IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 11, November 2015

Natural Language Database Interface

Asst. Prof. Rakhee Kundu[1], Asst. Prof. Poonam Gholap[2], Asst. Prof. Snehal Mane[3]

Computer Engineering, VESIT, Affiliated to Mumbai University, India [1, 2, 3]

Abstract: Databases have become ubiquitous. Almost all IT applications are storing and retrieving information from databases. Retrieving information from the database requires knowledge of technical languages such as Structured Query Language (SQL). However majority of the users who interact with the databases do not have a technical background and are intimidated by the idea of using languages such as SQL. This has led to the development of a few Natural Language Database Interfaces (NLDBI). A NLDBI allows the user to query the database in a natural language (NL). This dissertation work highlights on architecture of new NLDBI system, which includes designing a grammar which convert NL statement to a machine understandable language like a query which is fired on a database, constructing parse tree/s and analyzing them. In most of the typical NLDBI systems the NL statement is converted into an internal representation based on the syntactic and semantic knowledge of the NL. This representation is then converted into queries using a representation converter. Before a NL query is translated to an equivalent query in technical language like SQL it has to go through various steps. In this paper it highlights the steps of speech tagging followed by tagging of each word of the query, parsing the tagged sentence by a grammar and generating a grammar tree (parse tree) by applying the semantic analysis on that parse tree and finally SQL translator processes the parse tree to obtain the SQL query.

Keywords: Natural Language, Database, SQL Query, Speech tagging, Parse tree.

I. INTRODUCTION

Databases have become ubiquitous. Almost all IT applications are storing and retrieving information from databases. Retrieving information from the database requires knowledge of technical languages such as Structured Query Language (SQL). However majority of the users who interact with the databases do not have a technical background and are intimidated by the idea of using languages such as SQL. This has led to the development of a few Natural Language Database Interfaces (NLDBI). A NLDBI allows the user to query the database in a natural language (NL). This dissertation work highlights on architecture of new NLDBI system, which includes designing a grammar which convert NL statement to a machine understandable language like a query which is fired on a database, constructing parse tree/s and analyzing them. In most of the typical NLDBI systems the NL statement is converted into an internal representation based on the syntactic and semantic knowledge of the NL. This representation is then converted into queries using a representation converter. Before a NL query is translated to an equivalent query in technical language like SQL it has to go through various steps. This dissertation work highlights the steps of speech tagging followed by tagging of each word of the query, parsing the tagged sentence by a grammar and generating a grammar tree (parse tree) by applying the semantic analysis on that parse tree and finally SQL translator processes the parse tree to obtain the SQL query.

II. PROBLEM DEFINITION AND METHODOLOGY ADAPTED

Natural language computing (NLC) is becoming one of the most active areas in Human-Computer Interaction. The goal of NLC is to enable communication between people and computers without resorting to memorization of complex commands and procedures. In other words, NLC is a technique which can make the computer understand the languages naturally used by humans, but not by artificial or man-made language such as a programming language. This paper describes a NL interface that supports complex queries based on a grammar to relational database

A. Formalization of the Problem

The query to databases in NL is a very convenient and easy method of data access, especially for those persons who do not have a technical background of database query languages such as SQL. The experimental work is based on a unique concept of processing user NL statement, converting into a technical form so as to access the data from relational data storage, to generate unambiguous results. NLDBI is a system that allows users to access a database in NL and has been a popular field of study. The user has to access database in NLs. The attempt in the present work is to create simple reliable NL interface to relational databases.

If we consider an employee database and if the person wants to find his salary, he queries on to the database to get suitable results. A common man can have variety of NL queries like:

- What is the salary of Nikhil Karande?
- What is the salary of id 123?
- What is the salary of employee with id 123 and the name Nikhil Karande?
- What is the salary of employee with id 123?
- What is the earning of employee id 123?

B. Techniques and Methodologies:

NLDBI is a system that allows users to access a database in NL. Suppose we consider a properly normalized database. Now if the user wishes to access the data from

DOI 10.17148/IJARCCE.2015.41143

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 11, November 2015

the table, he/she accesses the tables in his/her language. The different techniques and methodologies discussed below

i) Domain Class Dictionary:
In practice, a particular database structure is created from a certain conceptual model about real application domain. This conceptual model is well known to database communities as a semantic data model (SDM) which is today used in any medium-sized legacy database creation. SDM is structurally almost transparent to a physical database structure, since the two structures can be transformed into each other automatically. In addition, each component of SDM is annotated with some linguistic descriptions. There are two types of linguistic descriptions in SDM, domain surrogates and domain propositions. Domain surrogates are noun phrases for naming each SDM component, such as 'customer', or 'date of order'. Domain propositions are definitional information about a certain domain reality, expressed by one NL sentence [14]. These linguistic annotations of SDM have the potential to conceptually bridge a NL question and a target database structure. Based on SDM, this section describes a linguistically motivated database semantics representation (LDBS), which consists of domain class dictionary and domain thematic frames. Among other SDMs, the E-R model is selected because it is currently the most popular modeling methodology and well established for real practical databases. In practice, some SDMs can lack linguistic annotations. However, it is not severe because a database designer can make descriptions easily with the help of commercial database modeling tools.

We call each SDM component a domain class. That is, domain classes of an E-R model correspond to entities, attributes, and relationships, which are the main building blocks of the E-R model. Each domain class except relationship types can be viewed equivalently as one physical database structure such as a table or a column. Thus, in this work, domain classes and database structures are used interchangeably. Domain class dictionary (DCD) has a linguistic term for its dictionary entry term, and a set of corresponding domain classes for its content. For each domain surrogate in SDM, DCD entry words are obtained from its nominal variations, and the associated SDM component (a domain class) becomes DCD content. As an example, if an entity 'dCUSTOMER' has a domain surrogate 'customer', the resulting DCD will be <customer: {TB_CUSTOMER}>. In this case, given a question 'Who ordered a refrigerator yesterday?' a linguistic concept 'cPERSON' for 'who' is translated into a table name 'TB_CUSTOMER' by conceptually matching 'cPERSON' with a DCD entry 'customer'. DCD entries are anticipated to cover representative domain terminologies since they are extracted from SDM that is assumed to contain representative linguistic descriptions about a target database domain.

ii) Augmented Transition Network:
The ATN parser differs in two important respects from the finite state grammar. Firstly, the arcs of one finite state network may be labeled with the names of other networks; thus, in the extremely simple grammar of three networks displayed in Figure 2.2 below, transition to state 2 requires the first word of a sentence (S) to be an aux(iliary verb), while transition to state 1 or from state 2 to 3 requires the satisfactory completion of the NP network, i.e. testing for the categories 'pron(oun)', 'det(erminer)', 'n(oun)' and reaching state 7 or state 8. The optional PP network – its optionality indicated by an arc looping back to the same state – requires the testing for'prep (osition)' and again the satisfactory completion of the NP network. As such, this parser would still be no more powerful than a phrase structure grammar. It can in fact be made equivalent to a transformational grammar.

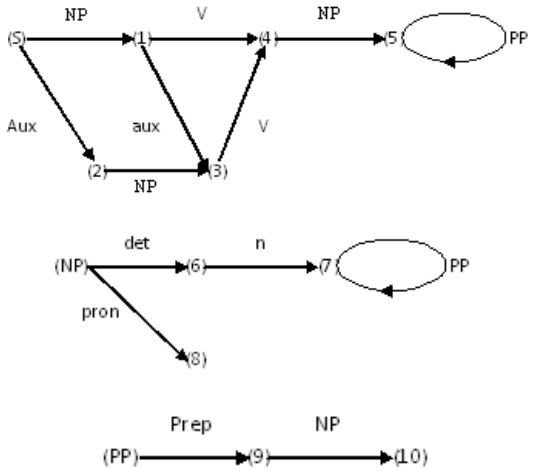

Figure 1. Partial ATN grammars.

Its 'transformational' capability is achieved by adding tests and conditions to the arcs and by specifying 'building instructions' to be executed if the arc is followed. Thus, for example, transition of arc 'aux' to state 2 would specify the building of the first elements of an interrogative (phrase) structure, which could be confirmed or rejected by the conditions or instructions associated with other arcs. Likewise, the transition of an arc recognizing a passive verb form would specify the building of elements of a passive construction to be confirmed or rejected as later information is acquired.

One of the principal attractions of ATN parsers is that they are by no means restricted to syntactic analysis. Indeed in AI systems they are commonly used for deriving semantic representations. Conditions may specify any type of linguistic data: thus, arcs can test for morphological elements (suffixes and verb endings) and for semantic categories ('animate', 'concrete', etc.); and instructions can build morphological analysis and semantic representations. Furthermore, because the arcs can be ordered, an ATN parser can make use of statistical data about the language and its grammatical and lexical structures. Normally ATN parsers operate top-down, with all the disadvantages that entails, principally in wasteful reiterated analysis of lower level constituents. However, it is also possible for ATN parsers to be implemented breadth-first, exploring all possible paths 'in parallel', and thus minimizing backtracking routines.

DOI 10.17148/IJARCCE.2015.41143

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 11, November 2015

iii) Context Free Grammar:
The Context Free Grammar (CFG) has been used to describe a NL for a long time and it is the simplest statistical model to analyze NL. The NL sentences are transformed into a tree structure through CFG and the grammar tree is analyzed according to user's requirements. CFG is a more powerful method of describing language. CFG is used for understanding the relationship of terms such as noun, verb and preposition and their respective phrases leads to a natural recursion because noun phrases may appear inside verb phrases and vice versa. CFG can capture important aspects of these relationships.
Consider a sentence w_{1m} that is a sequence of words w_1 w_2 $w_3 \ldots w_m$ (ignoring punctuations) and each string w_i in the sequence stands for a word in the sentence. The grammar tree of w_{1m} can be generated by a set of pre-defined grammar rules; usually more than one grammar tree may be generated. The importance of CFG has a formalization capability in describing most sentence structures and also CFG is so well formed that efficient sentence parser could be built on top of it.

iv) LEX and YACC
Yet another compilers compiler (YACC) is a parser. The code a grammar and input it to YACC. YACC will read the given grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, output assembly language.
LEX is a lexical analysis tool based on the theory of regular expressions, taking a stream of text and transforming it to a stream of tokens. YACC is a context free grammar based parser, taking a stream of tokens and producing a tree of tokens. Typically a stream would be parsed once with LEX, and then the lexemes generated would be parsed with YACC. These tools are designed to fit together in a natural manner, with similar conventions to each other, and an easy mechanism for their combination. Both LEX and YACC take as input a syntax file describing the language, and produce as output a C file that can be used to parse the language. A compiler is then needed to turn this C file into executable code, and parse the instructions embedded in the language description which are included in C code.

v) Parser:
A parser is one of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. The parser often uses a separate lexical analysis to create tokens from the sequence of input characters. Parsers may be programmed by hand or may be semi-automatically generated (in some programming language) by a tool (such as YACC) from a grammar written in Backus-Naur form.
The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

- Top-down parsing - Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules. LL parsers and recursive-descent parser are examples of top-down parsers.
- Bottom-up parsing - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

Another important distinction is whether the parser generates a leftmost derivation or a rightmost derivation (see context-free grammar). LL parsers will generate a leftmost derivation and LR parsers will generate a rightmost derivation (although usually in reverse).

LR Parser
LR parser is an efficient, bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. This technique is called LR parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse. The LR parser can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written. The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods. It can detect a syntactic error as soon as it is possible to do on a left-to-right scan of the input.
In domain class dictionary the translation ambiguities occur when a linguistic term is associated with two or more domain classes. The ATN parsers operate top-down, with all the disadvantages that entails, principally in wasteful reiterated analysis of lower level constituents. The CFG is new approach to parsing for context-free grammars, which is conceptually very simple. The significance of our approach is supported by recent trends in computer-related fields. In computational linguistics, much attention has been drawn to parsing of context-free grammars owing to the progress of context- free based grammatical frameworks for NLs. The practical NL interface systems are based on context-free (phrase structure) grammars.

III. ANALYSIS AND DESIGN

A) System Architecture
The system architecture of NL database interface developed is given in Figure 3.1, which depicts the layout of the processes included in converting NL query into a syntactical SQL query to be fired on the RDBMS. To process a query, the first step is speech tagging; followed

IJARCCE
ISSN (Online) 2278-1021
ISSN (Print) 2319 5940
International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 11, November 2015

by word tagging. The second step is parsing the tagged sentence by a grammar. The grammar parser analyzes the query sentence according to the tag of each word and generates the grammar tree/s. Finally, the SQL translator processes the grammar tree to obtain the SQL query.

The SQL translator generates query in technical language like SQL. From the input statement using grammar obtain the parse tree. After obtaining the parsed grammar tree, the next step is translating the leaves of the tree to the corresponding SQL. Actually the process is collecting the information from the parsed tree.

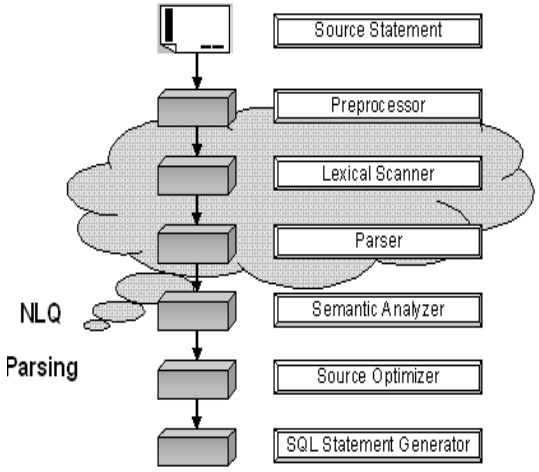

Figure2. Architecture of NLDBI System

B) Design of Context Free Grammar

The CFG has been used to describe a NL for a long time and it is the simplest statistical model to analyze NL. The NL sentences are transformed into a tree structure through CFG and the grammar tree is analyzed according to user's requirements. CFG is a more powerful method of describing language. CFG is used for understanding the relationship of terms such as noun, verb and preposition and their respective phrases leads to a natural recursion because noun phrases may appear inside verb phrases and vice versa. CFG can capture important aspects of these relationships

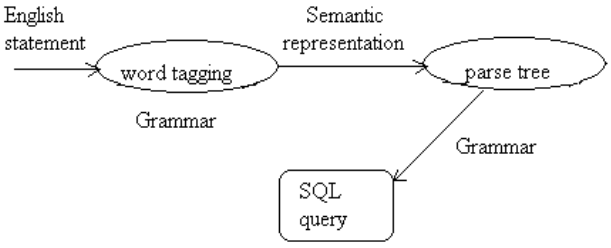

Figure3. Generation of SQL Query from English Statement

Figure3. Depicts the processing of English input statement to generate SQL query. The entire process involves tagging of input statement, apply grammar and semantic representation to generate parse tree, analyze the parse tree using grammar and translating the leaves of the tree to generate corresponding SQL query.

The database tables considered are EMP (empid, empname, salary, edepid, address, post, mobileno), DEPT (deptid, deptname, deptloc, dcapacity) and PROJECT (pid, pname, epid). From the input NL statement, to generate parse tree the grammar written is defined as tuple G = (V, ∑, P, S) in which:

Terminal set: ∑= {a, an, the, id, number, name, salary, income, earning, manager, boss, id, number, location, capacity, employee, worker, person, emp, employees, emps, workers, persons, project, projects, department, dept, dpt, departments, depts, dpts}, are the words corresponding to a leaf in the grammar tree.

Non-terminal set: V= {WhatKeyBank, AAnTheBank, empid, empname, salary, mgrid, deptid, deptname, deptloc, dcapacity, EmpTable, ProjectTable, DeptTable}, which is used to generate terminals, corresponding to non-leaf nodes in the grammar tree.

Designated start symbol S, which is an input NL statement.

P is set of rules: The grammar in the system consists of following rules;

WhatKeyBank → for | of | with | is | where | whose | having | in | on

AAnTheBank → a | an | the

empid → integer | id | number

empname → string | name

salary → integer | salary | income | earning

mgrid → integer | manager | boss | superior

edeptid → integer | id | number

deptid → integer | id | number

deptname → string | name

deptloc → string | location

dcapacity → integer | capacity

EmpTable → employee | worker | person | emp | employees | emps | workers |persons

ProjectTable → project | projects

DeptTable → department | dept | dpt | departments | depts | dpts

The experimental work is to design an interface for generating queries from NL statements/questions. It also consists of designing a parser for the NL statements, which will parse the input statement, generate the technical query and fire it to the end-database. The experimental work will understand the exact meaning the end user wants to go for, generate a what- type sentence and then convert it into a query and give it to the interface. The interface further processes the query and searches for the database. The database gives the result to the system and the result is displayed to the user.

C) Use of LEX and YACC

The UNIX utility LEX parses a file of characters. It uses regular expression matching. Typically it is used to 'tokenize' the contents of the file. In that context, it is often used together with the YACC utility. The UNIX utility YACC parses a stream of token, typically generated by LEX, according to a user-specified grammar.

i) Structure of a LEX file:

A LEX file looks like:

...definitions...

IJARCCE
ISSN (Online) 2278-1021
ISSN (Print) 2319 5940
International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 11, November 2015

%%
...rules...
%%
...code...

Definitions: All code between %{ and %} is copied to the beginning of the resulting C file.
Rules: A number of combinations of pattern and action: if the action is more than a single command it needs to be in braces.
Code: This can be very elaborate, but the main ingredient is the call to yylex, the lexical analyzer. If the code segment is left out, a default main is used which only calls yylex.

ii) Structure of a YACC file
A YACC file looks much like a LEX file:
...definitions...
%%
...rules...
%%
...code...
Definitions: As with LEX, all code between %{ and %} is copied to the beginning of the resulting C file.
Rules: As with LEX, a number of combinations of pattern and action. The patterns are now those of a context-free grammar, rather than of a regular grammar as was the case with LEX.
Code: This can be very elaborate, but the main ingredient is the call to yyparse, the grammatical parse.

D) Pushdown Automata

A pushdown automaton (PDA) is essentially a finite state automaton augmented with an auxiliary tape on which it can read, write, and erase symbols. Its transitions from state to state can depend not only on what state it is in and what it sees on the input tape but also on what it sees on the auxiliary state, and its actions can include not only change of state but also operations on the auxiliary tape. The auxiliary tape works as a pushdown store, "last in, first out", like a stack of plates in some cafeterias. You can't 'see' below the top item on the stack without first removing (erasing) that top item.

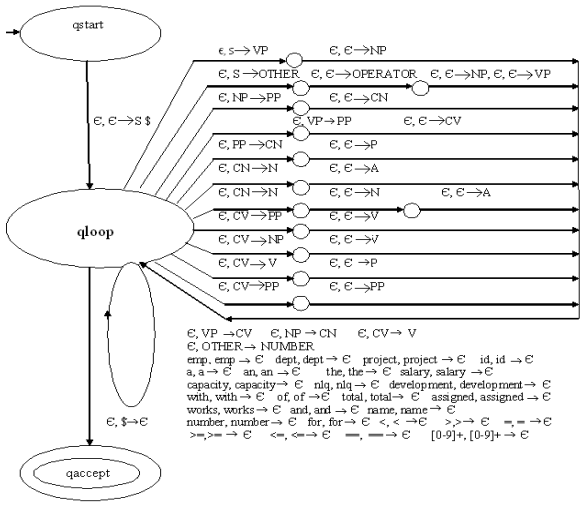

Figure4. Pushdown Automata for given CFG

Figure4. Shows the pushdown automata (PDA) for the above context free grammar. The above PDA accepts the different NL statements, such as "The salary of an employee".

A PDA is a class of machines recognizing the context free languages. Pushdown automata are equivalent in power to context free grammars. This equivalence is useful because it gives us two options for providing that a language is context free. We can give either a context free grammar generating it or a pushdown automata recognizing it. The context free grammar generates NL statement and the above pushdown automata recognize it.

IV. ALGORITHMS AND RELATED THEORY

The algorithm is designed to generate SQL query from NL statement. Microsoft visual C# and VC++ tools are used to implement the NLDBI system. The YACC is used to check whether the NL statement follows the defined CFG or not.

A. Algorithm
The algorithm used to generate SQL query from NL statement is as follow:
The user has input statement to system in NL. First step is to tag the input statement and then each word of statement to be tagged. Second step is the tagged words are checked with grammar and stored it into symbol table. The words from symbol table are extracted and checked with appropriate grammar and if the matching word is found it is used to generate the query. The system first converts input statement into standard what- type question/s and then the question is to convert it into SQL query and fire on the database to display the results to the user.

Algorithm 1 Generation of parse tree/s from NL statement using grammar.

1. Read Input Statement S
2. **for** each word W_i from S **do**
3. **if** ($W_i \in$ Grammar G) **then**
4. Add W_i to Symbol Table ST
5. **end if**
6. **end for**
7. **for** each W_i from ST **do**
8. **Add** W_i to parse tree/s T for *What*- type question/s
9. **end for**
10. Display *What*- type question/s Q
11. Read Input Q
12. **for** each W_i from Q **do**
13. **if** ($W_i \in$ G) **then**
14. **Add** W_i to parse tree for *SQL*- query
15. **end if**
16. **end for**
17. Display *SQL*- query

Figure5. Algorithm Generation of Parse Tree/s for input Statement Using Grammar.

B. Tools Used
The system is built using Microsoft Visual C++ programming language and it is run with the help of software Microsoft Visual C#.NET with Microsoft .NET Framework 2.0. The C#.NET is used to create the UI for NLDBI system and VC++ 2005 is used to write grammar

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 11, November 2015

and generate parse tree/s to convert NL statement to SQL query.

i) Visual C# and Visual C++:
Microsoft Visual C# is Microsoft's implementation of the C# programming language specification, included in the Microsoft Visual Studio suite of products. The term Visual denotes a brand-name relationship with other Microsoft programming languages such as Visual Basic, Visual FoxPro, Visual J# and Visual C++. All of these products are packaged with a graphical integrated development environment (IDE) and support rapid application development of Windows-based applications. **C#** is a multi-paradigm programming language encompassing imperative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. C# is one of the programming languages designed for the Common Language Infrastructure. C# is intended to be a simple, modern, general-purpose, object-oriented programming language.

Microsoft Visual C++ is a commercial IDE product engineered by Microsoft for the C, C++, and C++/CLI programming languages. It has tools for developing and debugging C++ code, especially code written for the Microsoft Windows API, the DirectX API, and the Microsoft .NET Framework. Visual C++ 2005 enables you to write managed applications for the .NET Framework that take advantage of the number of classes in the .NET Framework Class Library, including features such as garbage collection. The Visual C++ 2005 for writing simple yet powerful code for the .NET Framework's Common Language Runtime. Using Visual C++, you can write class libraries, console applications, or Windows Forms applications. Visual C++ 2005 also lets you use C++ to build 32-bit native code console applications that have access to the full Standard C and C++ libraries. You can also mix native and managed code in a single project, giving you the flexibility to use existing libraries as well as .NET Framework classes in the same application.
Visual C++ 2005 includes the Standard Template Library (STL). STL is a general purpose library of algorithms and data structures that is based on a concept known as generic programming. The library includes the container classes-such as vector, queue, list, and map-that are implemented using C++ templates. These work with any data type, including both built-in types as well as any types you define yourself.

ii) YACC**:**
YACC is a parser that used to find the given statement is follows the defined CFG. The working of LEX and YACC as follows:
Figure 6. illustrates the file naming conventions used by lex and yacc. First, we need to specify all pattern matching rules for lex (nlq.l) and grammar rules for yacc (nlq.y). Commands to create our compiler, nlq.exe, are listed below:
Yacc reads the grammar descriptions in nlq.y and generates a syntax analyzer (parser) that includes function

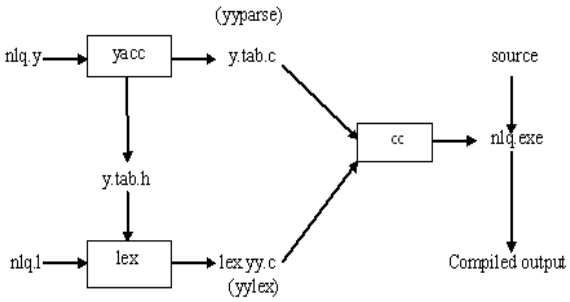

Figure6. Building a compiler with LEX/YACC.

yyparse, in file y.tab.c. Included in file nlq.y are token declarations. The –d option causes yacc to generate definitions for tokens and place them in file y.tab.h. Lex reads the pattern descriptions in nlq.l, includes file y.tab.h, and generates a lexical analyzer that includes function yylex, in file lex.yy.c.
Finally, the lexer and parser are compiled and linked together to form the executable, nlq.exe. From main, we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

V. CONCLUSION

The NL statement is converted into machine understandable form such as SQL. The experimental work understands the exact meaning the end user wants to go for, generates a query and gives it to the interface. The interface further processes the query and fires on the database. The results are extracted from the database and displayed to end user. The NLDBI system is tested for more than 75 different NL input statements and the system works satisfactorily. The advantage of NLDBI system is that it works on a Relational database, also ambiguity among the words is removed.

The NL input statements are compared with the standard YACC tool to check that the statement follows the defined CFG. More than 14 different NL input statements parsed correctly by the YACC.

The Future Scope of NLDBI systems are as follows.

- There is a need to translate NL statement into **what-type** question and finally, the what- type question is to be translated into **SQL- query**.
- Limited Data Dictionary (EMP, DEPT and PROJECT).
- All the input names need to be in double quotes and also the system considers only selection of data.

So far, our NLDBI system considers selection of data and performing queries onto the database and JOINS operation with some constraints. The CFG is enhanced to generate unambiguous parse tree. The next step of research would be to optimize grammar to accommodate more complex queries with emphasis to incorporate update, insert and delete commands in the application as well. Interface which solely use keyboard input of natural language are not likely to be practical; in the long term, the use of spoken input is more likely to be the route to practical success.

DOI 10.17148/IJARCCE.2015.41143

REFERENCES

[1] Bei-Bei Huang, Guigang Zhang, Phillip C-Y Sheu "A Natural Language Database Interface Based On Probabilistic Context Free Grammar" The State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China, Department of EECS, University of California, Irvine 92697, USA,2008.

[2] In-Su Kang, Jae-Hak J. Bae, Jong-Hyeok Lee "Database Semantics Representation for Natural Language Access". Department of Computer Science and Engineering, Electrical and Computer Engineering Division Pohang University of Science and Technology (POSTECH) and Advanced Information Technology Research Center (AITrc), 2002.

[3] Woods, W., Kaplan, R. "Lunar rocks in natural English: Explorations in natural language question answering". Linguistic Structures Processing. In Fundamental Studies in Computer Science, 5:521-569, 1977.

[4] Androutsopoulos, I., Richie, G.D., Thanisch, P. "Natural Language Interface to Database – An Introduction". Journal of Natural Language Engineering, Cambridge University Press. 1(1), 29-81, 1995.

[5] Linguistic Technology. English Wizard – Dictionary Administrator's Guide. Linguistic Technology Corp., Littleton, MA, USA, 1997.

[6] Akama, S. (Ed.) Logic, language and computation, Kulwer Academic publishers, pp. 7-11, 1997.

[7] ELF Software CO. Natural-Language Database Interfaces from ELF Software Co, cited November 1999.

[8] Hendrix, G.G., Sacerdoti, E.D., Sagalowicz, D., Slocum, J. "Developing a natural language interface to complex data", in ACM Transactions on database systems, 3(2), pp. 105-147, 1978.

[9] Joseph, S.W., Aleliunas, R. "A knowledge-based subsustem for a natural language interface to a database that predicts and explains query failures", in IEEE CH, pp. 80-87, 1991.

[10] Mitrovic, A. A knowledge-based teaching system for SQL, University ofCanterbury, 1998. Moore, J.D. "Discourse generation for instructional applications: making computer tutors more like humans", in Proceedings AI-ED, pp.36-42, 1995.

[11] Suh, K.S., Perkins, W.C., "The effects of a system echo in a restricted natural language database interface for novice users", in IEEE System sciences, 4, pp. 594-599, 1994.

[12] Weisenbaum, J., "ELIZA: A Computer program for the study of natural language communication between man and machine", Communications of the ACM, 9(1), 1966.

[13] Whenhua, W., Dilts, D.M. "Integrating diverse CIM data bases: the role of natural language interface", in IEEE Transactions on systems, man, and cybernetics, 22(6), pp. 1331-1347, 1992.

[14] Wintraecken, J.J.V.R. 1990. The NIAM Information Analysis Method: theory and practice. Dordrecht: Kluwer Academic.