B.Sc. in Computer Science and Engineering Thesis

Unfolding of Conjunctive Queries Using Context Free Grammar for Parsing

Submitted by

Kona Rani Showjal
ID- 201114019

Jannathun Naima
ID- 201114030

Fatama Toj Johora
ID- 201114043

Supervised by

Dr. Muhammad Masroor Ali

Professor

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka-1205, Bangladesh

Department of Computer Science and Engineering
Military Institute of Science and Technology

December 2014

CERTIFICATION

This thesis paper titled **“Unfolding of Conjunctive Queries Using Context Free Grammar for Parsing”**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in December 2014.

Group Members:

Kona Rani Showjal

Jannathun Naima

Fatama Toj Johora

Supervisor:

Dr. Muhammad Masroor Ali
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1205, Bangladesh

CANDIDATES’ DECLARATION

This is to certify that the work presented in this thesis paper, titled, “Unfolding of Conjunctive Queries Using Context Free Grammar for Parsing”, is the outcome of the investigation and research carried out by the following students under the supervision of Dr. Muhammad Masroor Ali, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka-1205, Bangladesh.

It is also declared that neither this thesis paper nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Kona Rani Showjal
ID- 201114019

Jannathun Naima
ID- 201114030

Fatama Toj Johora
ID- 201114043

ACKNOWLEDGEMENT

We are thankful to Almighty Allah for his blessings for the successful completion of our thesis. Our heartiest gratitude, profound indebtedness and deep respect go to our supervisor, Dr. Muhammad Masroor Ali, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka-1205, Bangladesh, for his constant supervision, affectionate guidance and great encouragement and motivation. His keen interest on the topic and valuable advices throughout the study was of great help in completing thesis.

We are especially grateful to the Department of Computer Science and Engineering (CSE) of Military Institute of Science and Technology (MIST) for providing their all out support during the thesis work.

Finally, we would like to thank our families and our course mates for their appreciable assistance, patience and suggestions during the course of our thesis.

Dhaka
December 2014

Kona Rani Showjal

Jannathun Naima

Fatama Toj Johora

ABSTRACT

There are two different notations for queries over relational databases throughout the thesis. The first is SQL, which is the language used to query relational data in commercial relational systems. SQL is a very complex language. For our discussion, we typically used only its most basic features such as selecting specific rows from a table, selecting specific columns from a table, combining data from multiple tables using the join operator, taking the union of two tables and computing basic aggregation functions. Secondly, we used the notation of Conjunctive Queries, which is based on mathematical logic.

Lexical analysis is the extraction of individual words or lexemes from an input stream of symbols and passing corresponding tokens back to the parser. If we consider a statement in a programming language, we need to be able to recognize the small syntactic units (tokens) and pass this information to the Yacc generator. We need to also store the various attributes in the symbol or literal tables for later use, *e.g.*, if we have an variable, the tokenizer would generate the token *var* and then associate the name of the variable with it in the symbol table. In this case, the variable name is the lexeme. The tokenization process takes input and then passes this input through a keyword recognizer, an identifier recognizer, a numeric constant recognizer and a string constant recognizer, each being put in to their own output based on disambiguating rules.

Our goal was to unfold Conjunctive Queries. We successfully implemented our production rules and tokenization process. The rest of our task is to implement unfolding Conjunctive Queries with C language and merge with Yacc generator. So, we are hopeful that we will be able to unfold Conjunctive Queries in future.

TABLE OF CONTENT

CERTIFICATION **ii**

CANDIDATES' DECLARATION **iii**

ACKNOWLEDGEMENT **iv**

ABSTRACT **1**

List of Figures **4**

List of Tables **5**

List of Abbreviation **6**

1 Introduction **7**
- 1.1 Review of Our Topic 7
- 1.2 Method Summary 8
- 1.3 Organization of Rest of the Thesis 8

2 SQL and Conjunctive Query **10**
- 2.1 SQL 10
  - 2.1.1 Example 11
  - 2.1.2 Another Example 12
  - 2.1.3 Subqueries 12
  - 2.1.4 Functional Dependencies 13
- 2.2 Conjunctive Query 14
  - 2.2.1 Example 14
  - 2.2.2 Datalog Programs 16

3 Lexical Analysis and Lexer Generator **20**
- 3.1 Lexical Analysis 20
  - 3.1.1 Token 21
  - 3.1.2 Tokenization 22
  - 3.1.3 Scanner 23
  - 3.1.4 Evaluator 23
- 3.2 Lexer Generator 25

4 Syntax Analysis and Syntax Generator **28**
- 4.1 Syntax Analysis 28

4.2 The Role of the Parser 29
4.3 Parsing Representative Grammars 30
4.4 Derivation 30
4.4.1 Example Grammar 31
4.5 Syntax Trees and Ambiguity 32
4.6 Syntax Error Handling 33
4.7 Syntax-Directed Definitions 34

5 Lexical Code **36**
5.1 Explanation of Our Lex Code 36
5.1.1 Example 37

6 Yacc Code **39**
6.1 Explanation of Our Yacc Code 39
6.1.1 Example 40

7 Experiment and Result **42**
7.1 Example 42
7.2 Conditional Query Example 43

8 Discussion and Conclusion **44**
8.1 Discussion 44
8.1.1 Future Expansion 44
8.2 Conclusion 45

References **45**

A Codes **47**
A.1 Lexical Code 47
A.2 Yacc Code 48

LIST OF FIGURES

3.1 Compilation Sequence. 26

4.1 Syntax analyzer. 29
4.2 Parser Sequence. 30
4.3 Alternative syntax tree for the string aabbbcc using Grammar. 33

8.1 Parse Tree. 45

LIST OF TABLES

3.1 Token Example 22
3.2 Pattern Matching Primitives 25

LIST OF ABBREVIATION

BNF	: Backus-Naur-form
CFG	: Context-free-grammar
FSM	: Finite-State Machine
SD	: Syntax-diagram
SQL	: Structure Query Language

CHAPTER 1
INTRODUCTION

1.1 Review of Our Topic

Structure Query Language abbreviated as SQL [1] is a widely-used programming language for working with relational databases. A relational database is a digital database whose organization is based on the relational model of data, as proposed by E.F. Codd [2] in 1970. This model organizes data into one or more tables of rows and columns, with a unique key for each row. Generally, each entity type described in a database has its own table, the rows representing instances of that entity and the columns representing the attribute values describing each instance. Because each row in a table has its own unique key, rows in other tables that are related to it can be linked to it by storing the original row's unique key as an attribute of the secondary row. The various software systems used to maintain relational databases are known as Relational Database Management Systems.

We used two different notations for queries over relational databases throughout the thesis. The first is SQL, which is the language used to query relational data in commercial relational systems. Unfortunately, SQL is not known for its aesthetic aspects and hence not convenient for more formal expositions. Hence, in some of the more formal discussions, we use the notation of Conjunctive Queries, which is based on mathematical logic. SQL is a very complex language. For our discussion, we typically use only its most basic features: selecting specific rows from a table, selecting specific columns from a table, combining data from multiple tables using the join operator, taking the union of two tables and computing basic aggregation functions. A Conjunctive Query is a restricted form of first-order queries. Many first-order queries can be written as Conjunctive Queries. In particular, a large part of queries issued on relational databases can be expressed in this way. Datalog is a truly declarative logic programming language that syntactically is a subset of Prolog. It is often used as a query language for deductive databases. In recent years, Datalog has found new application in data integration. The semantics of Datalog Programs are based on Conjunctive Queries. So, for our purpose we used Datalog Program.

Lex [3] helps write programs whose control flow is directed by instances of regular expressions in the input stream. The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it. Lex can generate analyzers in C. Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program.

A Yacc generator is a software component that takes input data (frequently text) and builds a data structure often some kind of parse tree, abstract syntax tree or other hierarchical structure giving a structural representation of the input, checking for correct syntax in the process. Lexical and syntactical analysis can be simplified to a machine that takes in some program code and then returns syntax errors, parse trees and data structures. We can think of the process of description transformation, where we take some source description, apply a transformation technique and end up with a target description, this is inference mapping between two equivalent languages, where the destination is a machine executable.

1.2 Method Summary

There are two methods that we used for our task. First one is lexical analysis and second one is syntax analysis. From source code, lexical analysis produces tokens, the words in a language which are then parsed to produce a syntax tree. It checks that tokens conform with the rules of a language. Semantic analysis is then performed on the syntax tree to produce an annotated tree. In addition to this, a literal table which contains information on the strings and constants used in the program and a symbol table which stores information on the identifiers occurring in the program (*e.g.*, variable names, constant names, procedure names etc) are produced by the various stages of the process. An error handler also exists to catch any errors generated by any stages of the program (*e.g.*, a syntax error by a poorly formed line). The syntax tree forms an intermediate representation of the code structure and has links to the symbol table.

1.3 Organization of Rest of the Thesis

Chapter 2 discusses about Conjunctive Query and SQL. As, SQL is complex language so we used Conjunctive Query for our purpose. Actually, we used Datalog for Conjunctive Query. These connect with relational database.

Chapter 3 discusses basic concept about lexical analysis and lexical generator. Here, we define identifiers. Chapter 4 discusses basic concept about syntax analysis and syntax generator.

Chapter 5 discusses about our lexical code. Chapter 6 discusses about our syntax code.

Chapter 7 discusses about experiment and results of lexical and syntax code. Chapter 8 is discussion and conclusion of whole thesis. All references and codes are included at the end of thesis.

CHAPTER 2
SQL AND CONJUNCTIVE QUERY

We used Conjunctive Query [4] in our experiment. As it connects with the relational databases, it also connects with the SQL. SQL and Conjunctive Query both are large area of knowledge. So, it is not possible to describe all. Here, we discuss the related topics about our thesis.

2.1 SQL

SQL is a complex language so we didn't use this language for our purpose. But there is a relation between SQL and Conjunctive Query.

SQL is a special-purpose programming language designed for managing data held in a relational database management system, or for stream processing in a relational data stream management system. Originally based upon relational algebra and tuple relational calculus, SQL consists of a data definition language and a data manipulation language. The scope of SQL includes data insert, query, update, delete, schema creation, modification and data access control. Although SQL is often described as, and to a great extent is, a declarative language, it also includes procedural elements. This review has been prepared using the sources [1] [5].

The SQL language has several parts:

Data-definition language: The SQL Data-definition language provides commands for defining relation schemas, deleting relations and modifying relation schemas.

Interactive data-manipulation language: The SQL data-manipulation language includes a query language based on both the relational algebra and the tuple relational calculus. It includes also commands to insert tuples into, delete tuples from and modify tuples in the database.

View definition: The SQL Data-definition language includes commands for defining views.

Transaction control: SQL includes commands for specifying the beginning and ending of transactions.

Embedded SQL and dynamic SQL: Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, Cobol, Pascal and Fortran.

Integrity: The SQL Data-definition language includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.

Authorization: The SQL Data-definition language includes commands for specifying access rights to relations and views.

A relational database consists of a collection of relations, each of which is assigned a unique name. The basic structure of an SQL expression consists of many important clauses such as select, from and where.

- The **select** clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.
- The **from** clause corresponds to the Cartesian-product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.
- The **where** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the from clause.
- The **group by** clause is used to project rows having common values into a smaller set of rows. The group by clause is often used in conjunction with SQL aggregation functions or to eliminate duplicate rows from a result set. The where clause is applied before the group by clause.
- The **having** clause includes a predicate used to filter rows resulting from the group by clause. Because it acts on the results of the group by clause, aggregation functions can be used in the having clause predicate.
- The **order by** clause identifies which columns to use to sort the resulting data and in which direction to sort them (ascending or descending). Without an order by clause, the order of rows returned by an SQL query is undefined.

2.1.1 Example

To give an example, imagine a relational database for storing information about students, their addresses, the courses they visit and their gender. Finding all male students and their addresses who attend a course that is also attended by a female student is expressed by the following SQL query:

```
select l.student, l.address

from    attends a1, gender g1,
        attends a2, gender g2,
        lives l

where   a1.student=g1.student
        and   a2.student=g2.student
        and   l.student=g1.student
        and   a1.course=a2.course
        and   g1.gender='male'
        and   g2.gender='female';
```

2.1.2 Another Example

The following query asks for pairs of recruiters and candidates where the recruiter got a low grade on their performance review. The answer to this query may reveal candidates who we may want to reinterview.

```
SELECT recruiter, candidate
FROM Interview, EmployeePerformance
WHERE recruiter = name AND EmployeePerformance.grade < 2.5
```

2.1.3 Subqueries

Queries can be nested so that the results of one query can be used in another query via a relational operator or aggregation function. A nested query is also known as a subquery. While joins and other table operations provide computationally superior alternatives in many cases, the use of subqueries introduces a hierarchy in execution that can be useful or necessary. In the following example, the aggregation function AVG receives as input the result of a subquery:

For example,

```
 SELECT isbn,
        title,
        price
 FROM   Book
```

```
WHERE price < (SELECT AVG(price) FROM Book)
ORDER BY title;
```

Nested Subqueries

SQL provides a mechanism for nesting subqueries. A subquery is a **select-from-where** expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons and determine set cardinality.

As we need to generate more and more functions in real life, so that we face some constraints such as functional dependency.

2.1.4 Functional Dependencies

Functional dependencies play a key role in differentiating good database designs from bad database designs. A functional dependency is a type of constraint that is a generalization of the notion of key. Functional dependencies are constraints on the set of legal relations. They allow us to express facts about the enterprise that we are modeling with our database. we defined the notion of a superkey as follows.

Let *R* be a relation schema. A subset *K* of *R* is a superkey of *R* if, in any legal relation *r(R)*, for all pairs *t1* and *t2* of tuples in *r* such that $t1 \neq t2$ then $t1[K] \neq t2[K]$. That is, no two tuples in any legal relation *r(R)* may have the same value on attribute set *K*. The notion of functional dependency generalizes the notion of superkey.

Consider a relation schema *R* and let $\alpha \subseteq R$ and $\beta \subseteq R$. The functional dependency $\alpha \rightarrow \beta$ holds on schema *R* if, in any legal relation *r(R)*, for all pairs of tuples *t1* and *t2* in *r* such that $t1[\alpha] = t2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$. Using the functional-dependency notation, we say that *K* is a superkey of *R* if $K \rightarrow R$. That is, *K* is a superkey if, whenever *t1[K] = t2[K]*, it is also the case that *t1[R] = t2[R]* (that is, *t1 = t2*). We shall use functional dependencies in two ways:

1. To test relations to see whether they are legal under a given set of functional dependencies. If a relation r is legal under a set F of functional dependencies, we say that r satisfies F.
2. To specify constraints on the set of legal relations. We shall thus concern ourselves with only those relations that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on schema R that satisfy a set F of functional dependencies, we say that F holds on R.

To remove these constraints, we used Conjunctive Query.

2.2 Conjunctive Query

In database theory, a Conjunctive Query is a restricted form of first order queries. Many first-order queries can be written as Conjunctive Queries. In particular, a large part of queries issued on relational databases can be expressed in this way. Conjunctive Queries also have a number of desirable theoretical properties that larger classes of queries do not share. Conjunctive Queries without distinguished variables are called boolean Conjunctive Queries.

As an example of why the restriction to domain independent first order logic is important, consider $x1.\exists x2.R(x2)$ which is not domain independent; see Codd's theorem [6]. This formula can not be implemented in the select-project join fragment of relational algebra and hence should not be considered a Conjunctive Query.

2.2.1 Example

To give an example, imagine a relational database for storing information about students, their addresses, the courses they visit and their gender. Finding all male students and their addresses who attend a course that is also attended by a female student is expressed by the following Conjunctive Query:

`(student, address) . ∃(student2, course) . attends(student, course) ∧ gender(student, 'male') ∧ attends(student2, course) ∧ gender(student2, 'female') ∧ lives(student, address)`

The above query can be written as an SQL query of the Conjunctive Query fragment as example 2.1.1.

We briefly review the formalism for Conjunctive Queries. A Conjunctive Query has the following form:

`Q(X'):-R1(X1'),....,Rn(Xn'),c1,...,cm`

In the query,

`R1(X1'),....,Rn(Xn')`

are the *subgoals (or conjuncts)* of the query and together form the *body* of the query.

The $R_j's$ are database relations and the `X's` are tuples of variables and constants. Note that the same database relation can occur in multiple subgoals. Unless we explicitly give the query a different name, we refer to it as *Q*. The variables in `X'` are called *distinguished* *variables* or *head variables* and the others are *existential variables*. The predicate *Q* denotes the answer relation of the query. Arity is the number of elements in `X'`. We denote by *Vars(Q)* the set of variables that appear in its head or body.

The $C_j{}'s$ are interpreted *atoms* and are of the form X θ Y where X and Y are either variables or constants and at least one of *X* or *Y* is a variable. The operator θ is an interpreted predicate such as =, $\leq$, $\neq$, or $\geq$. We assume the obvious meaning for the interpreted predicates and unless otherwise stated, we interpret them over a dense domain. The semantics of a Conjunctive Query *Q* over a database instance *D* is as follows. Consider any mapping ψ that maps each of the variables in *Q* to constants in *D*. Denote by ψ (R_j) the result of applying ψ to $R_i(X_i')$, by ψ (C_i) the result of applying ψ to (C_j) and by ψ*(Q)* the result of applying ψ to *Q(X')* , all resulting in ground atoms. If

- each of ψ (R_1),...,ψ (R_n) is in D and
- for each $1 \leq j \leq m, \psi(C_j)$ is satisfied,

then (and only then) ψ(Q) is in the answer to Q over D.

To illustrate the correspondence between SQL queries and Conjunctive Queries, the following Conjunctive Query is the same as the SQL query in 2.1.2.

```
Q1 (Y,X) :- Interview (X,D,Y,H,F),

            EmployeePerformance (E,Y,T,W,Z), W < 2.5
```

Note that the join in the Conjunctive Query is expressed by the fact that the variable Y appears in both subgoals. The predicate on the grade is expressed with an interpreted atom.

Conjunctive Queries must be safe that is, every variable appearing in the head also appears in a non-interpreted atom in the body. Otherwise, the set of possible answers to the query may be infinite (*i.e.*, the variable appearing in the head but not in the body can be bound to any value).

We can also express disjunctive queries in this notation. To express disjunction, we write two (or more) Conjunctive Queries with the same head predicate.

The following query asks for the recruiters who performed the best or the worst:

```
Q1 (E,Y) :- Interview (X,D,Y,H,F),
             EmployeePerformance (E,Y,T,W,Z), W < 2.5
Q1 (E,Y) :- Interview (X,D,Y,H,F),
             EmployeePerformance (E,Y,T,W,Z), W > 3.9
```

For queries with negation, we extend the notion of safety as follows: any variable appearing in the head of the query must also appear in a positive subgoal. This review has been prepared using the source [4].

As discussed by author in [7] we can further show how this affects Description Logic [8].

A serious shortcoming of many Description Logic based knowledge representation systems is the inadequacy of their query languages. In this paper we present a novel technique that can be used to provide an expressive query language for such systems. One of the main advantages of this approach is that, being based on a reduction to knowledge base satisfiability, it can easily be adapted to most existing (and future) Description Logic implementations. We believe that providing Description Logic systems with an expressive query language for interrogating the knowledge base will significantly increase their utility.

2.2.2 Datalog Programs

A Datalog Program is a set of rules, each of which is a Conjunctive Query. Instead of computing a single answer relation, a Datalog Program computes a set of intensional relations (called IDB relations), one of them being designated as the query predicate.

In Datalog, we refer to the database relations as the EDB relations (extensional database). Intuitively, the extensional relations are given as a set of tuples (also referred to as ground facts), while the intensional relations are defined by a set of rules. Consequently, the EDB relations can occur in only the body of the rules, whereas the IDB relations can occur both in the head and in the body.

For example, Consider a database that includes a simple binary relation representing the edges in a graph:

`edge (X,Y)` holds if there is an edge from X to Y in the graph.

The following Datalog query computes the paths in the graph, edge is an EDB relation and path is an IDB relation.

```
r1 path (X,Y) :- edge (X,Y)
r2 path (X,Y) :- edge (X,Z), path (Z,Y)
```

The first rule states that all single edges form paths. The second rule computes paths that are composed from shorter ones. The query predicate in this example is path. Note that replacing r2 with the following rule would produce the same result.

```
r3 path (X,Y) :- path (X,Z), path (Z,Y)
```

The semantics of Datalog Programs are based on Conjunctive Queries. We begin with empty extensions for the IDB predicates. We choose a rule in the program and apply it to the current extension of the EDB and IDB relations. We add the tuples computed for the head of the rule to its extension. We continue applying the rules of the program until no new tuples are computed for the IDB relations.

The answer to the query is the extension of the query predicate. When the rules do not contain negated subgoals, this process is guaranteed to terminate with a unique answer, independent of the order in which we applied the rules.

In data integration we are interested in Datalog Programs mostly because they are sometimes needed in order to compute all the answers to a query from a set of data sources.

We know that Datalog is a subset of Prolog. All SQL queries can not be expressed in Datalog. In particular, there is no support in datalog for grouping and aggregation and for outer joins. SQL does support limited kinds of recursion but not arbitrary recursion.

Datalog Programs, *i.e.*, Prolog programs without function symbols are considered. It is assumed that a variable appearing in the head of a rule must also appear in the body of the rule. The input of a program is a set of ground atoms (which are given in addition to the program's rules) and therefore, can be viewed as an assignment of relations to some of the program's predicates.

Two programs are equivalent if they produce the same result for all possible assignments of relations to the extensional predicates (*i.e.*, the predicates that do not appear as heads of rules). Two programs are uniformly equivalent if they produce the same result for all possible assignments of initial relations to all the predicates (*i.e.*, both extensional and intentional). The equivalence problem for Datalog Programs is known to be undecidable. It is shown that uniform equivalence is decidable and an algorithm is given for minimizing a Datalog Program under uniform equivalence. A technique for removing parts of a program that are redundant under equivalence (but not under uniform equivalence) is developed. A procedure for testing uniform equivalence is also developed for the case in which the database satisfies some constraints.

Example Datalog Program:

```
parent(bill,mary).
parent(mary,john).
```

These two lines define two facts. They can be intuitively understood that the parent of mary is bill and the parent of john is mary.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

These two lines describe the rules that define the ancestor relationship. A rule consists of two main parts separated by the :- symbol. The part to the left of this symbol is the head and the part to the right is the body of the rule. A rule is read (and can be intuitively understood) as (head) if it is known that (body). Uppercase letters stand for variables. Hence in the example the first rule can be read as X is the ancestor of Y if it is known that X is the parent of Y. And the second rule as X is the ancestor of Y if it is known that X is the parent of some Z and Z is the ancestor of Y. The ordering of the clauses is irrelevant in Datalog in contrast to Prolog which depends on the ordering of clauses for computing the result of the query call.

Datalog distinguishes between extensional and intensional predicate symbols. Extensional predicate symbols are defined by facts and intensional predicate symbols are defined by rules. In the example above *ancestor* is an intensional predicate symbol and *parent* is extensional. Predicates may also be defined by facts and rules and therefore neither be purely extensional nor intensional but any Datalog Program can be rewritten into an equivalent program without such predicate symbols with duplicate roles.

```
?- ancestor(bill,X).
```

Features

Unlike in Prolog, statements of a Datalog Program can be stated in any order. Furthermore, Datalog queries on finite sets are guaranteed to terminate, so Datalog does not have Prolog's cut operator. This makes Datalog a truly declarative language.

In contrast to Prolog, Datalog

- disallows complex terms as arguments of predicates, *e.g.*, `p(1, 2)` is admissible but not `p(f(1), 2)`,
- imposes certain stratification restrictions on the use of negation and recursion,
- requires that every variable that appears in the head of a clause also appears in a nonarithmetic positive (*i.e.* not negated) literal in the body of the clause,
- requires that every variable appearing in a negative literal in the body of a clause also appears in some positive literal in the body of the clause

Query evaluation with Datalog is based on first order logic and is thus sound and complete. However, Datalog is not Turing complete and is thus used as a domain-specific language that can take advantage of efficient algorithms developed for query resolution. Indeed, various methods have been proposed to efficiently perform queries, *e.g.*, the Magic Sets algorithm, tabled logic programming or SLG resolution.

Some widely used database systems include ideas and algorithms developed for Datalog. For example, the SQL:1999 standard includes recursive queries and the Magic Sets algorithm (initially developed for the faster evaluation of Datalog queries) is implemented in IBM's DB2. Moreover, Datalog engines are behind specialized database systems such as Intellidimension's database for the semantic web.

Several extensions have been made to Datalog, *e.g.*, to support aggregate functions, to allow object-oriented programming or to allow disjunctions as heads of clauses. These extensions have significant impacts on the definition of Datalog's semantics and on the implementation of a corresponding Datalog interpreter.

CHAPTER 3
LEXICAL ANALYSIS AND LEXER GENERATOR

Lex [9] is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

3.1 Lexical Analysis

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens, *i.e.* meaningful character strings. A program or function that performs lexical analysis is called a lexical analyzer, lexer, tokenizer or scanner, though 'scanner' is also used for the first stage of a lexer. A lexical analyzer is generally combined with a parser which together analyze the syntax of programming languages such as in compilers.

A lexical analyzer is itself a kind of parser-syntax of some programming language is divided into two pieces: the lexical syntax (token structure), which is processed by the lexer and the phrase syntax, which is processed by the parser.

The lexical syntax is usually a regular language, whose alphabet consists of the individual characters of the source code text. The phrase syntax is usually a context-free language, whose alphabet consists of the tokens produced by the lexer. While this is a common separation, alternatively, a lexer can be combined with the parser in scannerless parsing. This review has been prepared using the source [3]. Lexical analyzer is most often used for compilers . A lexer itself can be divided into two stages: the scanner, which segments the input sequence into groups and categorizes these into token classes and the evaluator, which converts the raw input characters into a processed value. Lexical analyzers are generally quite simple, with most of the complexity deferred to the parser or semantic analysis phases and can often be generated by a lexer generator . However, lexers can sometimes include some complexity, such as phrase structure processing to make input easier and simplify the parser and may be written partially or completely by hand, either to support additional features or for performance.

3.1.1 Token

A token is a string of one or more characters that is significant as a group. The process of forming tokens from an input stream of characters is called tokenization. The characters that form a token are called a lexeme. When a token represents more than one possible lexeme, the lexer saves the string representation of the token, so that it can be used in semantic analysis. The parser typically retrieves this information from the lexer and stores it in the abstract syntax tree. This is necessary in order to avoid information loss in the case of numbers and identifiers.

Tokens are identified based on the specific rules of the lexer. Some methods used to identify tokens include: regular expressions, specific sequences of characters known as a flag, specific separating characters called delimiters and explicit definition by a dictionary. Special characters, including punctuation characters, are commonly used by lexers to identify tokens because of their natural use in written and programming languages.

Tokens are often categorized by character content or by context within the data stream. Categories are defined by the rules of the lexer. Categories often involve grammar elements of the language used in the data stream. Programming languages often categorize tokens as identifiers, operators, grouping symbols or by data type. Written languages commonly categorize tokens as nouns, verbs, adjectives or punctuation. Categories are used for post-processing of the tokens either by the parser or by other functions in the program.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parentheses as tokens, but does nothing to ensure that each '(' is matched with a ')'.

Consider this expression in the C programming language:

```
sum = 3 + 2;
```

Tokenized are represented by the following Table 3.1

When a lexer feeds tokens to the parser, the representation used is typically an enumerated list of number representations. For example 'Identifier' is represented with 0, 'Assignment operator' with 1, 'Addition operator' with 2 etc.

Tokens are frequently defined by regular expressions, which are understood by a lexical

Table 3.1: Token Example

Lexeme	*Token*
sum	`'Identifier'`
$=$	`'Assignment operator'`
3	`'Integer literal'`
$+$	`'Addition operator'`
2	`'Integer literal'`
;	`'End of statement'`

analyzer generator such as Lex. The lexical analyzer (either generated automatically by a tool like Lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream and categorizes them into tokens. This is called 'tokenizing'. If the lexer finds an invalid token, it will report an error.

3.1.2 Tokenization

Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.

For example,

```
The quick brown fox jumps over the lazy dog.
```

The tokens could be represented in XML,

```
<sentence>
  <word>The</word>
  <word>quick</word>
  <word>brown</word>
  <word>fox</word>
  <word>jumps</word>
  <word>over</word>
  <word>the</word>
  <word>lazy</word>
  <word>dog</word>
</sentence>
```

3.1.3 Scanner

The first stage, the scanner, is usually based on a finite-state machine (FSM). It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are known as lexemes). For instance, an integer token may contain any sequence of numerical digit characters. In many cases, the first non-whitespace character can be used to deduce the kind of token that follows and subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token (this is known as the maximal munch rule, or longest match rule). In some languages, the lexeme creation rules are more complicated and may involve backtracking over previously read characters. For example, in C, a single 'L' character is not enough to distinguish between an identifier that begins with 'L' and a wide-character string literal.

3.1.4 Evaluator

A lexeme, however, is only a string of characters known to be of a certain kind (*e.g.*, a string literal, a sequence of letters). In order to construct a token, the lexical analyzer needs a second stage, the evaluator, which goes over the characters of the lexeme to produce a value. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser. Some tokens such as parentheses do not really have values and so the evaluator function for these can return nothing, only the type is needed. Similarly, sometimes evaluators can suppress a lexeme entirely, concealing it from the parser, which is useful for whitespace and comments. The evaluators for identifiers are usually simple. The evaluators for integer literals may pass the string on or may perform evaluation themselves, which can be involved for different bases or floating point numbers. For a simple quoted string literal, the evaluator only needs to remove the quotes, but the evaluator for an escaped string literal itself incorporates a lexer, which unescapes the escape sequences. For example, in the source code of a computer program, the string

```
net_worth_future = (assets - liabilities);
```

might be converted into the following lexical token stream; note that whitespace is suppressed and special characters have no value:

```
NAME net_worth_future
EQUALS
OPEN_PARENTHESIS
NAME assets
MINUS
NAME liabilities
```

```
CLOSE_PARENTHESIS
SEMICOLON
```

Though it is possible and sometimes necessary, due to licensing restrictions of existing parsers or if the list of tokens is small, to write a lexer by hand, lexers are often generated by automated tools. These tools generally accept regular expressions that describe the tokens allowed in the input stream. Each regular expression is associated with a production rule in the lexical grammar of the programming language that evaluates the lexemes matching the regular expression. These tools may generate source code that can be compiled and executed or construct a state table for a finite-state machine. Regular expressions compactly represent patterns that the characters in lexemes might follow. For example, for an English-based language, a NAME token might be any English alphabetical character or an underscore, followed by any number of instances of ASCII alphanumeric characters and/or underscores. The Lex programming tool and its compiler is designed to generate code for fast lexical analyzers based on a formal description of the lexical syntax. It is not generally considered sufficient for applications with a complicated set of lexical rules and severe performance requirements; for instance, the GNU Compiler Collection uses hand-written lexers.

Lex turns the users expressions and actions into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream and perform the specified actions for each expression as it is detected.

$$\texttt{Source} \rightarrow \texttt{Lex} \rightarrow \texttt{yylex}$$

$$\texttt{input} \rightarrow \texttt{yylex} \rightarrow \texttt{output}$$

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase, it is particularly easy to interface Lex and Yacc. Lex programs recognize only regular expressions. Yacc writes parsers that accept a large class of context free grammars but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream and the parser generator assigns structure to the resulting pieces.

For example, Lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a ‘(’ we push it on the stack. When a ‘)’ is encountered we match it with the top of the stack and pop the stack. However, Lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures. Yacc augments an FSA with a stack and can

Table 3.2: Pattern Matching Primitives

Metacharacter	*Matches*
.	`any character except newline`
*	`zero or more copies of the preceding expression`
+	`one or more copies of the preceding expression`
?	`zero or one copy of the preceding expression`
∧	`beginning of line`
[]	`character class`
`(ab)+`	`one or more copies of ab (grouping)`
`‘(a+b)’`	`literal ‘a+b’ (C escapes still work)`

process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching (see Table 3.2). Yacc is appropriate for more challenging tasks.

3.2 Lexer Generator

Lexers are often generated by a lexer generator, analogous to parser generators and such tools often come together. The most established is Lex, paired with the Yacc parser generator and the free equivalents flex/bison. These generators are a form of domain-specific language, taking in a lexical specification generally regular expressions with some markup and outputting a lexer.

These tools yield very fast development, which is particularly important in early development, both to get a working lexer and because the language specification may be changing frequently. Further, they often provide advanced features, such as pre condition and post conditions which are hard to program by hand. However, automatically generated lexer may lack flexibility and thus may require some manual modification or a completely manually written lexer.

Lexer performance is a concern and optimization of the lexer is worthwhile, particularly in stable languages where the lexer is run very frequently. Lex generated lexers are reasonably fast, but improvements of two to three times are possible using more tuned generators. Hand-written lexers are sometimes used, but modern lexer generators produce faster lexers than most hand-coded ones. The Lex family of generators uses a table-driven approach which is much less efficient than the directly coded approach.

There are many lexer generators. Such as,

ANTLR - Can generate lexical analyzers and parsers.

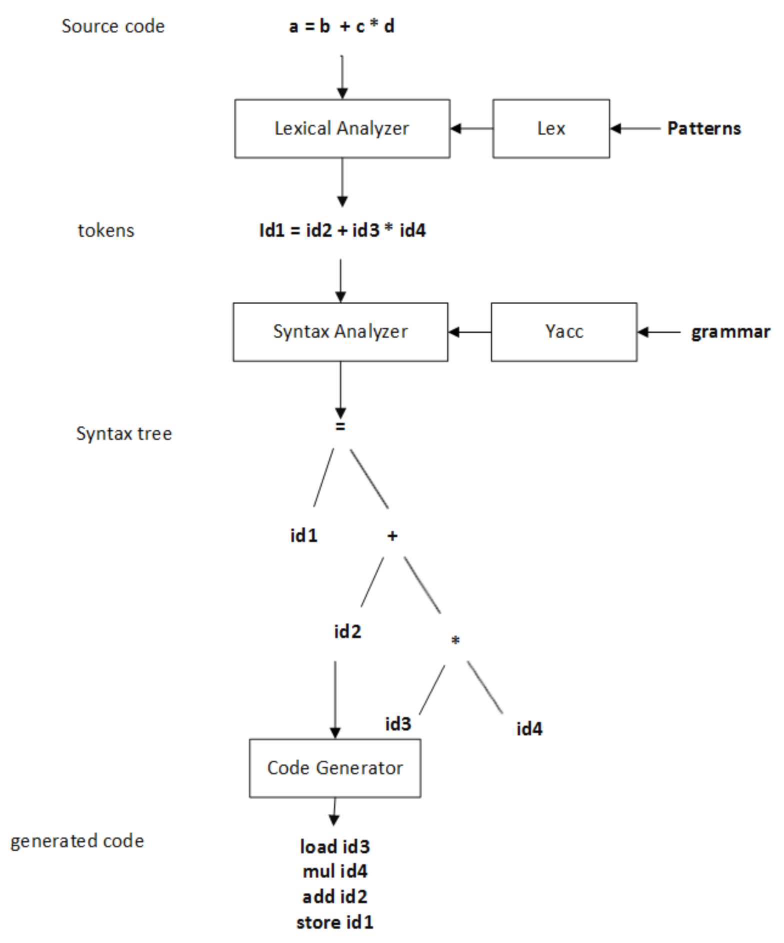

Figure 3.1: Compilation Sequence.

DFASTAR - Generates DFA matrix table-driven lexers in C++.

Flex - Alternative variant of the classic ‘Lex’ (C/C++).

JFlex - A rewrite of JLex.

Ragel - A state machine and lexer generator with output in C, C++, Objective-C, D, Java, Go and Ruby.

Lex generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton and therefore the size of the program generated by Lex.

In the program written by Lex, the users fragments are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine. Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg* and the input stream is *abcdefh* , Lex will recognize *aabb* and leave the input pointer just before *cd*. Such backup is more costly than the processing of simpler languages.

The patterns in the Figure 3.1 is a file we create with a text editor. Lex will read your patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns and converts the strings to tokens. Tokens are numerical representations of strings and simplify processing. When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

CHAPTER 4
SYNTAX ANALYSIS AND SYNTAX GENERATOR

Syntax analysis [10] is alternatively known as parsing. It is roughly the equivalent of checking that some ordinary text written in a natural language is grammatically correct. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc. Note that this sequence need not be meaningful; as far as syntax goes, a phrase such as ‘true + 3’ is valid but it doesn’t make any sense in most programming languages. The parser takes the tokens produced during the lexical analysis stage and attempts to build some kind of in-memory structure to represent that input. This review has been prepared using the sources [11] [12].

4.1 Syntax Analysis

A parser is a software component that takes input data (frequently text) and builds a data structure often some kind of parse tree, abstract syntax tree or other hierarchical structure giving a structural representation of the input, checking for correct syntax in the process. The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyzer, which creates tokens from the sequence of input characters; alternatively, these can be combined in scannerless parsing. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator. Parsing is complementary to templating, which produces formatted output. These may be applied to different domains but often appear together, such as the scanf/printf pair or the input (front end parsing) and output (back end code generation) stages of a compiler.

We take an example of simple variable declaration in C language Example:-

```
int a, b, c;
float x, y;
```

In syntax analysis, this example will be Figure 4.1. Different Formalisms are

- Syntax diagram (SD)
- Backus−Naur form (BNF)

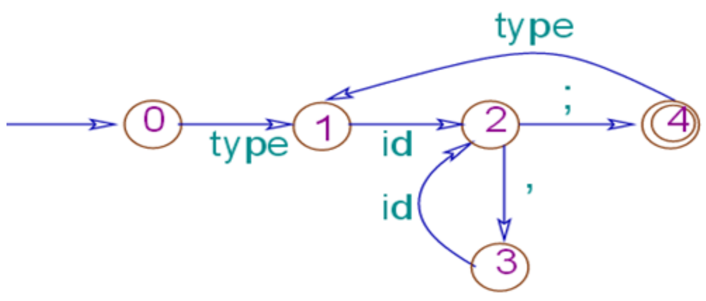

Figure 4.1: Syntax analyzer.

- Context−free grammar (CFG)

Different styles of specification have different purpose. SD is good for human understanding and visualization. The BNF is very compact. It is used for theoretical analysis and also in automatic parser generating softwares. But for most of our discussion we shall consider structural specification in the form of a context-free grammar (CFG).

4.2 The Role of the Parser

In our compiler model Figure 4.2, the parser obtains a string of tokens from the lexical analyzer. It then verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program.

Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly. Since checking and translation actions can be interspersed with parsing. Thus, the parser and the rest of the front end could well be implemented by a single module. There are three general types of parsers for grammars (universal, top-down and bottom-up). Universal parsing methods such as the Cocke-Younger-Kasami algorithm [13] and Earleys algorithm [14] can parse any grammar. These general methods are, however, too inefficient to use in production compilers. The methods commonly used in compilers can be classified as being either top-down or bottom-up. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves). Bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time. The most efficient top-down and bottom-up methods work only for subclasses of grammars. But several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars.

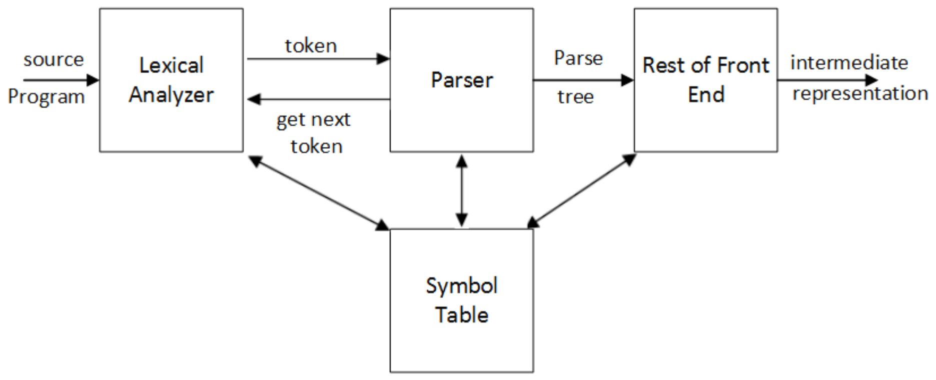

Figure 4.2: Parser Sequence.

The predictive-parsing approach works for LL grammars. Parsers for the larger class of LR grammars are usually constructed using automated tools.

4.3 Parsing Representative Grammars

Some of the grammars that will be examined are presented here for ease of reference. It constructs that begin with keywords like *while* or *int* are relatively easy to parse. The keyword guides the choice of the grammar production that must be applied to match the input. We therefore concentrate on expressions which present more of challenge, because of the associativity and precedence of operators. Associativity and precedence are captured in the following grammar. E represents expressions consisting of terms separated by + signs. F represents factors that can be either parenthesized expressions or identifiers.

1. $E \to E + T|T$
2. $T \to T * F|F$
3. $F \to (E)|id$

The above grammar belongs to the class of LR grammars that are suitable for bottom-up parsing. This grammar can be adapted to handle additional operators and additional levels of precedence. However, it cannot be used for top-down parsing because it is left recursive.

4.4 Derivation

The basic idea of derivation is to consider productions as rewrite rules. Whenever we have a nonterminal, we can replace this by the right-hand side of any production in which the nonterminal appears on the left-hand side. We can do this anywhere in a sequence of symbols (terminals and nonterminals) and repeat doing so until we have only terminals left. The resulting sequence of terminals is a string in the language defined by the grammar. Formally, we define the derivation relation by the three rules:

1. $\alpha N \beta \Rightarrow \alpha\gamma\beta$ if there is a production $N \rightarrow \gamma$
2. $\alpha \Rightarrow \alpha$
3. $\alpha \rightarrow \gamma$ if there is a β such that $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$

where α, β and γ are (possibly empty) sequences of grammar symbols (terminals and nonterminals). The first rule states that using a production as a rewrite rule (anywhere in a sequence of grammar symbols) is a derivation step. The second states that the derivation relation is reflexive, *i.e.*, that a sequence derives itself. The third rule describes transitivity, *i.e.*, that a sequence of derivations is in itself a derivation.

4.4.1 Example Grammar

$T \rightarrow R$

$T \rightarrow aTc$

$R \rightarrow$

$R \rightarrow RbR$

Derivation of the string aabbbcc using Grammar:

T

$\Rightarrow a\mathbf{T}C$

$\Rightarrow aa\mathbf{T}cc$

$\Rightarrow aa\mathbf{R}cc$

$\Rightarrow aaRb\mathbf{R}cc$

$\Rightarrow aa\mathbf{R}bcc$

$\Rightarrow aaRb\mathbf{R}bcc$

$\Rightarrow aaRb\mathbf{R}bRbcc$

$\Rightarrow aa\mathbf{R}bbRbcc$

$\Rightarrow aabb\mathbf{R}bcc$

$\Rightarrow aabbbcc$

4.5 Syntax Trees and Ambiguity

We can draw a derivation as a tree. The root of the tree is the start symbol of the grammar, and whenever we rewrite a nonterminal we add as its children the symbols on the right-hand side of the production that was used. The leaves of the tree are terminals which, when read from left to right, form the derived string. If a nonterminal is rewritten using an empty production, an $\in$ is shown as its child. This is also a leaf node, but is ignored when reading the string from the leaves of the tree. When we write such a syntax tree, the order of derivation is irrelevant: We get the same tree for left derivation, right derivation or any other derivation order. Only the choice of production for rewriting each nonterminal matters.

As an example, the derivations in 4.4.1 yield the same syntax tree, which is shown in new Figure 4.3. The syntax tree adds structure to the string that it derives. It is this structure that we exploit in the later phases of the compiler. For compilation, we do the derivation backwards: We start with a string and want to produce a syntax tree. This process is called syntax analysis or parsing. Even though the order of derivation does not matter when constructing a syntax tree, the choice of production for that nonterminal does. Obviously, different choices can lead to different strings being derived, but it may also happen that several different syntax trees can be built for the same string. When a grammar permits several different syntax trees for some strings we call the grammar ambiguous. If our only use of grammar is to describe sets of strings, ambiguity is not a problem. However, when we want to use the grammar to impose structure on strings, the structure had better be the same every time. Hence, it is a desirable feature for a grammar to be unambiguous. In most (but not all) cases, an ambiguous grammar can be rewritten to an unambiguous grammar that generates the same set of strings, or external rules can be applied to decide which of the many possible syntax trees is the ‘right one’.

Unambiguous version of Grammar 4.4.1

$T \rightarrow R$

$T \rightarrow aTc$

$R \rightarrow$

$R \rightarrow bR$

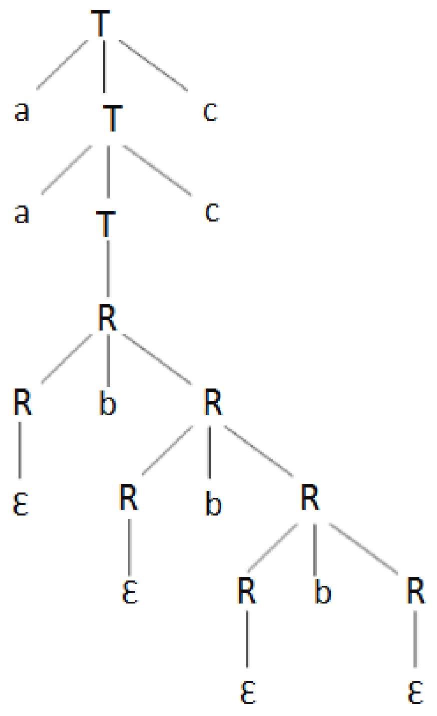

Figure 4.3: Alternative syntax tree for the string aabbbcc using Grammar.

4.6 Syntax Error Handling

If a compiler had to process only correct programs, its design and implementation would be simplified greatly. However, a compiler is expected to assist the programmer in locating and tracking down errors that inevitably creep into programs, despite the programmers best efforts. Strikingly, few languages have been designed with error handling in mind, even though errors are so common place. Our civilization would be radically different if spoken languages had the same requirements for syntactic accuracy as computer languages. Most programming language specifications do not describe how a compiler should respond to errors. Error handling is left to the compiler designer. Planning the error handling right from the start can both simplify the structure of a compiler and improve its handling of errors. Common programming errors can occur at many different levels.

Lexical errors: include misspellings of identifiers, keywords, or operators - *e.g.*, the use of an identifier elipsesize instead of ellipsesize - and missing quotes around text intended as a string.

Syntactic errors: include misplaced semicolons or extra or missing braces, that is, ‘f’ or ‘g’. As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error. However, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code.

Semantic errors: include type mismatches between operators and operands. An example is a return statement in a Java method with result type void.

Logical errors: can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmers intent.

The precision of parsing methods allows syntactic errors to be detected very efficiently. Several parsing methods, such as the LL and LR methods, detect an error as soon as possible. That is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language. More precisely, they have the viable-prefix property, meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language. Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when parsing cannot continue. A few semantic errors, such as type mismatches, can also be detected efficiently. However, accurate detection of semantic and logical errors at compile time is in general a difficult task. The error handler in a parser has goals that are simple to state but challenging to realize:

1. Report the presence of errors clearly and accurately.
2. Recover from each error quickly enough to detect subsequent errors.
3. Add minimal overhead to the processing of correct programs.

Fortunately, common errors are simple ones. A relatively straightforward error-handling mechanism often suffices. How should an error handler report the presence of an error? At the very least, it must report the place in the source program where an error is detected. There is a good chance that the actual error occurred within the previous few tokens. A common strategy is to print the offending line with a pointer to the position at which an error is detected.

4.7 Syntax-Directed Definitions

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. For example: If X is a symbol and a is one of its attributes, then we write X:a to denote the value of a at a particular parse-tree node labeled X.

If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. Attributes may be of any kind: numbers, types, table references, or strings, for instance. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

CHAPTER 5
LEXICAL CODE

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed. We have done our experiment with lexical analysis. Lex generator generates tokens. Yacc generator uses these tokens. Here is a brief description of our Lex code for unfolding queries.

5.1 Explanation of Our Lex Code

A Lex program has three parts, these are

```
declarations
%%
translation rules
%%
auxiliary functions
```

In a Lex code, **declaration part** is to declare identifiers for tokenization. Before doing tokenization this part is needed. Our declaration part is given here,

```
WS       -> [\t]+
DIGIT    -> [0-9]
CAPITAL -> [A-Z]
SMALL    -> [a-z]
Q        -> Q{DIGIT}+
```

Where, `WS` is an identifier for white space, `DIGIT` is for numbers from 0-10, `CAPITAL` is an identifier which indicates capital letter from A-Z, `SMALL` is an identifier which indicates small letter from a-z, `Q` is an identifier which is followed by capital `Q` and infinite number of digits.

Translation rules are to return tokens to Yacc generator.

```
{Q}                {return Q;}
{SMALL}+           {return SMALL;}
{CAPITAL}          {return CAPITAL;}
"("                {return LPAREN;}
")"                {return RPAREN;}
":-"               {return COLONDASH;}
","                {return COMMA;}
"\n"               {return NEWLINE;}
{WS}               {}
```

Here, we are sending all these as tokens for parsing. These tokens are named as `Q`, `SMALL`, `CAPITAL`, `LPAREN`, `RPAREN`, `COLONDASH`, `COMMA`, `NEWLINE`.

Auxiliary functions are built functions in Lex code.

```
main()
{
        freopen("input.txt","r",stdin);
        yylex();
}
```

This part is written in main function. An input file, which we call input.txt, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms input.txt to a C program, in a file that is always named lex.yy.c .

input.txt → **LEX Compiler** → lex.yy.c

lex.yy.c → **C compiler** → Output.out

input stream → **Output.out** → sequence of tokens

Our experimental Lex code is given in Code A.1.

5.1.1 Example

Consider the following example, where `Q3` is defined in terms of `Q1` and `Q2`. The relation Flight stores pairs of cities between which there is a direct flight and the relation Hub stores the set of hub cities of the airline. The query `Q1` asks for pairs of cities between which there is a flight that goes through a hub. The query `Q2` asks for pairs of cities that are on the same outgoing path from a hub.

```
Q1(X,Y) :- Flight (X,Z), Hub (Z), Flight (Z,Y)
Q2(X,Y) :- Hub (Z), Flight (Z,X), Flight (X,Y)
Q3(X,Z) :- Q1 (X,Y), Q2 (Y,Z)
```

First, the predicate are `Q3 (X,Z), Q1 (X,Y), Q2 (Y,Z)` and the subgoals are `Flight(X, Z), Hub (Z), Flight(Z,Y), Flight(Z,X), Flight(X,Y)`. In our Lex code, we define identifiers as `Q3, Q1, Q2` and small and capital letters. Then we define tokens as `Q1, Q2, Q3, Flight, Hub, ( , ) , :- .`

We described this example and result in Chapter 7.

CHAPTER 6
YACC CODE

Yacc is available as a command on the UNIX system and has been used to help implement many production compilers. Yacc generator is used to facilitate the construction of the front end of a compiler. It provides a general tool for describing the input to a computer program. Yacc user specifies the structures of it's input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

6.1 Explanation of Our Yacc Code

A **Yacc** source program has three parts :

```
declarations
%%
translation rules
%%
supporting C routines
```

There are two sections in the declarations part of a Yacc program. Both are optional. In the first section, we put ordinary C declarations, delimited by

`%{ and %}.`

Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections. This section contains the include−statement

```
#include<stdio.h>
#include<stdlib.h>
```

that causes the C preprocessor to include the standard header file of input and output. Also in the declarations part are declarations of grammar tokens. The statement

```
%token COLONDASH COMMA LPAREN RPAREN SMALL CAPITAL Q NEWLINE
```

declares `COLONDASH`, `COMMA`, `LPAREN`, `RPAREN`, `SMALL`, `CAPITAL`, `Q`, `NEWLINE` to be tokens, we declared in Chapter 5 in lexical analysis phase. Tokens declared in this section can then be used in the second and third parts of the Yacc specification. If Lex is used to create the lexical analyzer that passes token to the Yacc parser, then these token declarations are also made available to the analyzer generated by Lex.

A set of productions that we have been writing:

```
<head> -> <body>1|<body>2|...|<body>n
```

would be written in Yacc as

```
<head> :<body>1 {<semantic action>1}
        |<body>2 {<semantic action>2}
        |<body>n {<semantic action>n}
        ;
```

In our experimental code, we declared production rules as

```
          line: NEWLINE
              |predicate NEWLINE

          sub: SMALL LPAREN CAPITAL COMMA CAPITAL RPAREN
```

In supporting C routines,

```
main()
 {
        yyparse();
        exit(0);
 }
```

Our experimental Yacc code is given in Code A.2.

6.1.1 Example

Consider the following example, where Q3 is defined in terms of Q1 and Q2. The relation Flight stores pairs of cities between which there is a direct flight and the relation Hub stores the set of hub cities of the airline. The query Q1 asks for pairs of cities between which there is a flight that goes through a hub. The query Q2 asks for pairs of cities that are on the same outgoing path from a hub.

```
Q1 (X,Y) :- Flight (X,Z), Hub (Z), Flight (Z,Y)
Q2 (X,Y) :- Hub (Z), Flight (Z,X), Flight (X,Y)
```

```
Q3 (X,Z) :- Q1 (X,Y), Q2 (Y,Z)
```

In Section 5.1.1, tokens are declared. The production rules are

```
sub: SMALL LPAREN CAPITAL COMMA CAPITAL RPAREN
```

which defines a subgoal, such as

```
Flight (X, Z).
```

Again,

```
predicate: Q LPAREN CAPITAL COMMA CAPITAL RPAREN COLONDASH Q
                LPAREN CAPITAL COMMA CAPITAL RPAREN
```

This one defines that,

```
Q3(X,Z) :- Q1(X,Y), Q2(Y,Z)
```

Applying other production rules, on Example 6.1.1

```
predicate:  Q LPAREN CAPITAL COMMA CAPITAL RPAREN COLONDASH sub
```

This production rule defines that a predicate can be subgoal. Such as

```
Q1(X,Y) :- Flight(X,Y)
```

In addition to,

```
predicate :  Q LPAREN CAPITAL COMMA CAPITAL RPAREN COLONDASH sub
COMMA Q LPAREN CAPITAL COMMA CAPITAL RPAREN
```

This production rule defines that a predicate can be subgoal. Such as

```
Q2(X,Y) :- Q1(X,Z), Flight(X,Y)
```

We described this example and its result in Chapter 7.

CHAPTER 7
EXPERIMENT AND RESULT

We know, a Conjunctive Query has the following form:

```
Q(X′):-R1(X1′),....,Rn(Xn′),c1,...,cm
```

Our main goal of this experiment was to unfold Conjunctive Queries (see in Section 2.2). For this experiment, we need to implement a parse tree. To generate a parse tree, we need grammar rules and tokens. We generated our grammar rules in Yacc generator and we sent tokens from lexical analyzer. But our parse tree is not completed yet.

7.1 Example

Consider the following example, where Q3 is defined in terms of Q1 and Q2. The relation Flight stores pairs of cities between which there is a direct flight and the relation Hub stores the set of hub cities of the airline. The query Q1 asks for pairs of cities between which there is a flight that goes through a hub. The query Q2 asks for pairs of cities that are on the same outgoing path from a hub.

```
Q1 (X,Y) :- Flight (X,Z), Hub (Z), Flight (Z,Y)
Q2 (X,Y) :- Hub (Z), Flight (Z,X), Flight (X,Y)
Q3 (X,Z) :- Q1 (X,Y), Q2 (Y, Z)
```

In our Lex code, we define identifiers as `Q3, Q1, Q2` and small and capital letters. Then we define tokens as `Q1, Q2, Q3, Flight, Hub, ( , ) , :-.`

Then in Yacc generator, tokens are again declared so that it can merge with lexical analyzer. In translation rule phase, we defined a set of production rules for parsing tree. We can successfully compile our code. As there is no error, we can assure that our grammatical rules are correct and we can use these rules for our future expansion. In syntax analysis, parse trees are used to show the structure of the sentence, but they often contain redundant information due to implicit definitions. Due to short time we could not make our parse tree for this experiment.

7.2 Conditional Query Example

The following query asks for the recruiters who performed the best or the worst:

```
Q1 (E,Y) :- Interview (X, D, Y, H, F),
            EmployeePerformance (E, Y, T, W, Z), W < 2.5
Q1 (E,Y) :- Interview (X, D, Y, H, F),
            EmployeePerformance (E, Y, T, W, Z), W > 3.9
```

Where we define identifiers as Q1 and small and capital letters. And `Q1, Interview, EmployeePerformance, W` are tokens.

CHAPTER 8
DISCUSSION AND CONCLUSION

8.1 Discussion

Lexical and syntax analysis can be simplified to a machine that takes in some program code and then returns syntax errors, parse trees and data structures. We can think of the process of description transformation, where we take some source description, apply a transformation technique and end up with a target description - this is inference mapping between two equivalent languages, where the destination is a machine executable.

Lexical analysis is the extraction of individual words or lexemes from an input stream of symbols and passing corresponding tokens back to the parser.

If we consider a statement in a programming language, we need to be able to recognise the small syntactic units (tokens) and pass this information to the parser. We need to also store the various attributes in the symbol or literal tables for later use, *e.g.*, if we have an variable, the tokeniser would generate the token var and then associate the name of the variable with it in the symbol table - in this case, the variable name is the lexeme.

Other roles of the lexical analyzer include the removal of whitespace and comments and handling compiler directives (*i.e.*, as a preprocessor).

In our code, the tokenization process takes input and then passes this input through a keyword recognizer, an identifier recognizer, a numeric constant recognizer and a string constant recognizer, each being put in to their own output based on disambiguating rules.

We can assure that our generated grammar rule will make an unambiguous parse tree and able to make simple and not redundant queries which is discussed in Section 8.1.1.

8.1.1 Future Expansion

As we already made our grammar rules in Yacc generator and tokenized in lexical analyzer and our rest of the work is unfolding Conjunctive Queries. So that, we have to make parse tree. Here, we gave a Figure 8.1, which is an example of simple calculator. The following parse tree corresponds to a leftmost derivation. This gives us a view of our future parse tree which is related to our experiment.

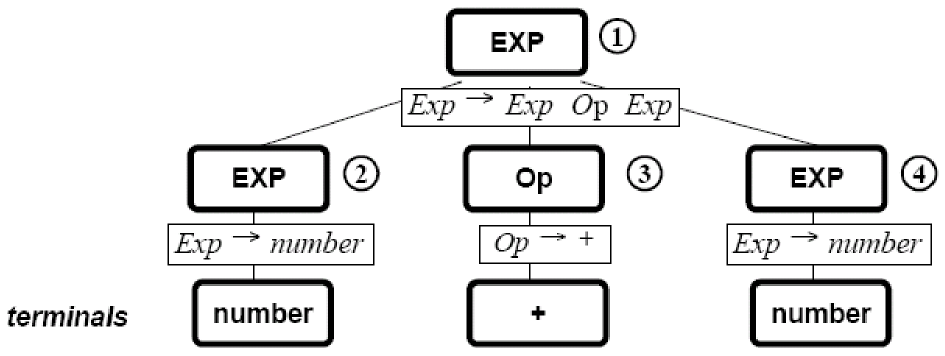

Figure 8.1: Parse Tree.

8.2 Conclusion

Structure Query Language is a widely-used programming language for working with relational databases. A relational database is a digital database whose organization is based on the relational model of data. We use two different notations for queries over relational databases throughout the thesis. The first is SQL, which is the language used to query relational data in commercial relational systems. The second is Conjunctive Query. A Conjunctive Query is a restricted form of first-order queries. Many first-order queries can be written as Conjunctive Queries. Datalog is a truly declarative logic programming language that syntactically is a subset of Prolog.

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. From source code, lexical analysis produces tokens, the words in a language, which are then parsed to produce a syntax tree, which checks that tokens conform with the rules of a language.

Yacc generator is then performed on the syntax tree to produce an annotated tree. In addition to this, a literal table, which contains information on the strings and constants used in the program and a symbol table, which stores information on the identifiers occurring in the program (*e.g.*, variable names, constant names, procedure names, etc), are produced by the various stages of the process. An error handler also exists to catch any errors generated by any stages of the program.

Our goal was to unfold Conjunctive Queries. We successfully implemented our production rules and tokenization process. The rest of our task is to implement unfolding Conjunctive Queries with C language and merge with Yacc generator. So, we are hopeful that we will be able to unfold Conjunctive Queries in future.

REFERENCES

[1] "Sql." http://en.wikipedia.org/wiki/SQL. last accessed December 09 2014.

[2] E. F. Codd, "Information retrieval P. BAXENDALE, editor A relational model of data for large shared data banks," Apr. 02 2008.

[3] "Lexical analysis." http://en.wikipedia.org/wiki/Lexical_analysis. last accessed December 10 2014.

[4] A. Doan, A. Y. Halevy, and Z. G. Ives, *Principles of Data Integration*. Morgan Kaufmann, 2012.

[5] A. Silberschatz, H. F. Korth, and S. Sudarshan, *"Database System Concepts"*. McGraw-Hill Co., 3rd ed., 1997.

[6] "Codd's theorem." http://en.wikipedia.org/wiki/Codd%27s_theorem. last accessed December 09 2014.

[7] I. Horrocks and S. Tessaris, "A conjunctive query language for description logic aboxes," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA* (H. A. Kautz and B. W. Porter, eds.), pp. 399–404, AAAI Press / The MIT Press, 2000.

[8] R. A. Kowalski, "Logic for data description," in *Logic and Data Bases*, pp. 77–103, 1977.

[9] M. E. Lesk and E. Schmidt, "LEX—Lexical Analyzer Generator." Unix Programmer's Manual.

[10] K. J. Gough, *Syntax analysis and software tools*. Reading, MA: Addison-Wesley, 1988.

[11] "Parsing." http://en.wikipedia.org/wiki/Parsing. last accessed December 10 2014.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, eds., *Compilers: principles, techniques, and tools*. Boston, MA, USA: Pearson/Addison Wesley, second ed., 2007.

[13] J. Oncina, "The cocke-younger-kasami algorithm for cyclic strings," in *ICPR*, pp. II: 413–416, 1996.

[14] J. Aycock and R. N. Horspool, "Practical earley parsing," *The Computer Journal*, vol. 45, no. 6, pp. 620–630, 2002.

APPENDIX A
CODES

A.1 Lexical Code

We use this code to take large queries as input by using lexical analyzer.

```
%{
#include <malloc.h>
#include <stdlib.h>
#include"y.tab.h"
%}
WS        [ \t]+
DIGIT     [0-9]
CAPITAL [A-Z]
SMALL     [a-z]
Q         Q{DIGIT}+
%%
{Q}          {return Q;}
{SMALL}+     {return SMALL;}
{CAPITAL}    {return CAPITAL;}
"("          {return LPAREN;}
")"          {return RPAREN;}
":-"         {return COLONDASH;}
","          {return COMMA;}
"\n"         {return NEWLINE;}
{WS}         {}
%%
main()
{
freopen("input.txt","r",stdin);
yylex();
}
```

A.2 Yacc Code

We use this code to tokenize and parsing by using syntax analyzer.

```
%{
#include<stdio.h>
#include<stdlib.h>
extern yylex();
extern yyerror();
%}

%token  COLONDASH COMMA LPAREN RPAREN SMALL CAPITAL Q NEWLINE

%%
input:                  /* empty string */
        |input line

line:   NEWLINE
        |predicate NEWLINE

sub:    SMALL LPAREN CAPITAL COMMA CAPITAL RPAREN

predicate : Q LPAREN CAPITAL COMMA CAPITAL RPAREN COLONDASH
         sub
        |Q  LPAREN CAPITAL COMMA CAPITAL RPAREN COLONDASH
         sub COMMA Q  LPAREN CAPITAL COMMA CAPITAL RPAREN
        |Q  LPAREN CAPITAL COMMA CAPITAL RPAREN COLONDASH Q
         LPAREN CAPITAL COMMA CAPITAL RPAREN
        |Q  LPAREN CAPITAL COMMA CAPITAL RPAREN COLONDASH Q
         LPAREN CAPITAL COMMA CAPITAL RPAREN COMMA sub
%%

main()
{
        yyparse();
        exit(0);
}
```