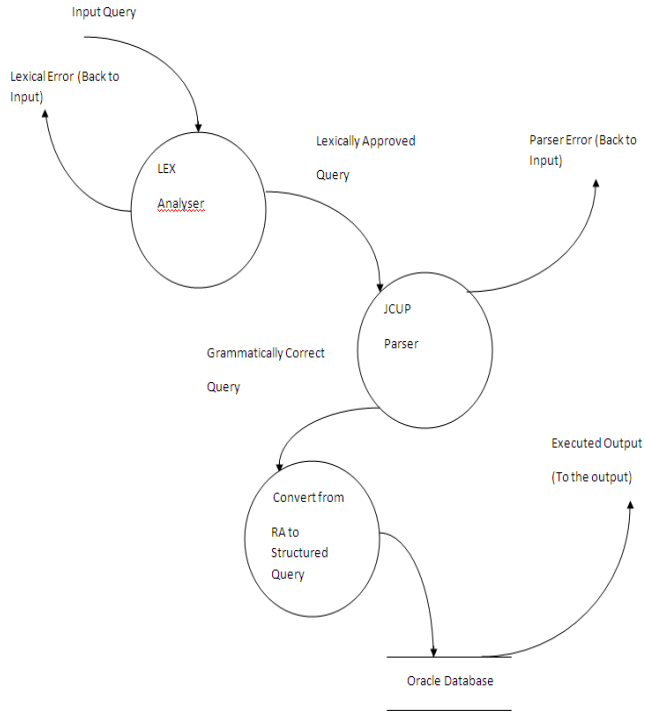


Figure 3: Level 1 DFD

- First, a Relational Algebra query serves as an input to the lexical analyser *i.e.* JLex.
- JLex scans the particular, divides it into various lexemes, and checks whether the query matches the concerned regular expressions or not.
- If an error is found in this phase, then the error is reported and the command prompt gets ready for another input.
- If there is no error in the lexical phase then the query is passed to the parser tool which is JCUP.
- JCUP analyses the input query and checks whether the query follows a CFL or not.
- If it does not follow the grammatical rules, error is reported and the control is back to the command prompt.
- If query is correct grammatically, then query is sent for conversion from Relational Algebra to Structured Query Language.
- After converting the query to SQL, the query is executed on an Oracle Database and the output is shown in the command window.

B. Use Cases

There are three use cases in the proposed software as shown in Fig. 3 and are explained below.

Use Case 1: Check the Syntax of an Input Query

- Write the query in the first text box.
- Click on the button 'Check Syntax' to check the correctness of the query.
- If the query is correct a message box with message 'Correct Query' is shown else a message box with message 'Incorrect Query' along with error location and type is displayed.

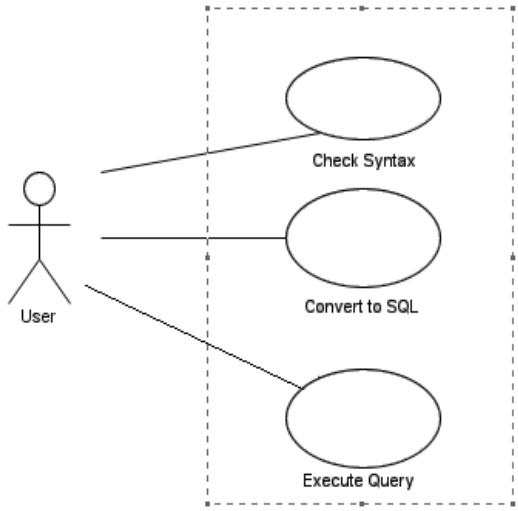


Figure 4: Use Case Diagram

Use Case 2: Convert the input query to SQL

- First check the correctness of the query by following steps in Use Case 1.
- If the query is correct, then click the 'Convert to SQL' button.
- The converted query is displayed in the second text box.

Use Case 3: Execute the query

- First follow the steps given in Use Case 1 and 2.
- Click the button 'Execute the query'.
- The output is shown in a separate window.

For a better understanding of the flow of control, the flowchart diagram is shown in Fig. 4.

C. Design for lexical analyzer

JLex, a lexical analyzer generator in Java, is proposed to be used for implementing phase of lexical analysis in the interpreter. The first phase of compilation is lexical analysis - the decomposition of the input into tokens. A token is usually described by an integer representing the kind of token, possibly together with an attribute, representing the value of the token. The lexical analyzer copes with text that may not

be lexically valid by producing an error message.

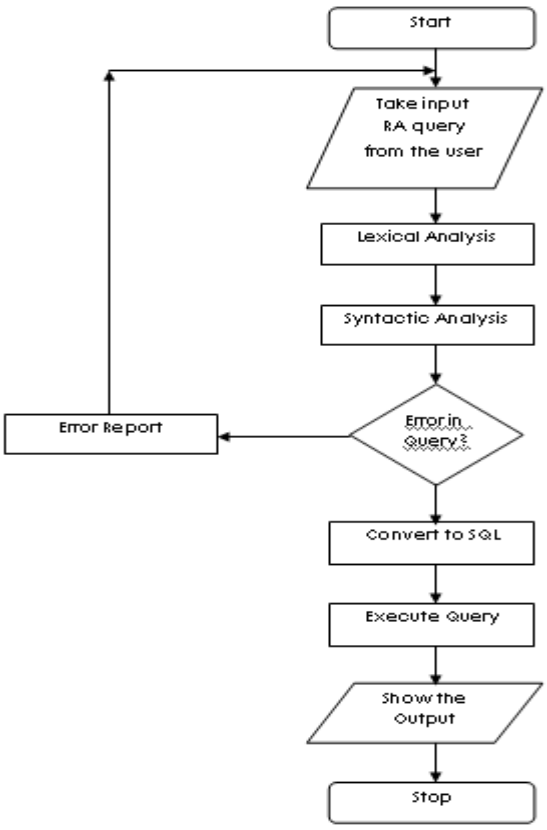


Figure5:.Flowchart

A JLex input file is organized into three sections, separated by double-percent directives (``%%"). A proper JLex specification has the following format. [II]

user code
%%
JLex directives
%%
regular expression rules

In the second section, various state names and macros could be declared viz. keywords like project, rename, select, join, intersect, minus, times etc, symbol sets like digit(1 to 9), letters(a-z,A-Z), comparison operators (<,>,=), whitespace(/n,/t,/b), bar brackets([,]), semicolon(;), comma(,), quotes(‘) etc…

In the third section, the action sequence in java code will be specified when the lexical analyzer encounters a particular token. As for example if a token matching the keyword ‘project’ is encountered, then the symbol corresponding to the state which is 3 is returned. Similarly, all the other tokens are duly taken care of.

D. Design for parser

CUP, a system for generating LALR parsers from simple specifications written in Java, is proposed to be used for parsing. Using CUP involves creating a simple specifications based on the grammar for which a parser is needed, along with construction of a scanner capable of breaking characters up into meaningful tokens (such as keywords, numbers, and special symbols), which could be done using JLex in my case. The specification contains three main parts. The first part provides preliminary and miscellaneous declarations to specify how the parser is to be generated, and supply parts of the runtime code. The second part of the specification declares terminals and non terminals, and associates object classes with each. Here, WEwill illustrate the grammar of RA to be used. [III]

Query ::= Expr SEMI | error;
Expr ::= ProjExpr | SelectExpr | RenameExpr | UnionExpr | MinusExpr | IntersectExpr | JoinExpr | TimesExpr |STR;
ProjExpr ::= PROJECT [AttrList] (Expr);
RenameExpr ::= RENAME [AttrList] (Expr);
AttrList ::= STR | AttrList , STR;
UnionExpr ::= (Expr UNION Expr);
MinusExpr ::= (Expr MINUS Expr);
IntersectExpr ::= (Expr INTERSECT Expr);
JoinExpr ::= (Expr JOIN Expr);
TimesExpr ::= (Expr TIMES Expr);
SelectExpr ::= SELECT [Condition] (Expr);
Condition ::= SimpleCondition | SimpleCondition AND Condition;
SimpleCondition ::= Operand COMP Operand;
Operand ::= STR | ‘ STR ‘ | NUMBER;

As, it can be seen above, the terminals were AND, PROJECT, STR, QUOTE, RENAME, TIMES, SELECT, BAROPEN, BARCLOSE, JOIN, COMMA, SEMI, MINUS, BRACOPEN, INTERSECT, BRACCLOSE, NUMBER, COMP, UNION and the non-terminals were Query, Expr, ProjExpr, Operand, SimpleCondition, AttrList, SelectExpr, Condition, RenameExpr, UnionExpr, MinusExpr, IntersectExpr, JoinExpr, TimesExpr, which constitute the second part of the specification. The final part of the specification contains the grammar shown above.

E. Algorithm for converter

To convert a Relational Algebra Query into a Structured Query Language Query, all the keywords in RA are needed to be separately taken care of. Now, each of the keywords will follow the steps as shown below.

1) Project:-

a) Replace ‘project’ with ‘select’.

b) Remove the bar brackets enclosing the attributes.

c) Add the word ‘from’ after the attributes.

d) If before the next ‘project’ and ‘rename’ token a tokens of the order “(“, “<string>”, “)” is found, then remove them from the original string and append the ‘<string>’ after ‘from’. Otherwise don’t take any action and exit.

2) Rename

a) If the ‘rename’ token is not followed by ‘project’ token before the end of rename statement, then replace it with ‘project’ token and follow the procedures as those of ‘project’ and exit. Otherwise, do the following.

b) Insert next to all the occurrences of attributes mentioned after “project” token the corresponding attributes after “rename” token.

c) Remove the tokens of “rename” and all the attributes after it.

3) Select

a) Replace 'select' token with 'where'.

b) Remove the bar brackets enclosing the condition after that.

4) Join

Replace the 'join' token with ') natural join ('. This will facilitate execution of natural join command in Oracle 10g.

5) Minus or Times or Union or Intersect

Replace the token with ") <current token> (".

IV. Implementation

Class Diagram

As shown below in Fig. 6, the project includes five classes, each of which have been briefly described below.

User Interface: RAI class is responsible for creating a user interface using Java Swing. It consists of two text boxes, one for accepting input from user and other for showing the converted SQL query. The user can check the syntax, convert input query from RA to SQL, execute queries on a database and empty the contents of the text boxes using concerned buttons of the interface.

Syntax Analyzer: JLex and JCup are used to make the parser. The design of the input to these has already been explained in the design phase.

Converter: The class RAtoSQL converts the input RA query to SQL using various methods of the class ToolsforConversion.

Execute: This class is responsible for connecting the application with a database and execute the SQL query on it.
USER INTERFACE

We have used JFrame for constructing the Interface. The main class inherits from the class JFrame. At the beginning the constructor of parent class is called. As shown in the figure 6, following are the main components of the user interface:

i) Text Boxes: The text box on the top is to accept an input query from the user in RA. The second text box would contain the converted SQL query, once the user supplies a correct input and clicks on the button 'Convert to SQL'.

ii) Buttons: Four buttons are used in the interface namely Check Syntax, Convert to SQL, Execute Query and Reset.

- Check Syntax: Checks whether the input query is lexically and syntactically correct or not. If the query is correct then a message box showing the message "Correct Query" is displayed. If the query is incorrect, then a message showing the message "Incorrect Query", along with the location and type of error is displayed.
- Convert to SQL: This button converts a correct query to SQL and shows the output in the second text box.
- Execute Query: Executes the SQL query on a database. The output is shown in another window.
- Reset: Clears the text in the two text boxes.

Following are the methods used in the module along with their brief descriptions:

i) main (): This method first makes certain adjustments in the graphs of the interface. Then it calls the constructor RAI ().

ii) RAI (): This constructor calls the constructor of the parent class. Then it calls the method initializeComponent (). Ultimately it sets the value true to the setVisible property of the current object.

iii) initializeComponent (). This method is responsible for creating the whole of user interface. It creates all the text boxes, buttons and labels with appropriate captions. It also adds the ActionListener property to the buttons so that the button would respond on being clicked.

iv) Button1_actionPerformed (ActionEvent e): This method is called once the button with caption "Check Syntax" is clicked. It calls the method lexcheck ().

v) Button2_actionPerformed (ActionEvent e): This method is called once the button with caption "Convert to SQL" is clicked. It calls the method convert () of class converter passing the input query in first text box as a parameter.

vi) Button3_actionPerformed (ActionEvent e): This method is called once the button with caption "Execute Query" is clicked. It calls the main method of class SimpleOraJava passing the input query in first text box as a parameter.

vii) Button4_actionPerformed (ActionEvent e): This method is called once the button with caption "Execute Query" is clicked. It clears the text of the two text boxes.

viii) lexcheck(): This method collects the input query in a variable and send it for lexical and syntactic analysis by calling the methods of classes Yylex and parser.

ix) RAI (int): This constructor contains no code. It is used by the error detecting module of the project for the purpose of declaring a reference variable of this class.

A. Input to Lexical Analyzer

We have used JLex for the purpose of lexical analysis of the input query. **JLex** is a lexical analyzer generator (also known as scanner generator) for Java, written in Java. The first phase of compilation is lexical analysis - the decomposition of the input into tokens. A token is described by an integer representing the kind of token, possibly together with an attribute, representing the value of the token. The lexical analysis copes with text that may not be lexically valid by producing an error message.

A JLex input file is organized into three sections, separated by double-percent directives (``%%"). A proper JLex specification has the following format.

User code
%%
JLex directives
%%
regular expression ***rules***

The ``%%" directives distinguish sections of the input file and must be placed at the beginning of their line. The remainder of the line containing the ``%%" directives may be discarded and should not be used to house additional declarations or code. The user code section - the first section

of the specification file - is copied directly into the resulting output file. This area of the specification provides space for the implementation of utility classes or return types. The JLex directives section is the second part of the input file. Here, macros definitions are given and state names are declared. The third section contains the rules of lexical analysis, each of which consists of three parts: an optional state list, a regular expression, and an action. Next, WE will describe my input to JLex.

User code

Only one line was written in this part.import java_cup.runtime.Symbol; /* so that the there is no problem in running the code which We will write in the section of regular expression rules */

JLex directives

In this part, the following macros were declared:

1. digit – It contains all the digits from 0 to 9.
2. COMP – It contains the three operators '<', '>' and '=.
3. NUMBER – It contains all the possible combinations of digits 0 to 9.
4. BAROPEN – It contains the character '['.
5. BARCLOSE – It contains the character ']'.
6. whitespace – It contains the escape sequences '\t\n\t\f' and a space ' '.
7. letters – It contains all letters in both caps, the digits and the symbols '_', '.', '*' and ','.
8. PROJECT – It contains the keyword 'project'.
9. SELECT – It contains the keyword 'select'.
10. SEMI – It contains the symbol ';'.
11. BRACOPEN – It contains the symbol '('.
12. BRACCLOSE – It contains the symbol ')'.
13. COMMA – It contains the symbol ','.
14. RENAME – It contains the keyword 'rename'.
15. UNION – It contains the keyword 'union'.
16. INTERSECT – It contains the keyword 'intersect'.
17. MINUS – It contains the keyword 'minus'.
18. JOIN – It contains the keyword 'join'.
19. TIMES – It contains the keyword 'times'.
20. STR – It contains all the possible sequences of letters described in the macro 'letters'.
21. QUOTE – It contains the symbols of quotations.
22. AND – It contains the keyword 'and'.

B. Regular Expression Rules

This section tells the analyzer that what action it should take once it encounters a particular token in the input query. If it counters one of the macros: MINUS, JOIN, TIMES, SELECT, COMMA, BAROPEN, BARCLOSE, PROJECT, BRACOPEN, BRACCLOSE, SEMI, or a combination of macros COMP and letters, it returns a number corresponding to each of them as specified in the symbol table which is made by the analyzer itself. Example of the specification follows:

```
{AND} {return new Symbol(sym.AND);}
{SEMI} {return new Symbol(sym.SEMI);}
{COMP}+ {return new Symbol(sym.COMP);}
```

Thus, the input file was saved with *.lex* extension. Now, it was time to make the input file to JCup.

V. Input to Parser

We have used a java tool JCup for the purpose of parsing. **JCup** i.e. Java Based Constructor of Useful Parsers (CUP for short) is a system for generating LALR (Look Ahead Left to Write) parsers from simple specifications. Using CUP involves creating a simple specifications based on the grammar for which a parser is needed, along with construction of a scanner capable of breaking characters up into meaningful tokens (such as keywords, numbers, and special symbols). A CUP specification has three main parts. The first part provides preliminary and miscellaneous declarations to specify how the parser is to be generated, and supply parts of the runtime code. In this case we indicate that the java_cup.runtime and import javax.swing.* classes should be imported, and then supply a small bit of initialization code, and some code for invoking the scanner to retrieve the next input token. The second part of the specification declares terminals and non terminals, and associates object classes with each. In this case, we declare our terminals as being represented at runtime by two object types: token and int_token (which are supplied as part of the CUP runtime system), while various non terminals are represented by objects of types symbol and int_token (again supplied from the runtime system). The final part of the specification contains the grammar.

A. Initialization Code

Initially, some initialization code is added for the purpose of error detection and reporting. This is done by enclosing the code in parser code declaration as it allows methods and variable to be placed directly within the generated parser class. Following methods and global variables are placed in the code:

i) query: this variable stores the input query.

ii) errortype: This variable stores the number corresponding to the type of error encountered. Following are the error numbers and types

Error Number	Type
1	Expecting open bar bracket '['
2	Bar bracket not closed
3	Expecting open bracket '('
4	Invalid syntax of 'project'
5	Expecting a bracket or semicolon
6	Invalid syntax of 'rename'
7	Invalid syntax of 'union'
8	Invalid syntax of 'minus'
9	Invalid syntax of 'intersect'
10	Invalid syntax of 'join'
11	Invalid syntax of 'times'
12	Invalid syntax of 'select'

iii) ci: This variable stores the index value of the currently scanning token of the input query.

iv) thequery (): This method first declares a reference object variable of the class RAI. After that, it receives the input query of the user using the static variable of the class 'query' and subsequently stores in the global variable of this module called 'query'.

v) setci (char): This method sets the variable ci to the index after the first occurrence of the character passed as the parameter in the input query after the current ci.

vi) setci (string): This method sets the variable ci to the index after the first occurrence of the string passed as the parameter in the input query after the current ci.

vii) setzero: It sets the value of global variables ci and errortype to zero.

viii) settype (int): It assigns the type of error which is passed as a parameter to the variable errorype.

ix) displayerror(): It identifies the type of error encountered and displays it on a message box along with the query.

B. Terminals and Nonterminals

Following terminals and non terminal were declared:

Terminals: AND, PROJECT, STR, QUOTE, RENAME, TIMES, SELECT, BAROPEN, BARCLOSE, JOIN, COMMA, SEMI, MINUS, BRACOPEN, INTERSECT, BRACCLOSE, NUMBER, COMP, UNION

Non Terminals: Query, Expr, ProjExpr, Operand, SimpleCondition, AttrList, SelectExpr, Condition, RenameExpr, UnionExpr, MinusExpr, IntersectExpr, JoinExpr, TimesExpr

Note that the terminals were declared using large caps while the non terminals were declare in small caps.

C. The Grammar

The grammar specification is same as described in the previous chapter and as shown below:

Query ::= Expr SEMI | error;

Expr ::= ProjExpr | SelectExpr | RenameExpr | UnionExpr | MinusExpr | IntersectExpr | JoinExpr | TimesExpr |STR;

ProjExpr ::= PROJECT BAROPEN AttrList BARCLOSE BRACOPEN Expr BRACCLOSE;

RenameExpr ::= RENAME BAROPEN AttrList BARCLOSE BRACOPEN Expr BRACCLOSE;

AttrList ::= STR | AttrList COMMA STR;

UnionExpr ::= BRACOPEN Expr UNION Expr BRACCLOSE;

MinusExpr ::= BRACOPEN Expr MINUS Expr BRACCLOSE;

IntersectExpr ::= BRACOPEN Expr INTERSECT Expr BRACCLOSE;

JoinExpr ::= BRACOPEN Expr JOIN Expr BRACCLOSE;

TimesExpr ::= BRACOPEN Expr TIMES Expr BRACCLOSE;

SelectExpr ::= SELECT BAROPEN Condition BARCLOSE BRACOPEN Expr BRACCLOSE;

Condition ::= SimpleCondition | SimpleCondition AND Condition;

SimpleCondition ::= Operand COMP Operand;

Operand ::= STR | QUOTE STR QUOTE | NUMBER;

Still this is not the exact input in this part. Many codes were added in between the grammar specification for identification and reporting of errors. For example if the user forgets to type opening bar bracket '[' after the keyword 'project', it will encounter the following code:

```
/*----------
ProjExpr ::= {: parser.setci("project"); System.out.println("current="+parser.ci); parser.settype(1);:}PROJECT BAROPEN {: parser.setci('['); System.out.println("current="+parser.ci); parser.settype(2);:}AttrList {: parser.setci(']'); System.out.println("current="+parser.ci); parser.settype(3);:} BARCLOSE BRACOPEN {: parser.setci('('); System.out.println("current="+parser.ci); parser.settype(4);:}Expr {: parser.setci(')'); System.out.println("current="+parser.ci); parser.settype(5);:} BRACCLOSE ;
----------*/
```

As it can be seen that as soon as the parser sees the keyword 'project', it sets the error type to 1 since if at this stage there is an error in parsing, it means that there is a missing opening bar bracket. Hence as soon as the error types is saved and the parsing is terminated at the following statement:

```
/*---------------
Query ::= {: parser.thequery(); :}Expr SEMI {: JOptionPane.showMessageDialog (null,"Correct Query!"); parser.setzero();:} | error {:System.out.println(parser.ci);parser.displayerror(); parser.setzero(); :} ;
---------------*/
```

Here, we see that the flow of control of the parser is at the error part. Here it calls the displayerror() method which has been explained previously. Hence the error message box appears along with the error location and type. Ultimately all the global variable are set to zero so that the error code would work properly the next time.

Similarly, codes have been added to each line of the grammar to specify error type and set the value of ci.

VI. Conclusion and Future Extensions

A. Conclusion

During the course of making this project, the gain in terms of experience and knowledge is exemplary and should be highly useful to me in the long run. The gamut of learning covers UML diagrams, JAVA programming, designing algorithms and research works to name a few. We especially enjoyed exploiting the vast number of classes in java API while coding for conversion of query from RA to SQL.

Though WEhave tried to make my system as user-friendly and reliable as possible, it has certain limitations as explained in the following section.

B. Limitations

i) 'Rename' only supports renaming of attributes; it does not support renaming of the input relation.

ii) The inner join operator is not supported.

iii) It does not support any function other than the primitive operations of relational algebra like aggregate function viz. count etc.

C. Relevance in the present scenario

Databases are the focus of most introductory courses on database management systems. The formal relational query languages like relational algebra are therefore an important part of the curriculum. As a student, it is difficult to know

whether your queries expressed on paper in the formal languages are correct. As an instructor, it is often difficult to grade creative queries, especially those that have not been verified by an educational tool. The goal of the Relational Algebra Interpreter is to provide a mechanism by which the students can explore the formal relational query languages, getting immediate feedback by seeing the answers to the posed query. Consequently, the grading process is usually eased since the students can submit verified answers on homework assignments.

D. Future Extensions

Though the project is complete in itself and satisfies all the objectives encompassed by the scope, there are many enhancements which can be made in it. Some of these plausible up gradations are listed below:

i) Connecting with the database: The software will allow the user to choose the database on which the query is to be executed. This can be done by changing the URL while connecting with a database. In this way, the query could be executed on access, mySQL or any other database.

ii) Taking RA file as input: The software will take a file containing a series of RA queries and execute them sequentially. The output can be shown on a separate file.

iii) Create a database: The user can create his own database in which he makes various relations and perform relational algebra operations on them.

iv) Delete or modify relations: The user can edit the relations present in the database.

REFERENCES

[1] http://en.wikipedia.org/wiki/Interpreter_(computing)
[2] http://www.pasteur.fr/formation/infobio/python/ch05s02.html
[3] Compilers: Principles, Techniques, and Tools by Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
[4] http://www.databasteknik.se/webbkursen/relalg-lecture/index.html
[5] http://www.cs.sfu.ca/CC/354/zaiane/material/notes/Chapter3/node8.html
[6] http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf
[7] http://tinman.cs.gsu.edu/~raj/8710/f05/RA.pdf
[8] ftp://ftp.eas.asu.edu/pub/class/cse412/publications/WinRDBI-SIGCSE97.pdf
[9] http://infolab.stanford.edu/~widom/cs145/ra.html
[10] http://bmrc.berkeley.edu/courseware/cs164/spring98/proj/jlex/manual.html
[11] http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/manual.v0.9e.html
[12] www2.in.tum.de/projects/cup/manual.html
[13] http://www.developer.com/design/article.php/2206791
[14] http://ce.sharif.edu/~b_takhtaei/scannerparser.html
[15] http://www.cs.duke.edu/courses/fall06/cps116/faq-ra.html
[16] Head First Java by Kathy Sierra
[17] The Complete Reference Java2 by Herbert Schildt
[18] Theory of Computer Science by K.L.P. Mishra and N. Chandrasekaran
[19] Database System Concepts by Silberschatz, Korth, Sudarshan