On The Design and Implementation

of a Top-Down Datalog Interpreter in C++

Mohan Rao Tadisetty

A Major Project Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Masters in Computer Science at
Concordia University
Montreal, Quebec, Canada

July 1997

0-612-40235-5

ABSTRACT

On The Design and Implementation of a Top-Down Datalog Interpreter in C++

Mohan Rao Tadisetty

Datalog is a database query language based on the *logic programming paradigm*. *Datalog* is the language of deductive databases, obtained by extending the basic relational database model with the reasoning capability, that is, one can not only query about the facts stored explicitly in the database but also query about derived facts. *Datalog* provides the clauses with parameters, called logical variables. The interpreter for *Datalog* requires the matching of predicates and of logical variables, through *unification* and *substitution*. *Datalog* behaves like a programming language because it can return values as answers to queries, rather than just "yes" and "no" answers. In this report, the syntax and semantics of *Datalog*, the efforts to design and implement a top-down version of the *Datalog interpreter* in C++ and the experimental results are presented.

The name "***Datalog***" is chosen because of its connection with database query languages. In this implementation, the *Datalog interpreter* can process one query at a time.

ACKNOWLDEGEMENTS

I am deeply indebted to my advisor ***Professor Gregory Butler*** for introducing me to the Object Oriented Databases, Object Oriented Design concepts, programming and Deductive Databases. I am grateful to him for his excellent guidance, valuable suggestions, encouragement and the time he spent in carefully reading the manuscript. His in-depth knowledge of the fundamental issues and clear vision of the underlying nature of research has not only helped me in the preparation of this project but has also helped me acquire the proper approach for Datalog Interpreter design and implementation.

Special thanks are to my wife Uma Maheswari for her constant encouragement and moral support, to whom I dedicate this work. Finally I wish to thank my parents and friends.

Table of Contents

Table of Contents Continued ....

List Of Figures

1.0 INTRODUCTION

At an abstract level, mathematical logic provides a uniform framework for the expression and manipulation of information. One of its greatest strengths, from the point of view of computer science, is that the manipulation of information can be given semantics which is declarative. That is, the semantics can be expressed without reference to a sequence of operations. Research in the field of logic programming is concerned with developing logic-based programming systems which manipulate data efficiently. Prolog is a logic programming language which has been successfully used as a general programming language[9].

Techniques have been developed for traditional database query systems to manipulate large amounts of information very efficiently. The way information is handled in these systems can be expressed by a subset of logic. These systems typically allow the information to be transformed using a fixed set of operations, but fall short of providing a general computational mechanism for transforming data; for example, it is not possible to express transitive closure of a relation in a traditional database system. *Deductive databases* extend the expressive power of database systems by adding recursion[9]. At a semantic level they are equivalent to logic programs; operationally, however, a query can be processed using either a top-down or bottom-up computation method. These two methods are the extremes of the range of computation methods that might be employed by a deductive database system.

Recent years have seen substantial efforts in the direction of *merging* artificial intelligence and database technologies for the development of large and *persistent knowl-*

edge bases[5]. A *persistent knowledge base* is one whose data is stored on the disk. In other words, after leaving the program, the relations can be accessed again. An important contribution towards this goal comes from the *integration* of logic programming and databases. The focus has been concentrated mostly by the database theory community on well-formalized issues, like the definition of a new rule-based language, called ***Datalog***, which is designed specifically for interacting with large databases, and the definition of optimization methods for various types of Datalog rules, together with the study of their efficiency. In parallel, various experimental projects have shown the feasibility of Datalog programming environments.

Present efforts in the integration of artificial intelligence(**AI**) and databases(**DBs**) take a much more pragmatic approach; in particular, several attempts fall in the category of "*loose coupling*", where existing ***AI*** and ***DB*** environments are interconnected through ad-hoc interfaces. In other cases, *AI* systems have solved persistency issues by developing internal databases for their tools; but these internal databases typically do not allow *data sharing* and *recovery*; thus they do not properly belong to current database technology. The spread and success of such enhanced *AI* systems, however, indicate that there is a great need for them. Loose coupling has been attempted in the area of Logic Programming and databases by interconnecting Prolog systems to relational databases[11]. Most studies indicate that simple interfaces are too inefficient; an enhancement in efficiency is achieved by intelligent interfaces. This indicates that loose coupling might solve today's problems, but in the future, strong integration will be required. More generally, we expect that *knowledge base management systems* will provide *direct access* to data and will support *rule-based interaction* as one of the programming paradigms. Datalog is a first step in this direction[5].

2.0 Datalog Interpreter

2.1 The Syntax of Datalog Programs

Datalog is in many respects a *simplified* version of general Logic Programming. A logic program consists of a finite set of *facts* and *rules*. *Facts* are assertions about a relevant piece of the world, such as: "*John is the father of Harry*". *Rules* are sentences which allow us to deduce facts from other facts. An example of a *rule* is: "*If X is a parent of Y and if Y is a parent of Z, then X is a grandparent of Z*". The rules, in order to be general, usually contain universally quantified variables (*X, Y, Z etc.,*). Both facts and rules are particular forms of knowledge. In the formalism of *Datalog*, both facts and rules are represented as *Horn clauses* of the general type L_0 :- L_1, ... L_n, where L_i is a *literal* of the form $p_i(t_1, t_2, ..., t_k)$ such that p_i is a ***predicate*** symbol and t_j are *terms*. A *term* is either a *constant* or a *variable*. The left-hand side of a Datalog clause is called its ***head*** and right hand side is called its ***body***. The *body* of a clause may be *empty*. Clauses with an empty body represent ***facts***; clauses with at least one literal in the body represent ***rules***. A fact should be a ground atom, that is, there should be no variables in the terms. The fact "*John is the father of Bob*", for example, can be represented as ***father(bob, john)***. The rule "*If X is a parent of Y and, if Y is a parent of Z, then X is a grandparent of Z*" can be represented as *grandpar(Z,X) :- par(Y,X), par(Z,Y).*

Here the symbols *par* and *grandpar* are predicate symbols, the symbol *john* and *bob* are constants, and *X*,*Y* and *Z* are *variables*. *Constants* and *predicate symbols* are strings beginning with an *lower-case* letter. For a given Datalog program, it is always clear from the context whether a particular non variable symbol is a constant or a predicate symbol. Variable symbols begin with an upper-case letter. Also, Datalog requires that all literals with the same predicate symbol are of the *same* arity, that is, that they have the same number of arguments. A literal, fact, rule, or clause which does not contain any variables is called ground. Any Datalog program ***P*** must satisfy the following safety conditions: ***1)*** Each fact of ***P*** is a ground atom; ***2)*** Each variable which occurs in the head of a rule of ***P*** must also occur in the body of the same rule. These conditions guarantee that the set of all facts that can be derived from a Datalog program is *finite*.

2.2 Datalog and Relational Databases

In general, logic programming it is usually assumed that all the *knowledge*(facts and rules) relevant to a particular application is contained within a single logic program ***P***. Datalog, on the other hand, has been developed for applications which use a large number of facts stored in a relational database[5]. Therefore, we will always consider two sets of clauses in ***P*** : a set of ground atoms, called the *Extensional Database* (***EDB***), physically stored in a relational database, and a set of rules called the *Intensional database*(***IDB***). The predicates occurring in ***P*** are divided into two disjoint sets: the *EDB-Predicates*, which are those occurring in the Extensional database, and the *IDB-predicates*, which occur in ***P*** but not in the ***EDB***. We require that the head predicate of each clause in ***P*** be an *IDB-predicate*. *EDB-predicates* occur *only* in clause bodies.

Ground atoms are stored in a relational database; we assume that *each EDB-predicate* ***r*** corresponds to exactly one relation ***R*** of our database such that each fact $r(c_1, \ldots, c_k)$ of the *EDB* is stored as a tuple $<c_1, \ldots, c_n>$ of ***R***. Also the *IDB-predicates* of ***P*** can be identified with relations, called *IDB-relations*, also called derived relations, defined by the rules in ***P*** and the *EDB*. *IDB* relations are not stored explicitly; they correspond to *relational views*. The *materialization* of these views, that is, their effective (and efficient) computation, is the *main task* of a *Datalog interpreter*.

As an example of a relational *EDB*, consider a database E_1 consisting of two relations with respective schemes ***PERSON****(NAME)* and ***PAR****(CHILD, PARENT)*. The first contains the names of persons and the second expresses a parent relationship between persons. Let the actual instances of these relations have the following values:

PERSON = *{ < ann >, < bertrand >, < charles >, < david >, < evelyn >, < fred >,*
< george >, < hanson > }

PAR = *{ < david, george >, < evelyn, george >, < bertrand, david >,*
< ann, david >, < ann, hanson >, < charles, evelyn > }

The *PAR Relationship Tree* is shown in ***Figure 1***. These relations express the set of ground atoms :

E_1 = *{ person(ann), person(bertrand), ..., par(david,george), ... ,*
par(charles, evelyn)}

So E_1 = PERSON $\cup$ PAR

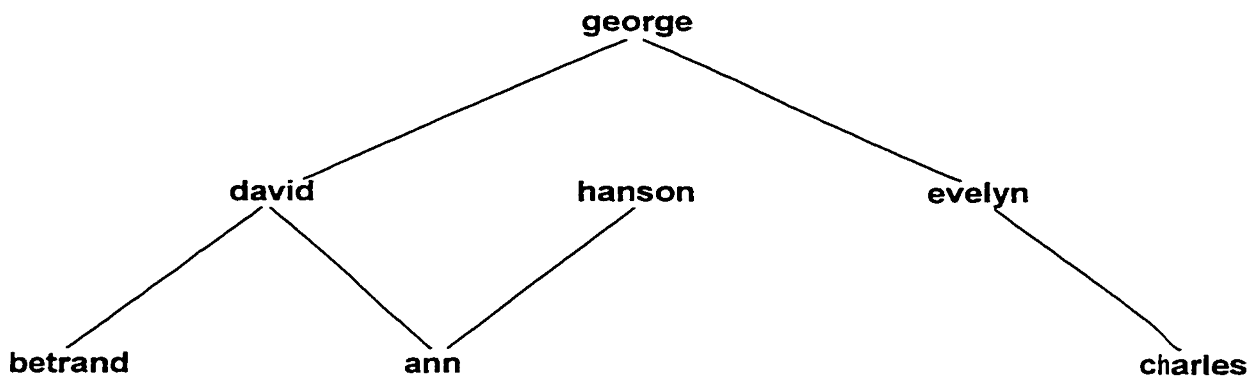

Figure 1: PAR Relationship Tree

Let P_1 be a Datalog Program consisting of the following clauses:

r1 : sgc(X, X) :- person(X).

r2 : sgc(X,Y) :- par(X,X1), sgc(X1, Y1), par(Y, Y1).

Due to rule *r1*, the derived relation *SGC* (*Same Generation Cousins*) will contain a tuple <*p, p*> for each person *p*. Rule *r2* is recursive and states that two persons are same generation cousins whenever they have parents which are in turn same generation cousins. The complete list of all tuples in the derived relation *SGC* are:

< *george, george* >,	< *david, david* >,	< *hanson, hanson* >,	< *evelyn, evelyn* >,
< *betrand, betrand* >,	< *ann, ann* >,	< *charles,charles* >,	< *david, evelyn* >,
< *evelyn, david* >,	< *betrand, ann* >,	< *ann, betrand* >,	< *ann, charles* >,
< *charles, ann* >,	< *betrand, charles* >	and	< *charles, betrand* >.

The program $\boldsymbol{P_1}$ can be considered as a query against the *EDB* $\boldsymbol{E_1}$, producing tuple answers in the relation *SGC*. In this setting, the distinction between the two sets of clauses, $\boldsymbol{E_1}$ and $\boldsymbol{P_1}$, makes yet more sense, because a query can be viewed as a function applied to the IDB to compute an instance of EDB. Usually a database(in our case the *EDB*) is considered as a time-varying collection of information. A query(in our case, a program ***P***), on the other hand, is a time-invariant mapping which associates a result to each possible database state. For this reason, we will formally define the semantics of a Datalog program ***P*** as a *mapping* from database states to result states. The database states are collections of *EDB-facts* and the result states are *IDB-facts*.

Usually Datalog programs define large IDB-relations. It often happens that a user is interested in a subset of these relations. For instance, one might want to know the same generation cousins of *ann* rather than all the same generation cousins of all persons in the database. To express such an additional constraint, one can specify a goal to a Datalog program. A *goal* is a single literal. Goals usually serve to formulate ad hoc queries against a view defined by a Datalog program. For example, the goal *?-sgc(a,X)*, (to get all the tuples of same generation cousins of *ann)*, when submitted to a Datalog interpreter yields the tuples *<ann, ann>*, *<ann, betrand>* and *<ann, charles>* as the answers.

2.3 Top-Down Evaluation of Datalog Goals

The *top-down* method one way of *evaluating* Datalog programs. Proof trees are constructed from the top to the bottom[5]. This method is particularly appropriate when a goal is specified together with a Datalog program. Consider the program $\boldsymbol{P_1}$ and the *EDB*

E_1 of our "same generation" example. Assume that the goal *?-sgc(ann, X),* (to get all names which are same generation cousins of *ann)* is specified. One way to find the required answers is to compute first the entire relation *sgc(X, X)* by bottom-up derivation from the EDB and then delete all facts in SGC which are not subsumed by our goal and then project onto the second attribute position. This would be a waste, since we would derive many more facts than necessary. The other possibility is to start with the goal and construct proof trees from the top to the bottom by applying the *Elementary Production Principle*(**EPP**) "*backwards*", similar to resolution-based theorem provers. **EPP** resolution refers to a general inference rule, which produces new Datalog facts from given Datalog rules and facts. Such methods are also referred to as *backward chaining.* **EPP** can be considered as being a meta-rule, since it is independent of any particular Datalog rules, and treats them just as syntactic entities. We present the top-down method for evaluating Datalog programs against an *EDB.* This method, called *Query-subquery approach*(***QSQ***), implicitly constructs all proof trees for a given goal in a *recursive* fashion.

2.4 Datalog Is a Database Language

Although expressing queries and views in Datalog is quite intuitive and fascinating from a user's view point, we should not forget that the *aim* of database query languages like Datalog is providing access to large quantities of data stored in mass memory. Thus, in order to enable an easy integration of Datalog with database management systems, we need to *relate* the logic programming formalism to a data retrieval language. We have chosen relational algebra as such a data retrieval language. This following section provides an informal description of the *translation* of Datalog programs and goals into relational algebra.

2.4.1 Translation of Datalog Queries into Relational Algebra

Each clause of a Datalog program is translated, by a syntax-directed translation algorithm, into an inclusion relationship in Relational Algebra(***RA***). The set of inclusion relationships that refer to the same predicate is then interpreted as an equation of relational algebra. Thus, we say that a Datalog program gives rise to a system of algebraic equations. Each *IDB-predicate* of the Datalog program corresponds to a variable relation; each *EDB-predicate* of the Datalog program corresponds to a constant relation. Determining a solution of the system corresponds to determining the value of the variable relations which satisfy the system of equations. The translation from Datalog to relational algebra[19] is described in the following paragraphs.

Relational Algebra is a system of operators that take one or two relations as arguments and return a relation as a result. *Select* (σ), *Project* (Π), and *Join* ($\bowtie$) are the fundamental operators of relational algebra. ***Select*** pulls out a subset of the tuples in a relation based on some selection condition. A *Selection* condition is a comparison between an attribute and a constant or between two attributes. *Selection* is denoted by σ, with the selection condition as a subscript. ***Project*** extracts a subset of the columns of a relation, rather than a subset of the tuples. *Project* is denoted by a Π with a subscript giving the attributes for the columns to be retained. ***Join*** (sometimes called *natural join*) combines two relations on the common attributes in their schemes. A tuple *t* is in the join of relations *r* and *s* if t agrees with some tuple in *r* on the scheme of *r*, and with some tuple in *s* on the scheme of s. The result of a join, then has a scheme that is the union of the schemes of the relations specified by the join's arguments.

Let us consider a Datalog clause **C:**

$$p(\alpha_1, \alpha_2, \ldots, \alpha_n) :- q_1(\beta_1, \ldots, \beta_k), \ldots\ldots\ q_m(\beta_s, \ldots, \beta_h).$$

The translation associates with **C** an inclusion relationship $Expr(Q_1, \ldots, Q_m) \subseteq P$, among the relations $P, Q_1, \ldots, Q_m$ that correspond to predicates $p, q_1, \ldots, q_m$, with the convention that relation attributes are named by the number of the corresponding argument in the related predicate. For example, the Datalog rules of the program ***P1*** from *Section 2.2:*

r1 : sgc(X, X) :- person(X).

r2 : sgc(X, Y) :- par(X, X1), sgc(X1, Y1), par(Y, Y1).

are translated into the inclusion relationships :

$$\Pi_{1,5}\ ((PAR \underset{2=1}{\bowtie} SGC) \underset{4=2}{\bowtie} PAR) \subseteq SGC \qquad \ldots\ldots\ldots\ldots\ldots\ldots \mathit{1}$$

$$\Pi_{1,1}\ PERSON \subseteq SGC \qquad \ldots\ldots\ldots\ldots\ldots\ldots \mathit{2}$$

The relationships *1* and *2* are Relational Algebraic expressions where $\bowtie$ denotes the *natural join* operation and Π denotes the *projection* operation and 1,5 in Π denotes the attribute number in the argument relation, that is, join *PAR* and *SGC* and *PAR* in this order and then *project* on columns 1 and 5. Similarly $\Pi_{1,1}$ *PERSON* defines a binary relation of the form (X,X), $\forall x \in PERSON$.

The rationale of the translation is that literals with common variables give rise to *joins*, while the head literal determines the *projection*. In order to obtain a two-column relation *SGC* in the second inclusion relationship, we have performed a double projection of the unique column of relation *PERSON*. For each *IDB* predicate *p*, we now collect all the inclusion relationships of the type $Expr_i(Q_1, \ldots, Q_m) \subseteq P$, and generate an algebraic equation having *P* as LHS, and the union of all the left-hand sides of the inclusion relationships as RHS:

$$P = Expr_i(Q_1, \ldots, Q_m) \cup Expr_2(Q_1, \ldots, Q_m) \cup Expr_{mp}(Q_1, \ldots, Q_m)$$

We also translate logic goals into algebraic queries. Input Datalog goals are translated into projections and selections over one variable relation of the system of algebraic equations. For example, the goal *"?-p(X)."* is equivalent to the algebraic query *"P"*, and *"?-q(a,X)"* which is equivalent to " $\sigma_{1=a} Q$".

2.5 The Expressive Power of Datalog

The system of equations produced by the above translation uses all the classical relational operations, with the exception of difference; we say that it is written in positive *relational algebra*, $\boldsymbol{RA^+}$[5]. It can be easily shown that each defining expression of RA^+ can also be translated into a Datalog program. This means that Datalog is at least expressive as RA^+; in fact, Datalog is strictly more expressive than RA^+ because in Datalog it is possible to express recursive queries, which are not expressible in RA^+. However, there are expressions in full relational algebra that cannot be expressed by Datalog programs. These are the queries that make use of the difference operator.

The relational algebra(*RA*) has negation but does not support recursion. On the other hand, Datalog has recursion but does not support negation. ***Figure 2*** graphically represents the situation, and illustrates the correspondence between *non-recursive Datalog* and *negation-free* subset of *relational algebra* RA^+. However, these expressions can

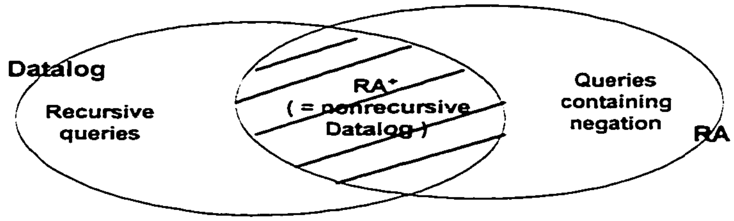

Figure 2 : The Expressive Power of Datalog

be captured by enriching pure Datalog with the use of logical negation(¬). Also, even though Datalog is syntactically a subset of first-order logic, strictly speaking they are not comparable. Indeed, the semantics of Datalog is based on the choice of a specific model (the *least Herbrand model*), while first-order logic does *not* a priori require a particular choice of the model.

2.6 Search Strategy

Evaluation of a Datalog goal can be performed in *two* different ways: *bottom-up*, starting from the existing facts and infering new facts, or *top-down*, trying to verify the premises which are needed in order for the conclusion to hold. In the AI literature, these are referred to as *forward-chaining* and *backward-chaining* respectively.

2.6.1 Bottom-up Evaluation Strategy

Bottom-up evaluations consider rules as productions. They apply the rules in a given program to the **EDB**, and produce all the possible consequences of the program, until no new fact can be deduced. Bottom-up methods can naturally be applied in a set-oriented fashion, that is, taking as input the entire relations of the **EDB**, using a relational database utility to retrieve large quantities of data from mass memory. On the other hand, bottom-up methods do not take immediate advantage of the selectivity due to the existence of to constants in the goal. The following example makes the bottom-up evaluation method more clear.

Example:

Suppose the query given is *?-sgc(ann, X)*, to get all names which are same generation cousins(*sgc*) of *ann*. Assume that *sgc* has a large number of tuples (related to this query) and only one of them belongs to the answer to this query. The bottom-up evaluation method computes all the tuples in *sgc* relations and at the end applies the selection operation to get the *sgc* of *ann*. This is wasteful, because the bottom-up evaluation method does not take advantage of tuple selection based on bound arguments in the goal.

2.6.2 Top-Down Evaluation Strategy

In *top-down* evaluation, rules are seen as problem generators[5]. Each goal is considered as a problem that must be solved. The initial goal is matched with the left-hand side of some rule, and generates other problems corresponding to the right-hand side predicates of that rule; this process is continued until no new problems are generated. In this case, if the goal contains some bound argument, then only facts that match the goal constants are involved in the computation. Thus, this evaluation mode already performs a relevant optimization because the computation automatically disregards many of the facts which are not useful in for producing the result. On the other hand, in top-down methods it is more natural to produce the answer one-tuple-at-a-time, and this is an undesirable feature in Datalog.

If we restrict our attention to top-down approach, we can further distinguish two search methods: *breadth-first* and *depth-first*. With the depth-first approach, we face the disadvantage that the order of literals in rule bodies strongly affect the performance of methods. This happens in Prolog, where not only efficiency, but even termination of programs is affected by the left-to-right order of subgoals in the rule bodies[9]. Instead, Datalog goals are executed through breadth-first techniques, as the result of the computation is neither affected by the order of predicates within right-hand sides of rules, nor by the order of rules within the program. The optimization methods should satisfy *three important properties*: *1) Methods must be sound*: they should not include in the result tuples which do not belong to it. *2) Methods must be complete* : they must produce all the tuples of the result. *3) Methods must terminate*: the computation should be performed in finite time.

Although we omit formal proofs [21, 22], the top-down efficient strategy called *Query-Subquery* presented in the next section satisfies the above properties.

2.7 Top-Down Evaluation

The *Query-Subquery(QSQ)* algorithm is a *top-down evaluation algorithm*, optimizing the behavior of *backward-chaining* methods. The objective of the *QSQ* method is to access the *minimum* number of facts needed in order to determine the answer. In order to do this, the fundamental notion of subquery is introduced. A goal, together with a program, determines a query. Literals in the body of any one of the rules defining the goal predicate are subgoals of the given goal. Thus, a subgoal, together with the program, yields a subquery; this definition applies recursively to sub-goals of rules which are subsequently activated. In order to answer the query, each goal is expanded in a list of subgoals, which are recursively expanded in turn.

Example:

For example, consider the EDB E_1 and the following Datalog rules from *Section 2.2* :

r1 : sgc(X, X) :- person(X).

r2 : sgc(X, Y) :- par(X, X1), sgc(X1, Y1), par(Y, Y1).

Suppose the given query is *?-sgc(ann, X)*, which gets all the same generation cousins (*sgc*) of *ann*. In a top-down evaluation, each goal is considered a problem/query that must be solved/answered. The top-down query processor tries to find those rules whose head *unifies* with the given goal. For instance, for the goal *sgc(ann, X)*, unifies with the head *sgc(X, X)* of rule *r1*, and yields the substitution *X* = *ann*. This leads to new goals in the rule body, that is, *person(ann)*, which is true, since it is given as a fact. Then the query processor explores remaining rules to find other possible answers to the query *sgc(ann, X)*. In this case, it *unifies* the goal with the head *sgc(X, Y)* of rule *r2*, producing a new goal list *par(ann, X1), sgc(X1, Y1)* and *par(X, Y1)*. Each of these goals are processed as described above, in the left-to-right order. Note that during query processing, the top-down query processor may need to *backtrack*, that is, during the exploration of the proof tree, if the top-down query processor encounters a goal that can not be established, it retraces its own course by going backwards along the last tree branch and resumes traversal by trying to re-satisfy the goal to the left of the one just failed.

The method maintains *two* sets: a set ***P*** of answer tuples, containing answers to the main goal and answers to intermediate subqueries, which is represented by a set of temporary relations (one relation for each IDB-predicate); and a set ***Q*** of current subqueries (or subquery instances), which contains all the subgoals that are currently under consideration. Thus the function of *QSQ* algorithm is twofold: generating new answers and generating new subqueries that must be answered. There are two versions of the *QSQ* algorithm, an *iterative* one(***QSQI***) and a *recursive* one (***QSQR***). *QSQI* uses breadth-first strategy and *QSQR* uses depth-first strategy. The difference between the two concerns which of these two functions has priority over the other: *QSQI* favours the production of answers, thus, when a new subquery is encountered, it is suspended until the end of the production of all the possible answers that do not require using the new subquery. *QSQR* behaves differently: whenever a new subquery is found, it is recursively expanded and the answering to the current subquery is postponed to when the new subquery has been completely solved. At the end of the computation, ***P*** includes the answer to the goal.

2.8 Query Processing

Different databases and application domains - such as business accounts, engineering designs, geometric and graphic data, text documents, scientific data, and hypermedia documents - have different requirements as to storage and retrieval capabilities.

We discuss the types of queries briefly. Note that Datalog only solves exact match and partial match queries. Some types of queries include :

a) ***Exact Match Queries*** : Specify a literal value (also called a ground value) for an attribute, and a match of that value is expected. A predicate(or relation) may have several attributes. A fully ground query specifies a literal value for each attribute, and requires confirmation whether this fact is in the database. An exact match may also refer to the case where a literal value is given for the primary key of a relation, and the retrieval of the complete record with the given key is required.

b) ***Partial Match Queries*** : Specify a literal value for some attributes, and a partial match is required - that is, a match is required for each of the attributes that have a literal value specified, but the other attributes can have any value: that is, the attributes match a "wild card".

c) ***Range Queries*** : Specify a range of values for an attribute. The ranges may be

- an open interval range, such as $low < attribute < high$;
- a closed interval range, $low \leq attribute \leq high$;
- a half-open interval range, such as $low \leq attribute \leq high$; or $low < attribute \leq high$;
- a semi-infinite interval range, such as $low < attribute$ or $attribute < high$;

d) ***Best Match Queries*** : Specify a literal value for some attributes but do not require that an exact match for each of the specified literals be found. In the event that such an exact match does not exist in the database, then the fact in the database which comes "nearest" to matching the query should be retrieved. One metric for "nearness" is the number of attributes whose value matches the value specified in the query.

e) ***Other Queries*** : Covers a broad range of queries such as : String matching query in a textual database and Boolean property query where the attributes take only boolean values, which indicate the presence or absence of some property. This list should be extended to include navigational queries common in object-oriented databases, and in libraries providing persistence.

The query(goal) itself will be in the form of a literal-list. The parser checks the syntax of the query and produces a parse tree. The parse tree denotes the type of every element of the query in order to check the validity of the query. This parse tree forms the *Datalog query*. The Datalog interpreter accepts and validates this query on a database and returns the solution.

3.0 Object Oriented Design of A Top-Down Datalog Interpreter

In this chapter, we introduce the object oriented architectural design of a top-down Datalog interpreter. Based on this design, we have developed an implementation, which can be found in *Appendix-A*.

The architectural overview of the *Datalog* interpreter is presented in ***Figure 3***. A formal discussion on *Datalog* semantics and the *Datalog* queries is given in *Section 2.0*. The **DDL**(Data Definition Language) allows the definition of rules and facts as clauses. The **DML** *(Data Manipulation Language)* only allows queries, which are posed as (headless) clauses: that is, a list of literals [2, 3].

3.1 Structural Overview

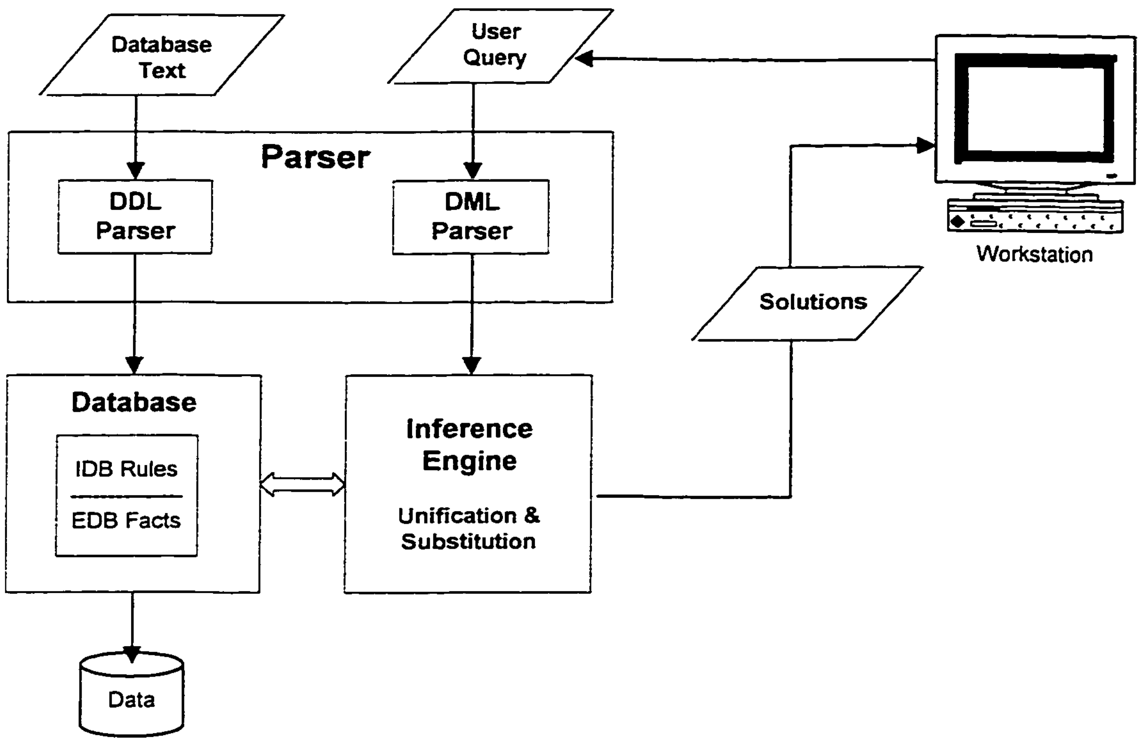

Figure 3: Overview of Datalog Architecture

3.2 Data Dictionary for Datalog

- ***Atomic Constant***

It is a primitive value and is indicated by an identifier which start with a *lower* case letter.

- ***Binding***

Associates a variable V with a value, which may be either a constant or another variable W. If V is bound to W and W is bound to a value, then both variables share the same value. A variable may be unbound; that is, not associated with any value. A binding is part of a substitution.

- ***Body***

  It is list of literals and forms part of a clause.

- ***Clause***

  It consists of a *head* and a *body*. A clause could be read as rule, "if the body is true then the head is true". It is one part of the definition of the predicate of the head. The definition of the predicate is the "or" of each of the clauses.

- ***Constant***

  Same as Atomic Constant.

- ***Extensional Data Base (EDB)***

  It is the collection of the facts explicitly stated as part of the database.

- ***Fact***

  It a *clause* which has an *empty body*. A fact may contain variables as arguments, but more often a fact is fully ground: that is, all arguments are constants.

- ***Goal***

  It is same as Query.

- ***Head***

  It is a literal, and forms part of a clause.

- ***Intensional Data Base (IDB)***

  It is a collection of clauses which define those facts which may be derived form the *EDB*. Often a program is viewed as precisely that part of the *IDB* needed to answer a specific query.

- ***Literal***

It is a reference to a predicate, which specifies the arguments of the predicate as either constants or variables.

- ***Predicate***

It is a relation.

- ***Predicate Name***

The symbolic name of a predicate.

- ***Program***

It is a set of clauses(or equivalently predicates) which define the relationship between the predicates in a query and those in the database.

- ***Query***

It is a list of literals. (like clause with an empty head)

- ***Relation***

Same as predicate.

- ***Solution***

It is a *set* of all facts that can be derived from the database, and that *satisfy* the query. The solution set may also be viewed as a set of *substitutions* for the variables in the query.

- ***Substitution***

It is a collection of bindings.

- ***Symbol***

Same as atomic constant.

- ***Unification***

It is a process which matches literals. Unification *determines* the most general substitution, called the most general unifier, which, when applied to the littorals being matched, gives an identical literal.

- ***Variable***

It stands as a place for values. It may be bound to different values, or be unbound. A variable is indicated by an identifier which starts with an upper case.

3.3 Grammar for Datalog Language

The *grammar* for the Datalog language is given below in ***Figure 4.*** The input of the database and the queries is translated by a recursive descent parser[1]. The *inference engine* for the interpreter uses backward-chaining with unification to process the *IDB* rules, and for partial match retrieval of the *EDB* facts. The retrieval of the facts is implemented using a simple list structure when facts are stored in memory.

Database	*::= database*	*| empty*
clause	*::= head :- body.*	*| head.*
head	*::= literal*	*| empty*
body	*::= literal.*	*| literal-list.*
literal	*::= predicate-name (argument-list)*	
argument	*::= constant*	*| variable*
query	*::= literal*	*| literal-list*

Figure 4: Grammar for Datalog Language

3.4 Classes And Data Structures

An ***Object model*** of the *Datalog language* concepts is presented in ***Figure 5*** and ***Figure 6***. The object model of the *Datalog interpreter* is presented in ***Figure 7***.

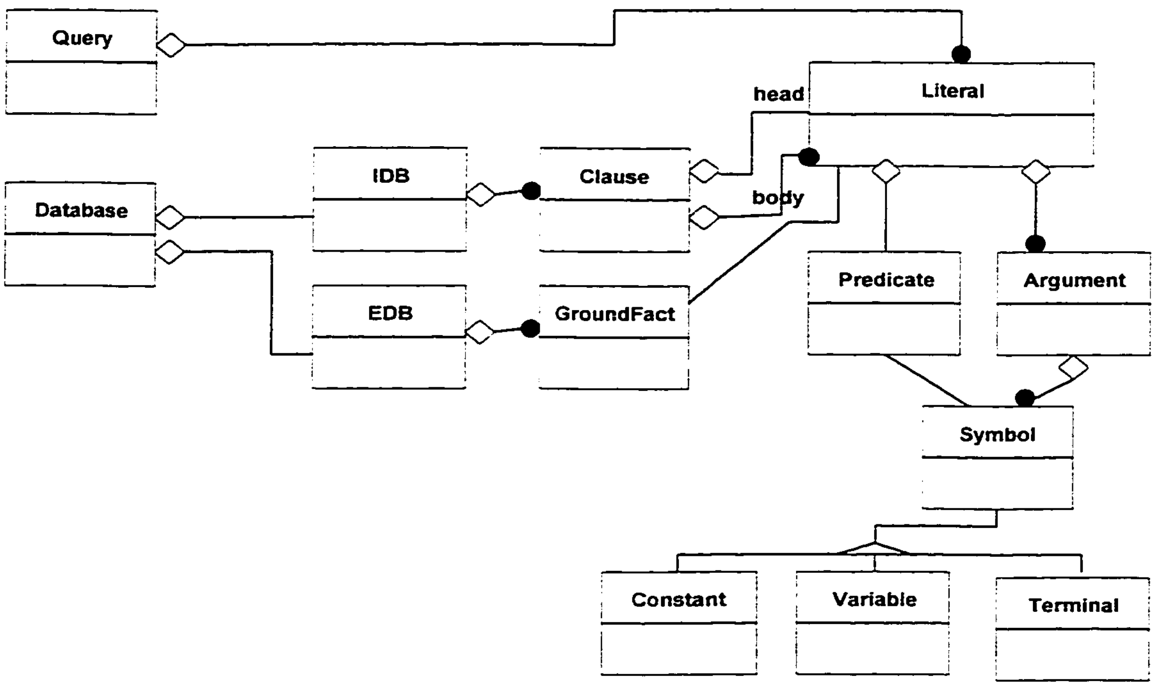

Figure 5: Datalog Object Model

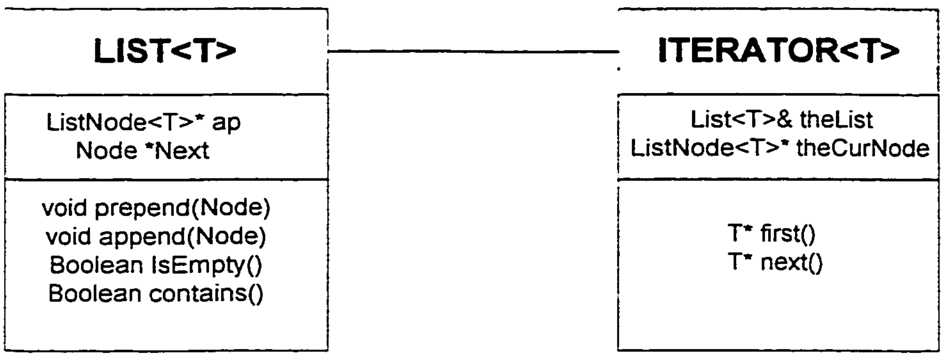

Figure 6: Object Model Details for List and ListIterator

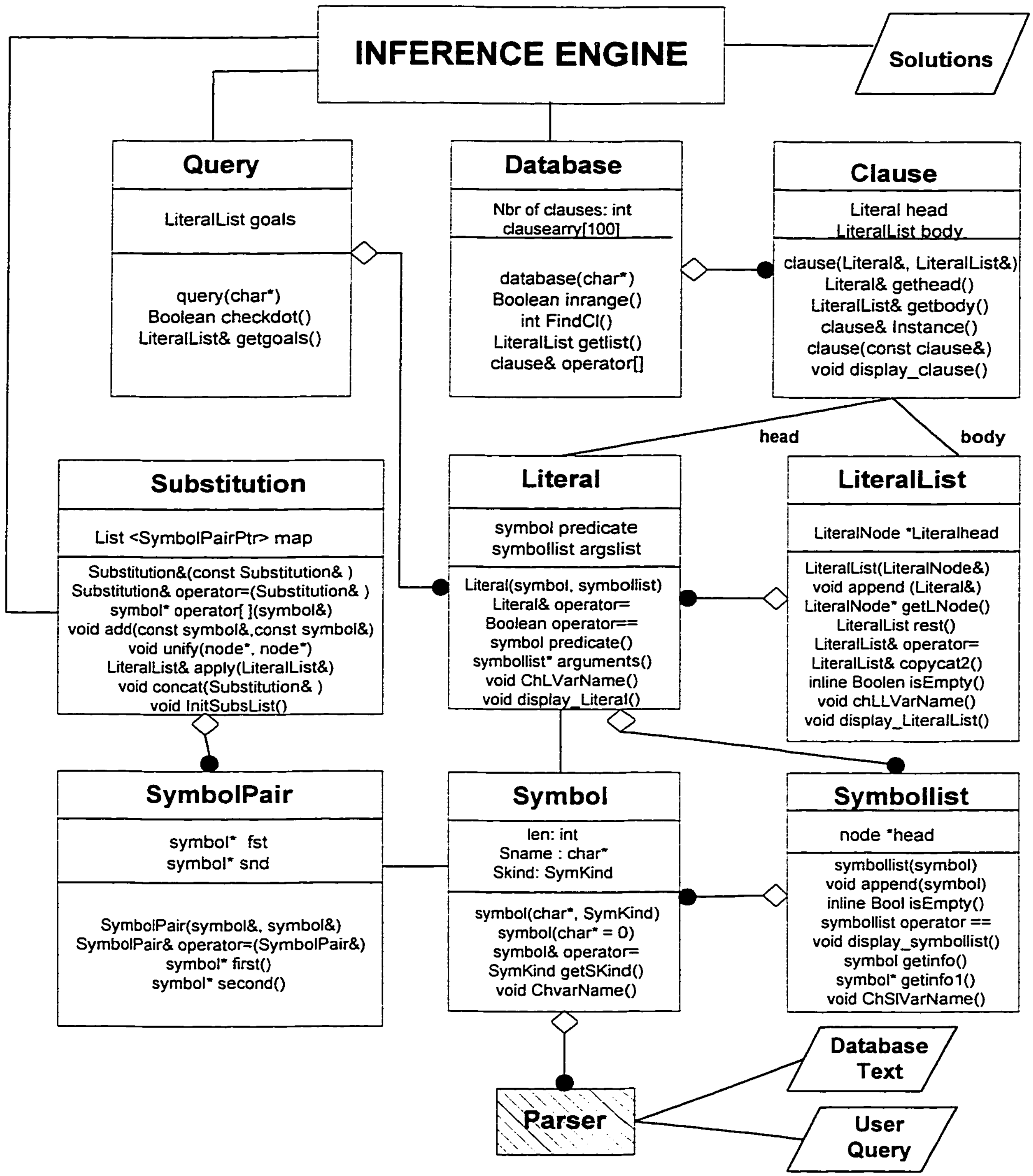

Figure 7: Datalog Interpreter Object Model Details

The following main classes are identified. The following paragraphs describe the data structures, class descriptions and the methods associated with each class. For brevity, the standard methods for each class such as constructors, destructors, copy constructors and assignment operators, are not discussed.

3.4.1 Class SYMBOL

3.4.1.1 Data Structure

- Len: Length of the string (type integer)
- Sname: Symbol Name (character string)
- Skind: Symbol Kind (enumeration: PRED, VAR , CONST)

3.4.1.2 Description

To store a Predicate/ Variable/ Constant symbol.

3.4.1.3 Methods

1. getSkind: Returns the SymKind of a given symbol.
2. ChVarName(): Changes the name of the symbol Variable to a new name

3.4.1.4. Friend Functions

A *friend function* can *access* a class's *private* data, even though it is not a member function of the class[18]. This is useful when one function must have access to two or more unrelated classes and when an overloaded operator must use, on its left side, a value of a class other than the one of which it is a member. Friends are also used to facilitate functional notation.

The Input/Output operators, *operator<<()*, *operator>>()* functions must be friends of the symbol class, since the *istream* and *ostream* objects appear on the left side of the operator. The *operator >>()* function takes an istream object, which will usually be *cin*, as its first argument, and an object of the symbol class as its second. It returns an *istream* so that the operator can be chained [12].

The *operator<<()* function is constructed similarly but uses ostream instead of *istream*. For similar reasons, the *operator==()* and *operator!=()* should be defined as *friends* of class Symbol.

ostream& operator<<(ostream&, const symbol&) : is an Output Operator

istream& operator>>(istream&, const symbol&) : is an Input Operator

Boolean operator==(const symbol&, const symbol&) : returns TRUE if the two given symbols are same and of same SymKind.

Boolean operator!=(const symbol&, const symbol&) : returns TRUE if the two given symbols are NOT same OR not of same SymKind.

3.4.2 Class NODE

3.4.1.1 Data Structure

- info: Symbol string
- next: Pointer to next node.

3.4.2.2 Description

To store a Symbol and a pointer to next Symbol.

3.4.2.3 Methods

1. node(const symbol&, node*) : creates a node for given symbol and with a given pointer to the next node.

3.4.2.4. Friend Class: class symbollist; - provides access to the private data of class symbollist.

3.4.3 Class SYMBOLLIST

3.4.4.1 Data Structure

- head: Pointer to the first node

3.4.4.2 Description

To maintain and manage a linked list of symbols.

3.4.4.3 Methods

1. append() : Appends a node at the end.
2. getnext() : Returns the pointer to the next node in the symbollist.
3. isEmpty() : checks whether the symbollist is empty or not.
4. display_symbollist(): Displays the symbollist.
5. getinfo() : Returns the pointer to the symbol in the node.
6. ChSlVarName() : Changes all the Variable names to new ones.

3.4.4.4 Friend Functions

istream& operator>>(istream&, const symbollist&) : Input Operator.

3.4.4 Class QUERY

3.4.4.1 Data Structure

- goals: query of type LiteralList

3.4.4.2 Description

To read the query from the file and to validate and build the query.

3.4.4.3 Methods

1. checkdot() : Senses the query end.
2. getgoals() : Returns the pointer to goals of type LiteralList.

3.4.4.4 Friend Functions :

istream& operator>>(istream&, const symbollist&) : Input Operator

3.4.5 Class LITERAL

3.4.5.1 Data Structure

- predicate : Name of the predicate of type symbol
- arguments: argument list of Literal of type symbollist.

3.4.5.2 Description

To hold a Predicate and the Argument list.

3.4.5.3 Methods

1. predicate() : Returns the predicate name of the Literal.
2. arguments(): Returns the arguments list of the Literal.
3. ChLVarName(): Changes the Variables names in the Literal's argument to a new name.
4. display_Literal(): Displays Literal's predicate and its arguments.

3.4.5.4 Friend Functions

istream& operator>>(istream&. const Literal&) : Input Operator

3.4.6 Class LITERALNODE

3.4.6.1 Data Structure

- Ltrl: Literal
- LtrlNext: Pointer to next LiteralNode

3.4.6.2 Description

To hold a Literal and a pointer to next Literal.

3.4.6.3 Methods

Friend Class: *class LiteralList* - To access private data of class LiteralList.

3.4.7 Class LITERALLIST

3.4.7.1 Data Structure

- LiteralHead: Pointer to a LiteralNode head.

3.4.7.2 Description

To maintain and manage a linked list of LiteralNodes(i.e., Literals).

3.4.7.3 Methods

1. append() : Appends a Literal to the existing LiteralList.
2. getLtrl() : Return the first LiteralNode in the LiteralList.
3. getNxtLtrl() : Returns pointer for rest of LiteralList.
4. isEmpty() : Checks whether the LiteraList is empty or not.
5 display_LiteralList(): Displays the LiteralLlist.

3.4.7.4 Friend Function :

istream& operator>>(istream&, const LiteralLlist&); : Input Operator

3.4.8 Class SYMBOLPAIR

3.4.8.1 Data Structure

- First: Holds the first symbol(symbol to be replaced) of the symbolpair.
- Second: Holds the second symbol(symbol for replacement) of the symbolpair.

3.4.8.2 Description

To maintain and manage Symbol Pairs for substitution.

3.4.8.3 Methods

1. SymbolPair() : Creates the symbol pair.

3.4.9 Class SUBSTITUTION

3.4.9.1 Data Structure

- List <SymbolPairPtr> map

3.4.9.2 Description

To maintain and manage substitution and unification aspects for the *Inference* engine.

3.4.9.3 Methods

1. operator[]() : Makes a Substitute for a symbol.
2. add() : adds a substitution symbol t for s.
3. unify() : Unifies the two Literals.
4. apply(): Applies the substitutions
5. InitSubsList(): Initializes the Substitutions List

3.4.10 Class CLAUSE

3.4.10.1 Data Structure

- head: Clause head of type Literal.
- body: Clause body of type LiteralList.

3.4.10.2 Description

To maintain and manage Clauses of the database

3.4.10.3 Methods

1. clause() : Build a clause with given head and body
2. gethead() : Returns the head(of type Literal) of the clause.
3. getbody() : Returns the body(of type LiteralList) of the clause.
4. Instance() : Returns the instantiated clause.

3.4.10.4 Friend Function :

istream& operator>>(istream&, const LiteralLlist&); : Input Operator

3.4.11 Class DATABASE

3.4.11.1 Data Structure

- array of clauses
- Number of clauses

3.4.11.2 Description

To maintain and manage the database which is an array of clauses.

3.4.11.3 Methods

1. inrange() : Checks whether the given clause index is within the range.

2. FindCl(): Returns the index to the matching clause.
3. getList() : Returns the body of a clause with given clause index.
4. operator[]() : Returns the reference to a clause for a given clause index.

3.4.11.4 Friend Function

istream& operator>>(istream&, const LiteralLlist&); : Input Operator

3.4.12 Class INFERENCE

3.4.12.1 Data Structure

- subs : List of substitutions

3.4.12.2 Description

To establish the given goal from the database rules and facts. Basically it performs the inference function.

3.4.7.3 Methods

1. Establish() : Inference Engine - Established the query on a Database.
2. Match() : Unification & Substitution

3.4.13 Class LISTNODE

The iterator pattern [8, pages 257-273] is used to access the elements of the *LIST* sequentially, regardless of the internal representation of the *LIST*. This pattern uses the LIST class and the *LIST ITERATOR* class. The *LIST ITERATOR* defines an interface to access and traverse elements of the *LIST*. While the LIST defines an interface for creating an iterator object. The main advantage of using this pattern are: the list representation can be changed without affecting the iterator since the list does not need to expose its internal structure to the iterator; different list traversals can be defined and used on the same list depending on the need.

3.4.13.1 Data Structure

- Data: any type depending on the instantiation of the list, since a list is a template.
- Next: a list node.

3.4.13.2 Description

The nodes to go in the List are of type ListNode. It is of template class.

3.4.13.3 Methods

1. getdata() : Returns the pointer to data.
2. getnext() : Returns the pointer to the next ListNode.
3. putdata() : Stores a data item in data.
4. putnext() : Appends the given ListNode at the end of the List.

3.4.14 Class LIST

3.4.14.1 Data Structure

ap : Pointer to ListNode of type template

3.4.14.2 Description

It is a container class used to store elements of any type. It could be used either with the LIFO strategy or the FIFO strategy depending on the need.

3.4.14.3 Methods

1. prepend : Inserts an element at the head of the list.

2. append : Inserts an element at the end of the list

3. IsEmpty: Checks whether the list is empty or not.

3.4.15 Class LISTITERATOR

3.4.12.1 Data Structure

- theList : List itself
- theCurrentNode: Pointer to the current ListNode.

3.4.15.2 Description

It is used to traverse the list sequentially regardless of its internal representation.

3.4.15.3 Methods

1. first: Initializes the iterator to point to the first element in the list.

2. next: Moves to the next element of the list.

4.0 Implementation of Inference Engine Mechanism

4.1 *Object Model*

Figure 8 shows all the classes involved in the inference mechanism implementation.

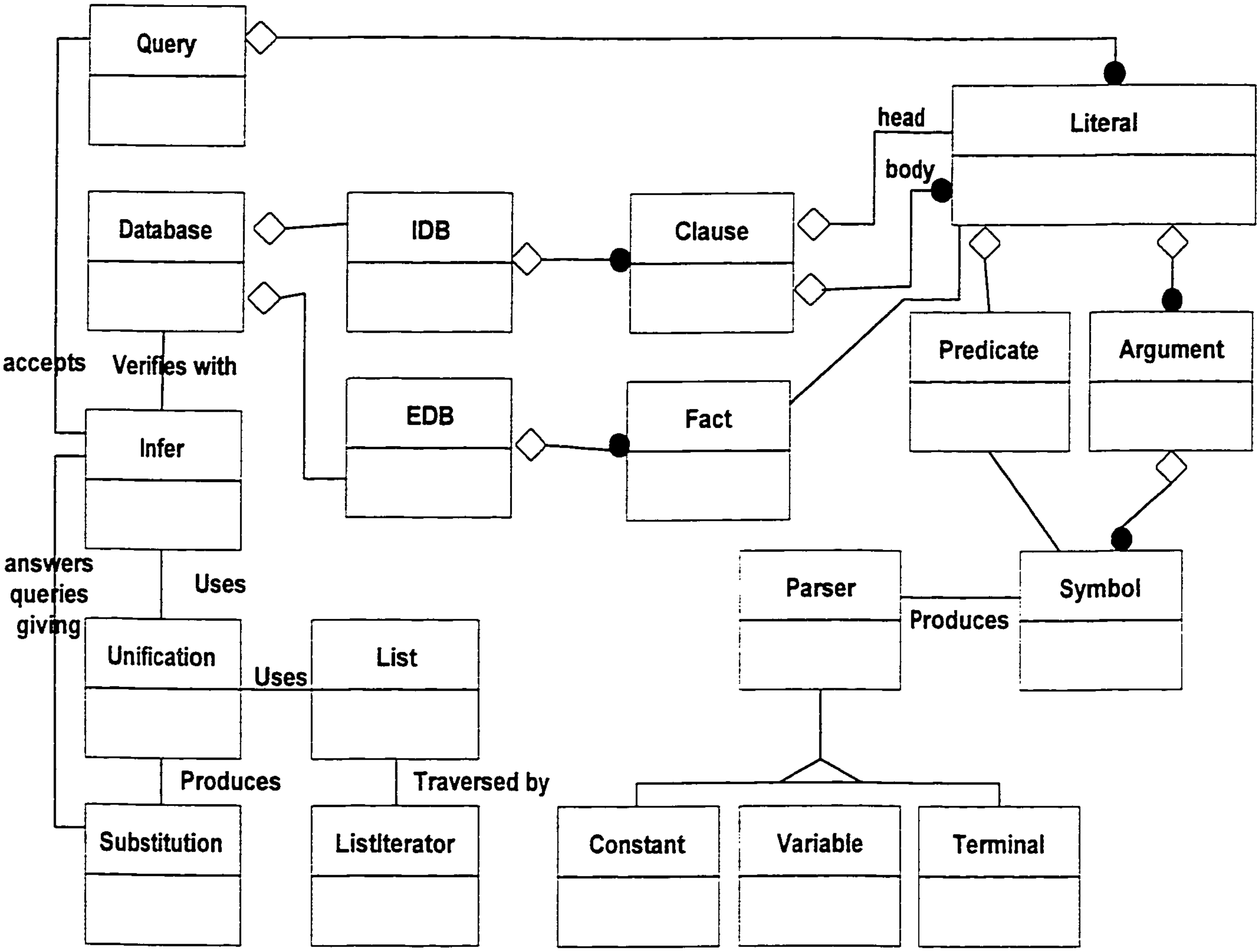

Figure 8: Detailed Object Model for Datalog Interpreter

Infer establishes the given query(goal list) on a given database and uses *Unification* algorithm which produces the substitutions necessary to *unify* two terms. The classes and associations are shown separately with their attributes and methods in ***Figure 9***.

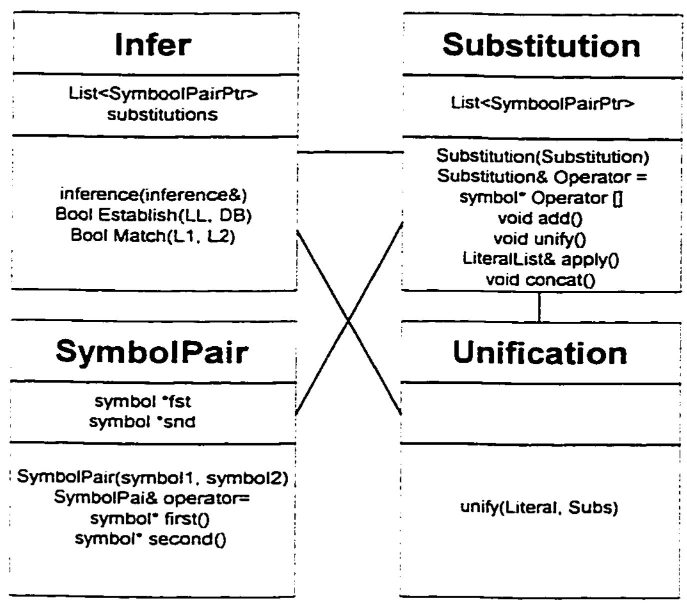

Figure 9: Inference Engine Implementaion Object Model Details

4.2 Data Structures

The ***Inference Engine*** of the Datalog interpreter processes the query on a given database. The key to this algorithm is to delay the actual choice of constants for variables in Datalog rules as long as possible. The *Datalog Inference* engine uses the *Unification algorithm* to unify the literals; this is the heart of the Inference Engine. The following paragraphs present the pseudo-code for the *Unification algorithm*[13] in ***Figure 10*** and the *Inference Engine Algorithm*[13] in ***Figure 11***.

4.2.1 Unification Algorithm

The purpose of this algorithm is to check whether the two given literals are unifiable. This algorithm takes two literals, ***Literal1*** and ***Literal2*** as input.

```
Function: Unify()
begin

If the predicate of Literal1 is not same as the predicate of Literal2
        return that there is no unification

Repeat
        If we have a variable term in Literal1
        begin
                If the two variable terms of Literal1 and Literal2 are same
                   do nothing
                else
                   begin
                        Add the replacement element pair (variable term of Literal1,
                                variable term of Literal2) to the Substitution List
                        Apply the above substitution to all the terms in the Literal1
                        Apply the above substitution to all the terms in the Literal2
                        Apply the above substitution to all the right sides of the existing
                        Substitution List
                   end
        end
        Else
           begin
                If we have a variable term in Literal2
                begin
                     Add the replacement element pair (variable term of Literal2,
                                variable term of Literal2) to the Substitution List
                     Apply the above substitution to all the terms in the Literal1
                     Apply the above substitution to all the terms in the Literal2
                     Apply the above substitution to all the right sides of the existing
                     Substitution List
                end
              end
        Else
               If the terms of Literal1 and Literal2 are not same
                     return that there is no unification

        go to the next terms of Literal1 and Literal2

Until all the terms are traversed through
Stop with success.
End
```

Figure 10: Unification Algorithm

The substitution list contains the list of individual substitutions which unify the two given literals.

4.2.2 Unification Example

Consider the following *two* Literals :

e(a, XX, b, ZZ), e(a, YY, YY, WW)

the first arguments match, so no replacements are generated for them. Comparing second arguments generates the substitution {*XX* = *YY*} and modifies the literals to: *e(a, YY, b. ZZ), e(a, YY, YY, WW)*. Comparing *b* and *YY* makes the substitution {*XX* = *b*, *YY* = *b*} and the literals *e(a, b, b, ZZ). e(a, b, b, WW)*. Matching the last arguments gives {*XX* = *b*, *YY*=*b*, *ZZ*=*WW*}as the value returned for the substitution. Both literals are now *e(a, b, WW)* under the substitution.

4.2.3 Inference Algorithm

The purpose of this algorithm is to establish the given goal on a database which contains a set of clauses. The algorithm takes a list of goals and tries to infer the given goal from the given database.

In our implementation, the *Establish()* function (Inference Engine) uses a data structure ***subst***, four functions (*predsym(), instance(), unify* and *apply()*), and a new version of the *concatenation* procedure. It has literals in the goals list. A value of type *subst* represents a substitution, which is a set of pairs of variables and constants. Each pair is called a *replacement*. A substitution says which constants should be substituted for which variables. Function *predsym()* just extracts the predicate symbol from a literal. Function

```
Function : Establish()
begin

        If the goal-list is empty
                Stop with success

        Get the first Literal of the goal-list
        Get the Predicate term from the Literal
        Search the database (start from the first clause) for a clause whose head has the
                    matching predicate that of goal Literal
        If the search is unsuccessful, that is, we do not have a clause with matching
                    goal predicate in its head, Stop with Failure

Repeat
        Make the Instance of the above clause, that is, copy the clause from the
                database into a new clause and change all the variables in it to new
                variables names which were not used before
        Try to unify the Literals from the instantiated clause head and the goal-
                literal using the above unification algorithm
        If the Literals can be unified
        begin
            replace the goal-literal with the body of the instantiated clause thus
                arriving at new goal-list
            Apply the replacements(contained in Substitution List ) returned by
                Unification algorithm to the entire goal-list
            Invoke the Inference algorithm(Establish) again with this new
                goal-list
            If the goal-list is established
                Stop with success
        end

        Get the first Literal of the goal-list
        Get the Predicate term from the Literal
        Search the database(start from the last matching clause seen + 1)
                for a clause whose head has the matching predicate that of
                goal Literal
        If the search is unsuccessful, that is, we do not have a clause with
                matching goal-predicate in its head, Stop with Failure

Until no more matching clauses are found
Stop with success
end
```

Figure 11: Inference Engine Algorithm

instance() takes a rule and uniformly changes all the variables in it to new variables that have not been used so far. It makes a copy of the rule, rather than modifying the rule itself. Algorithm unify takes the head of a rule and a goal literal and does a comparison to determine what replacements are needed to make them match. If they cannot be made to match, the function returns false. If they can match, the function returns true, and it also returns a substitution that contains replacements to make them match. Function *apply()* takes a substitution and a goal list and makes all the appropriate replacements. It is like the "replace all" function of an editor.

The concatenation procedure, *copycat2()* (page-5, Appendix-A), makes a copy of its second argument, so that *apply()* (page-5, Appendix-A), does not alter the current goal list when forming he next one. The unaltered goal list is needed for *backtracking* in case the recursive call with the new goal list fails. Since *instance()* (page-5, Appendix-A) copies its argument, *copycat2()* need not copy its first argument.

Figure 11 presents pseudo-code for the inference algorithm, ***establish()***. Given a goal list with variables, the current version of ***establish()*** will leave them as variables until a value for each is determined.

5.0 Extensions of Pure Datalog

The *Datalog* syntax we have been considering so far corresponds to a very *restricted* subset of first-order logic and is often referred to as ***pure Datalog***[15]. Several *extensions* of pure Datalog have been proposed in the literature. The most important of these extensions are *built-in predicates*, *negation*, and *complex objects*. In our project, these extensions are not looked into.

6.0 CONCLUSION

The *main attraction* of *Datalog* is the possibility of dealing, within the one *formalism*, with non-recursive expressions (or views) as well as with recursive ones. Although this area is still very active, we feel that some basic understanding has been established, thus allowing for systematic treatment. One of the major challenges that Datalog research has still to meet is to convince the knowledge base community of the practical merits of this theory. The weaknesses of Datalog work have been indicated as follows.

a) Very few applications have been shown which can take full advantage of *Datalog's expressive power*. In particular, no useful applications have been reported so far for nonlinear or mutually recursive rules.

b) *Datalog* is not considered as a programming language, but rather as a "*pure*" computational paradigm. For instance, Datalog *does not* provide support for writing user's interfaces, and does not support quite useful programming tools, such as modularization and structured types.

c) *Datalog* does not compromise its clean declarative style in any way; while sometimes it is required that the programmer may take control on inference processing, by stating the order and method of execution of rules. This is typical, for instance, of many expert system shells.

d) *Datalog* systems have been considered, until now, as closed worlds, that do not talk to other systems; while the current trend is towards supporting heterogeneous systems.

Some of the above *criticisms* are in fact well founded; and provide an indication of the directions in which we expect *Datalog* to move in order to become fully applicable. Datalog research will have to consider the advances in other research areas; in particular, Datalog can be *extended* to support complex terms; this is a first step towards the development of *new language paradigms* which use some of the concepts from object-oriented databases. In summary, we expect that supporting rule computation will be one of the ingredients of future knowledge base systems; Datalog research has *provided* exact methods and fairly good understanding for approaching this issue.

In this project, a top down version of a Datalog interpreter was designed in the Object Oriented paradigm making use of design patterns, and then implemented in C++. Some important topics had to be studied first in order to gather the necessary background. The second step, was the design and implementation of the Datalog interpreter. The design uses design patterns which makes it reusable in other programs or applications. The third step, was the understanding of the unification concept and its design and implementation. The unification module is at the heart of the inference mechanism. The final step was the building of inference mechanism, which provides a decidable way to answer any query based on the facts and rules stored in the database. We are confident about the correctness of this mechanism, we have tested it with the examples from a text book[13] and some examples prepared by the author and his supervisor. The results of the various tests are enclosed in *Appendix-B - Experimental Results*.

The importance of this work lies in the use of *Object Oriented Paradigm* in the design and implementation phases, which should make it easier to modify, extend, and reuse in different applications. Although the implementation is robust, it has one major limitation: It is a non-recursive one, that is, it can not operate well, if a predicate symbol in the body of a rule also appears as the predicate symbol in the head. However, this drawback can be eliminated by adopting an advanced inference algorithm given in chapter 6 of [13].

BIBLIOGRAPHY

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D.Ullman, *Compilers - Principles, Techniques, and Tools*, Addison Wesley Publishing Company, March 1988.

[2] G. Butler, *Datalog and TwoGroups and C++*, Integrating Symbolic Mathematical Computation and Artificial Intelligence, Jacques Calmet and John A. Campbell(eds), Lecture Notes in Computer Science 958, Springer-Verlag, Berlin, 1995, pp. 80-92.

[3] G. Butler, S.S. Iyer and E.A. O'Brien, *A Database of Groups of Prime-Power Order, Software - Practice and Experience 24*, 10 (October 1994) 911-951.

[4] Micheal A. Carrico, John E. Girard and Jennifer P. Jones, *Building Knowledge Systems*, McGraw Hill Book Company, 1989.

[5] Stefano Ceri, Georg Gottlob and Letizia Tanca, *What You Always Wanted to know About Datalog(And Never Dared to Ask)*, IEEE Transaction on Knowledge and Data Engineering, Vol 1. No. 1, March 1989, pp. 146-166.

[6] Ramez Elmasri, Shamkant B. Navatha, *Fundamentals of Database Systems*, Benjamins/Cummings Inc., Redwood City, California, 1989.

[7] Michael J. Folk and Bill Zoellick, *File Structures - A Conceptual Toolkit*", Addison Wesley Publishing Company, 1989.

[8] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Publishing Company, 1995.

[9] Peter M.D. Gray and Robert J. Lucas, *Prolog and Databases - Implementations and New Directions*, Johan Wiley & Sons, Chichester, 1988.

[10] Frank Van Harmelen, Peter Jackson and Han Reichgelt, *Logic-Based Knowledge Representation*, The MIT Press, 1989.

[11] Christopher John Hogger, *Essentials of Logic Programming*, Clarendon Press, Oxford, 1990.

[12] Stanley B. Lippman, *C++ Primer*, Addison Wesley Publishing Company, 1991.

[13] David Maier and David S. Warren, *Computing with Logic - Logic Programming with Prolog*, The Benjamin/Cummings Publishing Company Inc., 1988.

[14] Scott Meyers, *Effective C++*, Addison Wesley Publishing Company, 1992.

[15] Ulf Nilsson and Jan Maluszynski, *Logic, Programming and Prolog*, John Wiley & Sons , 1990.

[16] Wolfgang Pree, *Design Patterns for Object-Oriented Software Development*, Addison Wesley Publishing Company, 1995.

[17] J. Rumbaugh, M. Blaha, W.Premerlani, F.Eddy and W.Lorenson, *Object-Oriented Modelling and Design*, Prentice Hall, New Jersey, 1991.

[18] Bjarne Stroustrup, *Programming with C++*, 2nd edition, Addison Wesley Publishing Company, 1991.

[19] J.D. Ullman, *Principles of Databases and Knowledge-Base Systems, Volume I*, Computer Science Press, Inc, Rockville, Madison, 1988.

[20] J.D. Ullman, *Principles of Databases and Knowledge-Base Systems, Volume II*, Computer Science Press, Inc, Rockville, Madison, 1989.

[21] L.Vieille, *Recursive axioms in deductive databases: The Query Subquery approach*, in 8th ACM Symposium, Principles of Database Systems(PODS), March 1989, pp. 1-10.

[22] L.Vieille, *A database complete proof procedure based on SLD resolution*, ECRC, Munich, West Germany, Int. Rep. IR-KB-40, November 1986.

APPENDIX - A

Program Listings

```
/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       Clause.C                                 */
/*      Description     :       To create and manage the Clauses         */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#include "Lex.h"
#include "Syn.h"
#include "Symbol.h"
#include "Clause.h"
#include "Global.h"

clause::clause()                    // clause constructor
:head(),body()
{

}

clause::~clause()                   // clause destructor
{

}

clause::clause(const Literal& h, LiteralList& b)  // constructor
:head(h),body(b)
{

}

clause::clause(const clause& c)          // copy constructor
:head(c.head), body(c.body)
{

}

// **** To get the head of the Clause  ****

Literal& clause::gethead()               // To get the head of the clause
{
   return(head);
}

clause& clause::operator=(const clause& cl)
{
        if( this != &cl)
        {
          head = cl.head;
          body = cl.body;
        }
        return *this;
}

LiteralList& clause::getbody()           // To get the body of the clause
{
   return(body);
}
```

```
istream& operator>>(istream& inFile, clause& cl)   //input operator
{
        Literal h;
        LiteralList Ll;

        curr_tok = prev_tok = INI;
        get_token(inFile);
        parse(inFile);
        if (curr_tok == END || curr_tok == ERROR) return inFile;

        if(curr_tok == NAME) curr_tok = PREDICATE;  // FORCE to PREDICATE

        if (curr_tok == PREDICATE)                 // To confirm the head first
             inFile >> h;               // Reading the head
        else return inFile;

        get_token(inFile);              //checking for iff
        parse(inFile);

        if (curr_tok == ERROR) return inFile;

        if (curr_tok == iff_ok)
        {
          //cout << "LiteralList invoked" << endl;
          curr_tok = prev_tok = INI;
          inFile >> Ll;                 // Reading the body
        }

        cl = *new clause(h,Ll);         //construct new clause
        return inFile;
}

void clause::display_clause() const
{
        cout << "Clause: " ;
        head.display_Literal();
        cout << " :- ";
        body.display_LiteralList();
        cout << "." << endl;
}

clause& clause::Instance()
{
  cout << endl << "Clause - BEFORE Instantiation" << endl;
  head.display_Literal();
  cout << " :- ";
  body.display_LiteralList();
  cout << "." << endl;

  ++seqno;
  head.ChLVarName(seqno);
  body.ChLLVarName(seqno);

  cout << endl << "Clause - AFTER Instantiation" << endl;
  head.display_Literal();
  cout << " :- ";
  body.display_LiteralList();
  cout << "." << endl;

return(*this);
```

```
}

/***************************************************************************/
/*                                                                         */
/*                                                                         */
/*      Module          :       Database.C                                 */
/*      Description     :       To create and manage the Database          */
/*                                                                         */
/*                                                                         */
/***************************************************************************/

#include <fstream.h>
#include "Lex.h"
#include "Literal.h"
#include "LiteralList.h"
#include "Database.h"

database::database(char *infile)                //constructor
{
  ifstream inFile(infile, ios::in);

  if (! inFile )                               // File open failed
     {
        cout << "*** Sorry! can not open " << infile << " for input" << endl;
        curr_tok = ERROR;
     }
  else
     {
        //cout << "Input Data File: " << infile << endl ;
        inFile >> *this;
     }
}

database::~database()                           //destructor
{

}

Boolean database::inrange(int i) const          //range check
{
  return( i < no_of_clauses);
}

int database::FindCl(int i, const symbol& pred) //Predicate existence check
{
    Literal L;
    symbol s;

    cout << endl << "Finding Clause starting with i= " << i << " ->";

    for(; i<no_of_clauses;i++)
    {
       L = clausearray[i].gethead();
       s = L.predicate();
       if( s == pred)
          break;
    }

    if ( i < no_of_clauses)
        cout <<  "Clause Found at i= " << i << endl;
```

```
    else
        cout <<  "Clause Not found" << endl;

    return i;
}

istream& operator>>(istream& inFile, database& dbs)
{
  int i = 0;
  while ((!inFile.eof()) && (curr_tok != END) && (curr_tok != ERROR))
  {
       curr_tok = prev_tok = INI;
       inFile >> dbs[i++];
  }

  dbs.no_of_clauses = --i;
  cout << "Number of clauses: " << dbs.no_of_clauses << endl;
  return inFile;
}

clause& database::operator[](int i)       //to get clause reference
{
        return (clausearray[i]);
}

database::database()
{

}

/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       Error.C                                  */
/*      Description     :       Generates errors during input files parsing */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#include <iostream.h>
#include "Error.h"

int no_of_errors;

void error(const char* s)
{
     cerr << "Error : " << s << "\n";
     no_of_errors++;
}

/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       Infer.C                                  */
/*      Description     :       To create and manage the Clauses         */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#include "Bool.h"
```

```
#include "Infer.h"
#include "Substitution.h"

inference::inference()                      // constructor
{
}

inference::~inference()                     // destructor
{

}

Boolean inference::Establish(LiteralList& goals, database& db)
{
    clause       NxtCl;
    clause       ClInst;
    LiteralList  L1;
    symbol       Pred;
    int ClPos    = 0;
    Boolean      terminate = FALSE;

    if (goals.isEmpty()) return TRUE;

    cout << endl << "Goals : ";
    goals.display_LiteralList();

    Pred  = ( ((goals.getLNode())->getLtrl())->predicate() );

    cout << endl << "Looking for Clause with Predicate: "<< Pred ;
    ClPos = db.FindCl(0, Pred);

    if (db.inrange(ClPos))
    {
       NxtCl = db[ClPos];
       NxtCl.display_clause();              //Display this clause
    }
    else
       terminate = TRUE;


    while(terminate != TRUE)
    {
        ClInst = NxtCl.Instance();

        cout << endl << "Trying to Establish the goals : ";
        goals.display_LiteralList();

        LiteralList NewGoals(goals);  // Preserve the Goals for BackTracking

        if (Match(ClInst.gethead(), ((goals.getLNode())->getLtrl())) )
        {
           L1 = goals.rest();
           if (Establish( Subs.apply((ClInst.getbody()).copycat2(L1)), db) )
              return(TRUE);
        }

        // Unable to Establish the present goals. So backtrack to
        // preserved goals list and try for alternate choices

        Subs.InitSubsList();                //Initialize the Subs List
```

```
        goals = NewGoals;               // Get back the preserved goals list

        cout <<endl<<"*** Unable to Establish the goals - Back Tracking to -> ";
        cout << endl << "Goals : ";
        goals.display_LiteralList();

        Pred  = ( ((goals.getLNode())->getLtrl())->predicate() );
        ClPos = db.FindCl(++ClPos, Pred);

        if (db.inrange(ClPos))
            NxtCl = db[ClPos];
        else
            terminate = TRUE;
    }

return FALSE;
}


Boolean inference::Match( Literal& Head,  Literal* Goal)
{
   node   *SlH, *SlG, *SlHOrg, *SlGOrg;

   cout << endl << "Trying to Unify the Literals  : ";

   Head.display_Literal();
   cout << " and ";
   Goal->display_Literal();

   if(Head.predicate() != Goal->predicate())
      return(FALSE);                             //Wrong Rule - Return False

   /* Same Predicates - So proceed */
   symbollist* s = Head.arguments();
   SlH = SlHOrg  = s->gethead();

   s   = Goal->arguments();
   SlG = SlGOrg = s->gethead();

   while(SlH && SlG)
   {
       if ( ((SlH->getinfo()).getSKind()) == VAR)
       {
         if ( SlH->getinfo() == SlG->getinfo())  // Same variable ?
            {
              // YES - Same variables - Do nothing !
              continue;
            }
          else
            {
              // NO - Not the same variables - Try to unify
              Subs.add(SlH->getinfo(), SlG->getinfo());
              Subs.unify(SlHOrg, SlGOrg);
            }
       }
       else
       {
          if ( ((SlG->getinfo()).getSKind()) == VAR)
             {
               Subs.add(SlG->getinfo(), SlH->getinfo());
```

```
                Subs.unify(S1HOrg, S1GOrg);
              }
          else
              {
                if ( S1H->getinfo() != S1G->getinfo() ) //Both same constants ?
                    {
                     cout << endl << "Unification Failed : Constants not same"<<en
dl;
                     return FALSE;
                    }
              }
        }
        S1H = S1H->getnext();
        S1G = S1G->getnext();
    }

cout << endl << "OK - Unification is Sucessfull "<< endl;

return TRUE;
}
/******************************************    *******************************/
/*                                                                            */
/*                                                                            */
/*      Module          :       Lex.C                                         */
/*      Description     :       To perform Lexical Analyis on the given       */
/*                              input ascii file for clauses and query        */
/*                                                                            */
/*                                                                            */
/******************************************************************************/

// Input and Lexical analysis definitions

#include <stdio.h>
#include "Error.h"
#include "Lex.h"
token_value curr_tok;
token_value prev_tok;

token_value get_token(istream& inFile)
{
        char ch;
        do
        {
          if (!(inFile.get(ch))) return curr_tok = END;
        } while (isspace(ch));

        switch (ch)
        {
                case ':' :
                case ',' :
                case '.' :
                case '-' :
                case '(' :
                case ')' :

                        return curr_tok = token_value(ch);

                default :

                        if (isalnum(ch))
```

```
                              {
                                if(isupper(ch)) curr_tok = VARIABLE;
                                else            curr_tok = NAME;

                                // NAME can be PREDICATE or CONSTANT
                                inFile.putback(ch);
                                return curr_tok;
                              }
                              else
                              {
                                error("bad token");
                                return curr_tok = ERROR;
                              }
                }
}
/****************************************************************************/
/*                                                                          */
/*                                                                          */
/*      Module          :       Literal.C                                   */
/*      Description     :       To create and manage the Literal Class      */
/*                                                                          */
/*                                                                          */
/****************************************************************************/


#include "Literal.h"

Literal::Literal( symbol& pn, symbollist& SL )
: pred( pn ), arglist( SL )
{
}

Literal::Literal( const Literal& L )
: pred( L.pred ), arglist( L.arglist )
{
}

Literal::Literal()
{

}

Literal::~Literal()
{
}

Literal& Literal::operator = ( const Literal& L )
{
        if( this != &L )
        {
          pred = L.pred;
          arglist = L.arglist;
        }
        return *this;
}

Boolean Literal::operator == ( const Literal& L )
{
    if ( (L.pred == pred) && (((Literal )L).arglist == arglist) )
           return TRUE;
```

```
     else
          return FALSE;
}


symbol& Literal::predicate()
{
        return pred;
}

symbollist* Literal::arguments()
{
        return (&arglist);
}

istream& operator>>(istream& inFile, Literal& L)  //input operator
{
        symbol     predsym;
        symbollist sl;

        if (curr_tok == PREDICATE)             // To confirm the head first
        {
                prev_tok = curr_tok;
                inFile >> predsym;             // Building the predicate
        }
        else    return inFile;

        get_token(inFile);                     // Checking for LP
        parse(inFile);
        if (curr_tok == END || curr_tok == ERROR ||
            curr_tok != LP) return inFile;

        if(curr_tok == END  || curr_tok == ERROR) return inFile;

        inFile >> sl;                         // Building the arglist
        L = *new Literal(predsym,sl);         // Constructing the Literal

        return inFile;
}

void Literal::display_Literal() const
{
  cout << pred ;
  arglist.display_symbollist();
}

void Literal::ChLVarName(int seqno)
{
    arglist.ChSlVarName(seqno);

}
/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       LiteralList.C                            */
/*      Description     :       To create and manage the LiteralList Class  */
/*                                                                       */
/*                                                                       */
/*************************************************************************/
```

```
#include "LiteralList.h"

LiteralNode::LiteralNode(const Literal& L, LiteralNode *n)
{
   Ltrl = new Literal(L);
   LtrlNext = n;
}

LiteralNode::~LiteralNode()
{

}

LiteralList::LiteralList()
{
    Literalhead = NULL;
}

LiteralList::~LiteralList()
{

}

LiteralList::LiteralList(LiteralNode* n)
{
   Literalhead = n;
}

LiteralList::LiteralList(const LiteralList& LL)
{
   Literalhead = NULL;
   LiteralNode *last = NULL;
   LiteralNode *cursor = LL.Literalhead;
   if (cursor != NULL)
   {
       Literalhead = new LiteralNode(*(cursor->Ltrl), NULL);
       last = Literalhead;
       cursor = cursor->LtrlNext;
       while(cursor != NULL)
       {
       last->LtrlNext =  new LiteralNode(*(cursor->Ltrl), NULL);
       cursor = cursor->LtrlNext;
       last   = last->LtrlNext;
       }
     }
}

void LiteralList::append(const Literal& n)
{
   LiteralNode *cursor = Literalhead;
   if (cursor != NULL)
   {
       while(cursor->LtrlNext != NULL)
             cursor = cursor->LtrlNext;
       cursor->LtrlNext =  new LiteralNode(n, NULL);
   }
   else
       Literalhead =  new LiteralNode(n, NULL);
}
```

```
LiteralNode* LiteralList::getLNode()
{
   return (Literalhead);
}

LiteralList LiteralList::rest()
{
   if (Literalhead == NULL)
       return LiteralList();
   else
       return LiteralList(Literalhead->LtrlNext);
}

LiteralList& LiteralList::operator = (const LiteralList& LL)
{
   Literalhead = LL.Literalhead;
   return *this;
}

LiteralList& LiteralList::copycat2(LiteralList& L)
{
   if (isEmpty()) return(L);

   // The LiteralLists are not Null, So append the two lists
   LiteralNode *cursor = Literalhead;
   while(cursor->LtrlNext != NULL)
         cursor = cursor->LtrlNext;
   cursor->LtrlNext =  L.Literalhead;

return *this;
}


istream& operator>>(istream& inFile, LiteralList& LL)    //input operator
{
   Literal L;

   for( ; ; )
   {
     get_token(inFile);
     parse(inFile);
     if (curr_tok == END || curr_tok == DOT) break;
     if (curr_tok == NAME) curr_tok = PREDICATE;

     switch(curr_tok)
     {
        case PREDICATE :
             inFile >> L;          // Build the Literal
             LL.append(L);         // Append the new Literal to LiteralList
             break;

        case COMMA     :
             break;

        case ERROR     :
        default        :
           error ("Out of Sequence");
           curr_tok = prev_tok = ERROR;
           break;
      }
```

```
    }
    return inFile;
}

void LiteralList::display_LiteralList() const
{
  Literal Lit;

  LiteralNode *cursor = Literalhead;

  if(isEmpty()) cout<<"( No Body )";
  else
    {
      while(cursor != NULL)
      {
        (cursor->Ltrl)->display_Literal();
        cursor = cursor->LtrlNext;
        if(cursor != NULL) cout<<" , ";
      }
    }
}

void LiteralList::ChLLVarName(int seqno)
{
   LiteralNode *cursor = Literalhead;

   while (cursor != NULL)
   {
     (cursor->Ltrl)->ChLVarName(seqno);
      cursor = cursor->LtrlNext;
   }
}



/*****************************************************************************/
/*                                                                           */
/*                                                                           */
/*      Module          :       Main.C                                       */
/*      Description     :       Datalog Interpreter Program                  */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/

#include "Database.h"
#include "Infer.h"
#include "Query.h"

Boolean main( int argc, char **argv )
{
  if (argc > 3) {
     error("arguments error");
     return FALSE;}

  prev_tok = curr_tok=INI;
  cout << "*** Proceeding for Database creation ****" << endl;
  cout << "Input Clauses File: " << argv[1];

  database mydata(argv[1]);
```

```
  if (curr_tok != ERROR)
  {
    cout << endl << "Clause Errors = " << no_of_errors;
    cout << endl << "Database is created Successfully" << endl;
  }
  else
    return FALSE;

  inference my_infer;

  cout << endl << "*** Proceeding to build the Query ****" << endl;
  no_of_errors = 0;
  cout << "Input Query File: " << argv[2] ;
  cout << endl << "Query Errors = " << no_of_errors;
  cout << endl << "User's Query : " ;
  query qry(argv[2]);

  if(curr_tok == ERROR) return FALSE;
  cout << endl << endl;

  cout <<"**** Datalog Interpreter is Trying to Estblish User's Query ****";
  cout << endl;

  if (my_infer.Establish(qry.getgoals(), mydata) )
     cout<<endl<<"**** SUCCESS - User's Query CAN BE Established ****"<< endl;
  else
     cout<<endl<<"**** FAILURE - User's Query CAN'T BE Established ****"<< endl;
}
/****************************************************************************/
/*                                                                          */
/*                                                                          */
/*      Module           :      Query.C                                     */
/*      Description      :      To create and manage the Queries            */
/*                                                                          */
/*                                                                          */
/****************************************************************************/

#include <fstream.h>
#include "Bool.h"
#include "Lex.h"
#include "Query.h"

query::query()                          //constructor
:goals()
{

}

query::~query()                         //destructor
{

}

LiteralList& query:: getgoals()         //To get the goals pointer
{
   return goals;
}

query::query(const query& s)            //copy constructor
{
```

```
   goals = s.goals;
}

Boolean query::checkdot()
{
        char ch;
        cin.get(ch);
        if (ch == '.') return TRUE;
        else
           {
           cin.putback(ch);
           return FALSE;
           }
}

istream& operator>>(istream& inFile, query& s) //input operator
{
  LiteralList ll;
  prev_tok = iff_ok;         //to pretend as if head already been read
  curr_tok = INI;

  while ((!inFile.eof()) && (curr_tok != END) && (curr_tok != ERROR))
        {
           prev_tok = iff_ok;
           inFile >> ll;
        }

  ll.display_LiteralList();
  s.goals = ll;

  return inFile;
}

query::query(char *infile)                //constructor
{
  ifstream inFile(infile, ios::in);

  if (! inFile )                         // File open failed
     {
        cout << "*** Sorry! can not open " << infile << " for input" << endl;
        curr_tok = ERROR;
     }
  else
     {
        //cout << "Input Data File: " << infile << endl ;
        inFile >> *this;
     }
}




/***************************************************************************/
/*                                                                         */
/*                                                                         */
/*      Module          :       Substitution.C                             */
/*      Description     :       To create and manage the Substitution Class */
/*                              and its methods                            */
```

```
/*                                                                              */
/*                                                                              */
/******************************************************************************/

// A substitution is a map from Symbols to Symbols in Datalog

#include "Substitution.h"

Substitution::Substitution()
: map()
{
}

Substitution::Substitution( const Substitution& S )
: map( S.map )
{
}

Substitution::~Substitution()
{
}

Substitution& Substitution::operator =( Substitution& S )
{
        map =  S.map;
        return *this;
}

symbol* Substitution::operator[]( const symbol& s )
{
        ListIterator< SymbolPairPtr >   iter(map);
        symbol*                         symptr = NULL;

        for( SymbolPairPtr *sp = iter.first(); sp != NULL; sp = iter.next() )
             if( *((*sp)->first()) == s )
                 {
                    symptr = (*sp)->second();
                    break;
                 }
        return symptr;
}

LiteralList& Substitution::apply(LiteralList& Ll)
{
   LiteralNode *cursor = Ll.Literalhead;
   while (cursor != NULL)
   {
       this->unify( (cursor->getLtrl())->arguments()->gethead(), NULL );
       cursor = cursor->getLtrlNext();
   }
return(Ll);
}

void Substitution::unify(node* SlG, node* SlH)
{
  symbol  *symptr;
  Substitution *ptr = this;

  while(SlG)
  {
```

```
        symptr = NULL;
        if ( ((S1G->getinfo()).getSKind()) == VAR)
               symptr = ((*ptr)[(const symbol&)(S1G->getinfo())]);
        if (symptr != NULL)
          *(symbol*)((S1G)->getinfo1()) = *symptr;

        S1G = S1G->getnext();
   }

   while(S1H)
   {
        symptr = NULL;
        if ( ((S1H->getinfo()).getSKind()) == VAR)
               symptr = ((*ptr)[(const symbol&)(S1H->getinfo())]);
        if (symptr != NULL)
          *(symbol*)(S1H->getinfo1()) = *symptr;

        S1H = S1H->getnext();
     }
 return;
 }

void Substitution::InitSubsList()
{
        ListIterator< SymbolPairPtr >   iter(map);

        for( SymbolPairPtr* sp = iter.first(); sp != NULL; sp = iter.next() )
             *((*sp)->first())  = *((*sp)->second()) = 0;
}

void Substitution::add( const symbol& s, const symbol& t )
{
        ListIterator< SymbolPairPtr >   iter(map);

        for( SymbolPairPtr* sp = iter.first(); sp != NULL; sp = iter.next() )
           if( *((*sp)->second()) == s)
             *((*sp)->first()) = s;

        map.append( new SymbolPair( s, t ) );
}

void Substitution::concat( Substitution& s )
{
        ListIterator< SymbolPairPtr >   iter( s.map );

        for( SymbolPairPtr* p = iter.first(); p != NULL; p = iter.next() )
        {
           map.append( new SymbolPair( *((*p)->first()), *((*p)->second()) ) );
        }
}


SymbolPair::SymbolPair()
: fst( NULL ), snd( NULL )
{
}

SymbolPair::SymbolPair( const symbol& s, const symbol& t )
{
        fst = new symbol( s );
```

```
        snd = new symbol( t );
}

SymbolPair::~SymbolPair()
{
        delete fst;
        delete snd;
}

SymbolPair& SymbolPair::operator =( const SymbolPair& SP )
{
        if( this != &SP )
        {
                fst = SP.fst;
                snd = SP.snd;
        }
        return *this;
}


symbol* SymbolPair::first()
{
        return fst;
}

symbol* SymbolPair::second()
{
        return snd;
}


/***************************************************************************/
/*                                                                         */
/*                                                                         */
/*      Module          :       Symbol.C                                   */
/*      Description     :       To create and manage Symbol Class          */
/*                                                                         */
/*                                                                         */
/***************************************************************************/

#include <string.h>
#include <stdio.h>
#include "Symbol.h"
//#include "Global.h"

symbol::symbol(char* s , const SymKind& sk)             //constructor
{
  if (s != 0)
  {
        len = strlen(s) + 5;
        Sname=new char[len + 5 + 1];
        strcpy(Sname,s);
  }
  else
  {
        len = 0;
        Sname = new char[1];
        Sname[0] = '\0';
  }
  Skind = sk;
```

```
}

symbol::~symbol()                                          //destructor
{
}

symbol::symbol(char* s)                                    //constructor
{

  if (s != 0)
  {
        len = strlen(s) + 5;
        Sname=new char[len + 5 + 1];
        strcpy(Sname,s);
  }
  else
  {
        len = 0;
        Sname = new char[1];
        Sname[0] = '\0';
  }
  Skind = PRED;
}


symbol::symbol(const symbol& s)                    //copy constructor
{
        if (s == 0)
        {
           len = 0;
           Sname = new char[1];
           Sname[0] = '\0';
           Skind = PRED;
        }
        else
        {
           if ( s.Sname == 0)
           {
             len = 0;
             Sname = new char[1];
             Sname[0] = '\0';
             Skind = PRED;
           }
           else
           {
           len = s.len;
           Sname = new char[ len+1 ];
           strcpy (Sname, s.Sname);
           Skind = s.Skind;
           }
        }
}

symbol& symbol::operator=(const symbol& s)                //Assignment operator
{
        if ( this != &s )
        {
          len = strlen(s.Sname);
          Sname = new char[len+1];
          strcpy(Sname,s.Sname);
```

```
          Skind = s.Skind;
        }
        return *this;
}

SymKind& symbol::getSKind()
{
  return(this->Skind);

}

Boolean operator==(const symbol& s1, const symbol& s2)      //eqality operator
{
   if ( ((s1.Sname != 0) && (s2.Sname != 0)) &&
        (!strcmp(s1.Sname,s2.Sname)) && (s1.Skind == s2.Skind) )
          return TRUE;
        else
          return FALSE;
}

Boolean operator!=(const symbol& s1,const symbol& s2)      //ineqality operator
{
        if (s1.Sname == 0 && s2.Sname == 0)
           return FALSE;

        if (s1.Sname == 0 || s2.Sname == 0)
           return TRUE;

        if ( (strcmp(s1.Sname,s2.Sname)) || (s1.Skind != s2.Skind) )
           return TRUE;
        else
           return FALSE;
}

ostream& operator<<(ostream&os, const symbol& s)                //Output operato
r
{
   if(s.len == 0) cout<< "Length Error" << endl;
   for(int i = 0; i < s.len; i++)
      os.put(s.Sname[i]);

   switch(s.Skind)
   {
        case PRED : break;
        case VAR  : break;
        case CONST: break;
        default   : cout<< "-> ** ERROR SKIND"<< endl; break;
   }

   return os;
}

istream& operator>>(istream&is, symbol& s)              //Input Operator
{
   char buf[20];
   is >> buf;

   s.len =   strlen (buf);
   char *charPtr = new char [s.len + 1];
   strcpy (charPtr, buf);
```

```
   s.Sname = charPtr;
   switch(curr_tok)
   {
     case PREDICATE :
          s.Skind = PRED;
          break;

     case VARIABLE  :
          s.Skind = VAR;
          break;

     case CONSTANT  :
          s.Skind = CONST;
          break;
   }

   return is;
}

void symbol::ChVarName(int seqno)
{
  char buf[25];

  if(Skind == VAR)
    {
     sprintf(buf, "%s%03d", Sname, seqno);
     len = strlen(buf);
     strcpy(Sname,buf);
    }
  return;
}
/**************************************************************************/
/*                                                                        */
/*                                                                        */
/*      Module          :       Symbollist.C                              */
/*      Description     :       To create and manage Symbollist Class     */
/*                                                                        */
/*                                                                        */
/**************************************************************************/


#include "Error.h"
#include "Syn.h"
#include "Symbollist.h"

node::node(const symbol& s, node* t)    //node constructor
{
     info = new symbol(s);
     next = t;
}

node::~node()                           //node destructor
{

}

symbollist::symbollist()
{
        head = NULL;
```

```
}

istream& operator>>(istream& inFile, symbollist& sl)
{
        symbol Symb;

        for ( ; ; )
        {
           get_token(inFile);
           parse(inFile);
           if (curr_tok == END || curr_tok == RP) break;

           if(curr_tok == NAME) curr_tok = CONSTANT;
           switch(curr_tok)
           {
                case VARIABLE  :
                case CONSTANT  :
                        inFile >> Symb;
                        sl.append(Symb);
                        break;

                case COMMA :
                        break;

                case ERROR:
                default :
                        error ("Out of Sequence");
                        curr_tok = prev_tok = ERROR;
                        break;
           }
        }
        return inFile;
}

symbollist::symbollist(node* t)
{
      head = t;
}

symbollist::symbollist(symbol& ptr)               //constructor
{
      head = new node(ptr, NULL);
}

symbollist::symbollist(const symbollist& sl)     //copy constructor
{
          head = NULL;
          node *last = NULL;
          node *cursor = sl.head;
          if (cursor != NULL)
          {
             head = new node(*(cursor->info),NULL);
             cursor = cursor->next;
             last = head;
             while(cursor != NULL)
                {
                 last->next = new node(*(cursor->info),NULL);
                 cursor = cursor->next;
                 last = last->next;
                }
```

```
        }
}

void symbollist::append(const symbol& s)  //To append node at end
{
        node *cursor=head;
        if (cursor != NULL)
        {
           while(cursor->next != NULL)
             cursor = cursor->next;

           cursor->next = new node(s, NULL);
        }
        else
           head = new node(s, NULL);
}

node* node::getnext() const             //To get the next node pointer
{
  return (next);
}

Boolean symbollist::operator==(const symbollist& SL)
{
    node *sl = SL.head;
    node *cursor = head;

    if(cursor == NULL) return(FALSE);
    while(sl != NULL && cursor != NULL)
    {
       if(cursor->info != sl->info) return(FALSE);
       sl = sl->next;   cursor = cursor->next;
    }
    return(TRUE);
}

symbollist& symbollist::operator = (const symbollist& p) // assignment operator
{
        head = p.head;
        return *this;
}

void symbollist::display_symbollist() const       // to display symbollist
{
        node *cursor=head;

        if (isEmpty())  cout << "Empty Symbollist";

        cout << "(";
        while(cursor!=NULL)
        {
           cout << *(cursor->info);
           cursor = cursor->next;
           if(cursor != NULL) cout << ",";
        }
        cout << ")";
}

symbollist::~symbollist()
{
```

```
}

symbol& node::getinfo() const
{
   return *(info);
}

symbol* node::getinfo1() const          //to get the symbol pointer
{
   return (info);
}
void symbollist::ChSlVarName(int seqno) const
{
     node *cursor=head;
     while (cursor != NULL)
      {
        (cursor->info)->ChVarName(seqno);
        cursor = cursor->next;
      }
}
/**************************************************************************/
/*                                                                        */
/*                                                                        */
/*      Module          :       Syn.C                                     */
/*      Description     :       To parse the input ascii file of clauses  */
/*                              and query                                 */
/*                                                                        */
/*                                                                        */
/**************************************************************************/

#include "Error.h"
#include "Lex.h"
#include "Syn.h"

char ch;

parse(istream& inFile)
{
        switch (curr_tok)
                {
                  case COLON :                          // iff start

                        if (prev_tok == RP)
                           {
                           inFile.get(ch);
                           if (token_value(ch) == HYPHEN)    /second chr of iff
                              return (curr_tok = prev_tok = iff_ok);
                           else
                              {
                              error ("iff error");
                              break;                    //iff error
                              }
                           }
                        else
                           {
                             error("COLON not preceded by a RP");
                             break;
                           }
```

```
    case COMMA :

          if (prev_tok == VARIABLE || prev_tok == CONSTANT ||
              prev_tok == RP || prev_tok == NAME)
              return(prev_tok = curr_tok);

          error("COMMA not preceded by a Pred/Var/Constant");
          break;

    case LP :

          if (prev_tok == PREDICATE)
               return (prev_tok = curr_tok);

           error("LP not preceded by a Predicate");
           break;

   case RP :                         //read next predicate

          if (prev_tok == VARIABLE || prev_tok == CONSTANT ||
              prev_tok == NAME)
              return(prev_tok = curr_tok);

          error("RP not preceded by a Var/Const");
          break;


    case DOT :                       //end of clause

          if (prev_tok == RP)
              return (prev_tok = curr_tok);

          error("DOT not preceded by a RP");
          break;

    case VARIABLE :

           if (prev_tok == COMMA || prev_tok == LP)
                 return (prev_tok = curr_tok);

          error ("Invalid token before the VARIABLE");
          break;

    case NAME :

           // Here NAME can be a PREDICATE or a CONSTANT.
           // So check for it !!!

           if (prev_tok == COMMA  || prev_tok == INI ||
               prev_tok == iff_ok || prev_tok == LP)
               return(prev_tok = curr_tok);

           error ("Invalid token before a NAME");
           break;

    case END :

          return (prev_tok = curr_tok);

    case ERROR  :
```

```
                default     :
                     cout << curr_tok << endl;
                     error ("Invalid token ");
                     break;
        }
        return (curr_tok = prev_tok = ERROR);
}
```

```
/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       Bool.h                                   */
/*      Description     :       Boolean definitions                      */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#ifndef BOOL_H
#define BOOL_H

#define Boolean int
#define TRUE 1
#define FALSE 0

#endif
/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       Clause.h                                 */
/*      Description     :       Definitions for Clause Class and its methods*/
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#ifndef CLAUSE_H
#define CLAUSE_H

#include <iostream.h>
#include "Literal.h"
#include "LiteralList.h"

// ***  Class - Database with Clauses   ***

class clause
{
   Literal     head;
   LiteralList body;
   friend istream& operator>>(istream&, clause&);   //input operator

   public :

     clause();                                   // clause constructor
     ~clause();                                  // clause destructor
     clause(const Literal&, LiteralList&);       // constructor
     clause(const clause&);                      // copy constructor
     Literal& gethead();                         // To get the head
     LiteralList& getbody() ;                    // To get the body
     clause& Instance();                         // To build clause Instance
     clause& operator=(const clause&);           // Assignment operator
     void display_clause() const;                // To display the clause
};

#endif
/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       Database.h                               */
/*      Description     :       Definitions and methods for Database Class  */
```

```
/*                                                                              */
/*                                                                              */
/******************************************************************************/

#ifndef DATABASE_H
#define DATABASE_H

#define MAXCLAUSES 100

#include <iostream.h>
#include "Bool.h"
#include "Symbol.h"
#include "Clause.h"


//  ***  Class - Database ***

class database
{
  int no_of_clauses;
  clause clausearray[MAXCLAUSES];

  friend istream& operator>>(istream&,database&); //Input operator

  public :

    database(char *);                    //constructor
    ~database();                         //destructor
    database();                          //constructor
    Boolean inrange(int) const;          //range check
    int FindCl(int, const symbol& ) ;    //symbol existence check
    clause& operator[](int) ;            //to get clause reference

};

#endif
/*****************************************************************************
*****/
/*
    */
/*
    */
/*      Module            :       Error.h
    */
/*      Description       :       Definitions for error generating routine
    */
/*
    */
/*
    */
/*****************************************************************************
*****/

#ifndef ERROR_H
#define ERROR_H

extern int no_of_errors;
extern void error (const char*);
#endif
/******************************************************************************/
```

```
/*                                                                       */
/*                                                                       */
/*      Module          :       Global.h                                 */
/*      Description     :       Global definitions                       */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

int seqno;                          //Seqno for Instantiating a variable

/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       Infer.h                                  */
/*      Description     :       Definitions for Inference Class and its  */
/*                              methods                                  */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#ifndef INFER_H
#define INFER_H

#include "Bool.h"
#include "Symbollist.h"
#include "Database.h"
#include "Substitution.h"

//    Class - Inference Engine

class inference
{

  public :
    inference();                                    // Constructor
    ~inference();                                   // Destructor
    inference(const inference&);                    // Copy Constructor
    Boolean Establish(LiteralList&, database& );    // Inference Engine
    Boolean Match( Literal&,  Literal* );           // To match the Literals

  private :

    Substitution    Subs;
};

#endif

/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       Lex.h                                    */
/*      Description     :       Definitions for Lexical Analysis         */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#ifndef LEX_H
#define LEX_H

#include <ctype.h>
```

```
#include <iostream.h>

enum token_value { DOT = '.', HYPHEN = '-',  COMMA = ',', COLON = ':',
                   NAME, PREDICATE, VARIABLE, CONSTANT, LP = '(', RP = ')',
                   iff_ok, END, INI, ERROR };

extern token_value curr_tok;
extern token_value prev_tok;
extern token_value get_token(istream&);

#endif
/*****************************************************************************/
/*                                                                           */
/*                                                                           */
/*      Module          :       List.h                                       */
/*      Description     :       List Class definitions & Implementaitons     */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/
#ifndef LIST_H
#define LIST_H

#include <stream.h>
#include <stdlib.h>
#include "Bool.h"

// the nodes to go in the list are of type ListNode<T>

template < class T >
class ListNode{
public:
        ListNode( );
        ~ListNode( );
        T* getdata( );
        ListNode<T>* getnext();
        void putdata( T* );
        void putnext( ListNode<T>* );
        void incref( );
private:
        T *data;
        ListNode<T> *next;
        int ref;
};

template < class T > class ListIterator;

template < class T >
class List{
        friend class ListIterator< T >;

        friend T& head( List<T>& L );
        friend List<T>& tail( List<T>& L );

public:
        List( );
        List( const List<T>& );
        ~List( );
        List<T>& operator=( List<T>& );
```

```
        void prepend( const T& x );
        void append( const T& x );
        Boolean isEmpty( ) const;
        Boolean contains( const T& x ) const;

private:
        ListNode<T>* ap;
};

template < class T >
class ListIterator{
public:
        ListIterator( List< T >& L );
        ~ListIterator( );

        T* first();
        T* next();
private:
        List<T>&        theList;
        ListNode<T>*    theCurrentNode;
};

//g++ needs implementation in this file

// implementation of ListNode

template < class T >
ListNode<T>::ListNode( )
{
        data = NULL;
        next = NULL;
        ref = 0;
}

template < class T >
ListNode<T>::~ListNode( )
{
        if( (--ref) == 0 ){
                if( next != NULL )
                        delete next;
                if( data != NULL )
                        delete data;
        };
}

template < class T >
ListNode<T>* ListNode<T>::getnext( )
{
        return next;
}

template < class T >
T* ListNode<T>::getdata( )
{
        return data;
}

template < class T >
void ListNode<T>::putnext( ListNode<T>* L )
{
```

```
        next = L;
}

template < class T >
void ListNode<T>::putdata( T* x )
{
        data = x;
}

template < class T >
void ListNode<T>::incref( )
{
        ref += 1;
}

//implementation of List

template < class T >
List<T>::List(  )
{
        ap = NULL;
}

template < class T >
List<T>::List( const List<T>& L  )
{
        ap = L.ap;
        if( L.ap != NULL )
                L.ap->incref();
}

template < class T >
List<T>::~List( )
{
        if( ap != NULL )
                delete ap;
}

template < class T >
List<T>& List<T>::operator=( List<T>& L )
{
        ap = L.ap;
        if( L.ap != NULL )
                L.ap->incref();
        return *this;
}

template < class T >
T& head( List<T>& L )
{
        if( L.ap != NULL ){
                return  *(L.ap->getdata());
        }
        else{
                cout << "head of empty data \n";
                exit( 1 );
        }
}

template < class T >
```

```
List<T>& tail( List<T>& L )
{
        if( L.ap != NULL ){
                List<T>*        p = new List<T>;
                p->ap = L.ap->getnext();
                return  *p;
        }
        else{
                cout << "tail of empty data \n";
                exit( 1 );
        }
}

template < class T >
void List<T>::prepend( const T& x )
{
        //insert x at the head of the list
        ListNode<T>*    p = new ListNode<T>;
        T*              xp = new T;
        *xp = x;
        p->putdata( xp );
        p->putnext( ap );
        ap = p;
}

template < class T >
void List<T>::append( const T& x )
{
        //insert x at the end of the list

        //create new node
        ListNode<T>*    p = new ListNode<T>;
        T*              xp = new T;
        *xp = x;
        p->putdata( xp );
        p->putnext( NULL );

        if( ap == NULL )
        {
                ap = p;
        }
        else
        {
                //traverse to the end of the list
                ListNode<T>* last = ap;
                while(last->getnext() != NULL) last = last->getnext();
                last->putnext( p );
        }
}

template < class T >
Boolean List<T>::isEmpty( ) const
{
        return (ap == NULL);
}

template < class T >
Boolean List<T>::contains( const T& x ) const
{
        //dummy
```

```
        return FALSE;
}

// implementation of ListIterator

template < class T >
ListIterator<T>::ListIterator( List<T>& L )
: theList( L )
{
        theCurrentNode = L.ap;
}

template < class T >
ListIterator<T>::~ListIterator( )
{
}

template < class T >
T* ListIterator<T>::next( )
{
        ListNode<T>* ap;
        ap = ap->getnext();
        if( ap != NULL)
                return ap->getdata();
        else
                return NULL;
}

template < class T >
T* ListIterator<T>::first( )
{
        ListNode<T>* ap;
        ap = theList.ap;
        if( ap != NULL)
                return ap->getdata();
        else
                return NULL;
}

#endif
/***************************************************************************/
/*                                                                         */
/*                                                                         */
/*      Module          :       Literal.h                                  */
/*      Description     :       Definitions for Literal Class and methods  */
/*                                                                         */
/*                                                                         */
/***************************************************************************/

#ifndef LITERAL_H
#define LITERAL_H

#include <iostream.h>
#include "Bool.h"
#include "Lex.h"
#include "Syn.h"
#include "Symbol.h"
#include "Symbollist.h"

class Literal{
```

```
        friend istream& operator>>(istream& , Literal& ); //input operator

public :
        Literal();
        Literal( symbol&, symbollist& );
        Literal( const Literal& );
        ~Literal();

        Literal& operator=( const Literal& );
        Boolean operator==( const Literal& ) ;

        symbol&     predicate();                //fetch the predicate name
        symbollist* arguments() ;               //fetch the list of arguments
        void        ChLVarName(int);            //To Instantiate the Literal args
        void        display_Literal() const;    //To Display the Literal

private :
        symbol          pred;                   //Predicate
        symbollist      arglist;                //Arguments List
};

#endif



/**************************************************************************/
/*                                                                        */
/*                                                                        */
/*      Module          :       LiteralList.h                             */
/*      Description     :       Definitions for LiteralList Class and its */
/*                              methods                                   */
/*                                                                        */
/*                                                                        */
/**************************************************************************/

#ifndef LITERALLIST_H
#define LITERALLIST_H

// A list of literals for Datalog

#include <iostream.h>
#include "Bool.h"
#include "Error.h"
#include "Literal.h"


class LiteralNode
{

  friend class LiteralList;

  public :

    LiteralNode(const Literal&, LiteralNode*);          //Constructor
    ~LiteralNode();
    Literal*     getLtrl()           { return Ltrl; }
    LiteralNode* getLtrlNext() const { return LtrlNext; }
```

```
  private :
    Literal     *Ltrl;
    LiteralNode *LtrlNext;

};

class LiteralList
{

  friend istream& operator>>(istream&, LiteralList&);  //input operator

  public :
    LiteralList();                                    // Constructor
    ~LiteralList();                                   // Destructor
    LiteralList(LiteralNode&);                        // Constructor
    LiteralList(const LiteralList&);                  // Copy Constructor
    void append(const Literal&);                      // Append a Literal
    LiteralNode* getLNode();                          // Get the first LiteralNode
    LiteralList rest();                               // Get next LiteralList
    LiteralList& operator=(const LiteralList&);       // Assignment Operator
    LiteralList& copycat2(LiteralList&);              // To concatenate two LiteralLis
ts
    inline Boolean isEmpty() const;                   // To check for empty LitList
    void ChLLVarName(int);
    void display_LiteralList() const;

    LiteralNode     *Literalhead;

private:
    LiteralList(LiteralNode*);                        // Constructor

};


inline Boolean LiteralList::isEmpty() const
{
  return(Literalhead == NULL);
}

#endif

/**************************************************************************/
/*                                                                        */
/*                                                                        */
/*      Module          :       Query.h                                   */
/*      Description     :       Definitions for Query Class and its methods */
/*                                                                        */
/*                                                                        */
/**************************************************************************/

#ifndef QUERY_H
#define QUERY_H

#include <iostream.h>
#include "Bool.h"
#include "LiteralList.h"

class query
{
        friend istream& operator>>(istream&, query&);  //input operator
```

```
        public :
          query();                                 //constructor
          query(char*);                            //constructor
          ~query();                                //destructor
          query(const query&);                     //copy constructor
          Boolean checkdot();                      //To sense the end of query
          LiteralList &getgoals();                 //To get the goals pointer

        private :
          LiteralList goals;
};

#endif



/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*      Module          :       Substitution.h                           */
/*      Description     :       Definitions for Substitution Class and its */
/*                              methods                                  */
/*                                                                       */
/*************************************************************************/

#ifndef SUBSTITUTION_H
#define SUBSTITUTION_H

// A substitution is a map from Symbols to Symbols in Datalog

#include <iostream.h>
#include "Bool.h"
#include "Symbol.h"
#include "List.h"
#include "Literal.h"
#include "LiteralList.h"

class   SymbolPair;
friend  class symbollist;
typedef SymbolPair* SymbolPairPtr;

class Substitution
{
public :
        Substitution();
        Substitution(const Substitution&);
        ~Substitution();
        Substitution& operator=(Substitution&);

        symbol* operator[](const symbol&) ;         // the substitute for s
        void add(const symbol& , const symbol&);    // incorporate the new
                                                    // substitution
        void unify(node*, node*);                   // Unify the Literals
        LiteralList& apply(LiteralList&);           // Apply the substitutions
        void concat(Substitution&);                 // incorporate s
        void InitSubsList();                        // Initialize the SubsList

        List< SymbolPairPtr >   map;
```

```
};

class SymbolPair{
public:
        SymbolPair();
        SymbolPair( const symbol&, const symbol& );
        ~SymbolPair();
        SymbolPair& operator=(const SymbolPair&);

        symbol* first();                        // Get the First symbol
        symbol* second();                       // Get the Second symbol

private:
        symbol* fst;
        symbol* snd;
};
#endif
/***********************************************************************/
/*                                                                     */
/*                                                                     */
/*      Module          :       symbol.h                               */
/*      Description     :       Definitions for Symbol Class and its methods*/
/*                                                                     */
/*                                                                     */
/***********************************************************************/

#ifndef SYMBOL_H
#define SYMBOL_H

#include <iostream.h>
#include "Lex.h"
#include "Bool.h"


// PRED: Predicate, VAR: Variable, CONST: Constant

enum SymKind {PRED, VAR, CONST};

// *** Defining the Basic Element Structure used in the program ***

class symbol
{
   friend Boolean operator==(const symbol&, const symbol&);
   friend Boolean operator!=(const symbol&,const symbol&);
   friend ostream& operator<<(ostream&, const symbol&);
   friend istream& operator>>(istream&, symbol&);

public :
    symbol(char* = 0);
    symbol(char* , const SymKind&);          // constructor
    ~symbol();                               // destructor
    symbol(const symbol&);                   // copy constructor
    symbol& operator=(const symbol&);        // assignment operator
    SymKind& getSKind();                     // To get SymKind
    void ChVarName(int);                     // To change the Var
                                             // to new name
private :
      int       len;
      char*     Sname;
      SymKind   Skind;
```

```
};

#endif
/**************************************************************************/
/*                                                                        */
/*                                                                        */
/*      Module          :       Symbollist.h                              */
/*      Description     :       Definitions for SymbollistClass and its   */
/*                              methods                                   */
/*                                                                        */
/*                                                                        */
/**************************************************************************/

#ifndef SYMBOLLIST_H
#define SYMBOLLIST_H

#include <iostream.h>
#include "Bool.h"
#include "Symbol.h"

// *** Defining the Basic List Structure used in the program ***

class node
  {
    friend class symbollist;

    private :

       symbol *info;
       node *next;

    public :

       node(const symbol&, node*);           //node constructor
       ~node();                              //node destructor
       symbol& getinfo() const;              //to get the symbol
       symbol* getinfo1() const;             //to get the symbol pointer
       node* getnext() const;                //to get the next node pointer
  };

class symbollist
{
  friend istream& operator>>(istream&, symbollist&);  //input operator

  public :

     symbollist();                                 // constructor
     symbollist(symbol&);                          // constructor
     ~symbollist();                                // destructor
     symbollist(const symbollist&);                // copy constructor
     void append(const symbol&);                   // append a node
     node* gethead() { return head; }              // Return the first node
     inline Boolean isEmpty() const;               // to check for empty list
     symbollist& operator = (const symbollist&);   // assignment operator
     Boolean operator == (const symbollist&);      // equality operator
     void display_symbollist() const;              // to display the symbollist
     void ChSlVarName(int) const;                  // To instantiate the Sl

  private :
     node *head;
```

```
    symbollist(node*);                          // constructor
};

inline Boolean symbollist::isEmpty() const      // to check for empty Symlist
{
       return (head == NULL);
}

#endif
/***************************************************************************/
/*                                                                         */
/*                                                                         */
/*      Module          :       Syn.h                                      */
/*      Description     :       Definitions for Syntactic analysis        */
/*                                                                         */
/*                                                                         */
/***************************************************************************/

#ifndef SYN_H
#define SYN_H

#include <iostream.h>

extern  parse(istream&);

#endif
```

APPENDIX - B

Experimental Results

```
pasta ( X ) :- cup ( X ), egg ( X , Y ) .
cup ( one ) .
cup ( two ) .
egg ( one , one ) .
egg ( two , two ) .
```

```
*** Proceeding for Database creation ***
Input Clauses File: test0
Number of clauses: 5

Clause Errors = 0
Database is created Successfully

*** Proceeding to build the Query ***
Input Query File: qry
Query Errors = 0
User's Query : pasta(two)

*** Datalog Interpreter is Trying to Establish User's Query ***

Goals : pasta(two)
Looking for Clause with Predicate: pasta
Finding Clause starting with i= 0 ->Clause Found at i= 0
Clause: pasta(X) :- cup(X) , egg(X,Y).

Clause - BEFORE Instantiation
pasta(X) :- cup(X) , egg(X,Y).

Clause - AFTER Instantiation
pasta(X001) :- cup(X001) , egg(X001,Y001).

Trying to Establish the goals : pasta(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : pasta(X001) and pasta(two)
OK - Unification is Successfull

Goals : cup(two) , egg(two,Y001)
Looking for Clause with Predicate: cup
Finding Clause starting with i= 0 ->Clause Found at i= 1
Clause: cup(one) :- ( No Body ).

Clause - BEFORE Instantiation
cup(one) :- ( No Body ).

Clause - AFTER Instantiation
cup(one) :- ( No Body ).

Trying to Establish the goals : cup(two) , egg(two,Y001)
Substitutions: (X001, two)
Trying to Unify the Literals  : cup(one) and cup(two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : cup(two) , egg(two,Y001)
Finding Clause starting with i= 2 ->Clause Found at i= 2

Clause - BEFORE Instantiation
cup(two) :- ( No Body ).

Clause - AFTER Instantiation
cup(two) :- ( No Body ).

Trying to Establish the goals : cup(two) , egg(two,Y001)
Substitutions: No Substitutions
```

```
Trying to Unify the Literals  : cup(two) and cup(two)
OK - Unification is Successfull

Goals : egg(two,Y001)
Looking for Clause with Predicate: egg
Finding Clause starting with i= 0 ->Clause Found at i= 3
Clause: egg(one,one) :- ( No Body ).

Clause - BEFORE Instantiation
egg(one,one) :- ( No Body ).

Clause - AFTER Instantiation
egg(one,one) :- ( No Body ).

Trying to Establish the goals : egg(two,Y001)
Substitutions: No Substitutions

Trying to Unify the Literals  : egg(one,one) and egg(two,Y001)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : egg(two,Y001)
Finding Clause starting with i= 4 ->Clause Found at i= 4

Clause - BEFORE Instantiation
egg(two,two) :- ( No Body ).

Clause - AFTER Instantiation
egg(two,two) :- ( No Body ).

Trying to Establish the goals : egg(two,Y001)
Substitutions: No Substitutions

Trying to Unify the Literals  : egg(two,two) and egg(two,Y001)
OK - Unification is Successfull

*** SUCCESS - User's Query is Established ***
```

```
pasta ( X ) :- cup ( X ) .
cup ( X ) :- cup2 ( Y ), egg ( X , Y ) .
cup2 ( one ) .
cup2 ( three ).
egg ( two , three ) .
```

```
*** Proceeding for Database creation ***
Input Clauses File: test1
Number of clauses: 5

Clause Errors = 0
Database is created Successfully

*** Proceeding to build the Query ***
Input Query File: qry
Query Errors = 0
User's Query : pasta(two)

*** Datalog Interpreter is Trying to Establish User's Query ***

Goals : pasta(two)
Looking for Clause with Predicate: pasta
Finding Clause starting with i= 0 ->Clause Found at i= 0
Clause: pasta(X) :- cup(X).

Clause - BEFORE Instantiation
pasta(X) :- cup(X).

Clause - AFTER Instantiation
pasta(X001) :- cup(X001).

Trying to Establish the goals : pasta(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : pasta(X001) and pasta(two)
OK - Unification is Successfull

Goals : cup(two)
Looking for Clause with Predicate: cup
Finding Clause starting with i= 0 ->Clause Found at i= 1
Clause: cup(X) :- cup2(Y) , egg(X,Y).

Clause - BEFORE Instantiation
cup(X) :- cup2(Y) , egg(X,Y).

Clause - AFTER Instantiation
cup(X002) :- cup2(Y002) , egg(X002,Y002).

Trying to Establish the goals : cup(two)
Substitutions: (X001, two)
Trying to Unify the Literals  : cup(X002) and cup(two)
OK - Unification is Successfull

Goals : cup2(Y002) , egg(two,Y002)
Looking for Clause with Predicate: cup2
Finding Clause starting with i= 0 ->Clause Found at i= 2
Clause: cup2(one) :- ( No Body ).

Clause - BEFORE Instantiation
cup2(one) :- ( No Body ).

Clause - AFTER Instantiation
cup2(one) :- ( No Body ).

Trying to Establish the goals : cup2(Y002) , egg(two,Y002)
Substitutions: (X001, two) (X002, two)
```

```
Trying to Unify the Literals   : cup2(one) and cup2(Y002)
OK - Unification is Successfull

Goals : egg(two,one)
Looking for Clause with Predicate: egg
Finding Clause starting with i= 0 ->Clause Found at i= 4
Clause: egg(two,three) :- ( No Body ).

Clause - BEFORE Instantiation
egg(two,three) :- ( No Body ).

Clause - AFTER Instantiation
egg(two,three) :- ( No Body ).

Trying to Establish the goals : egg(two,one)
Substitutions: (X001, two) (X002, two) (Y002, one)
Trying to Unify the Literals   : egg(two,three) and egg(two,one)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : egg(two,one)
Finding Clause starting with i= 5 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : cup2(Y002) , egg(two,Y002)
Finding Clause starting with i= 3 ->Clause Found at i= 3

Clause - BEFORE Instantiation
cup2(three) :- ( No Body ).

Clause - AFTER Instantiation
cup2(three) :- ( No Body ).

Trying to Establish the goals : cup2(Y002) , egg(two,Y002)
Substitutions: No Substitutions

Trying to Unify the Literals   : cup2(three) and cup2(Y002)
OK - Unification is Successfull

Goals : egg(two,three)
Looking for Clause with Predicate: egg
Finding Clause starting with i= 0 ->Clause Found at i= 4
Clause: egg(two,three) :- ( No Body ).

Clause - BEFORE Instantiation
egg(two,three) :- ( No Body ).

Clause - AFTER Instantiation
egg(two,three) :- ( No Body ).

Trying to Establish the goals : egg(two,three)
Substitutions: (Y002, three)
Trying to Unify the Literals   : egg(two,three) and egg(two,three)
OK - Unification is Successfull

*** SUCCESS - User's Query is Established ***
```

```
pasta ( X ) :- cup ( X ) .
cup ( X ) :- cup2 ( X ) , egg ( X , Y ) .
cup ( X ) :- cup2 ( Y ), egg ( X , Y ) .
cup2 ( one ) .
cup2 ( three ).
egg ( one , one ) .
egg ( two , three ) .
```

```
*** Proceeding for Database creation ***
Input Clauses File: test2
Number of clauses: 7

Clause Errors = 0
Database is created Successfully

*** Proceeding to build the Query ***
Input Query File: qry
Query Errors = 0
User's Query : pasta(two)

*** Datalog Interpreter is Trying to Establish User's Query ***

Goals : pasta(two)
Looking for Clause with Predicate: pasta
Finding Clause starting with i= 0 ->Clause Found at i= 0
Clause: pasta(X) :- cup(X).

Clause - BEFORE Instantiation
pasta(X) :- cup(X).

Clause - AFTER Instantiation
pasta(X001) :- cup(X001).

Trying to Establish the goals : pasta(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : pasta(X001) and pasta(two)
OK - Unification is Successfull

Goals : cup(two)
Looking for Clause with Predicate: cup
Finding Clause starting with i= 0 ->Clause Found at i= 1
Clause: cup(X) :- cup2(X) , egg(X,Y).

Clause - BEFORE Instantiation
cup(X) :- cup2(X) , egg(X,Y).

Clause - AFTER Instantiation
cup(X002) :- cup2(X002) , egg(X002,Y002).

Trying to Establish the goals : cup(two)
Substitutions: (X001, two)
Trying to Unify the Literals  : cup(X002) and cup(two)
OK - Unification is Successfull

Goals : cup2(two) , egg(two,Y002)
Looking for Clause with Predicate: cup2
Finding Clause starting with i= 0 ->Clause Found at i= 3
Clause: cup2(one) :- ( No Body ).

Clause - BEFORE Instantiation
cup2(one) :- ( No Body ).

Clause - AFTER Instantiation
cup2(one) :- ( No Body ).

Trying to Establish the goals : cup2(two) , egg(two,Y002)
Substitutions: (X001, two) (X002, two)
```

```
Trying to Unify the Literals  : cup2(one) and cup2(two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : cup2(two) , egg(two,Y002)
Finding Clause starting with i= 4 ->Clause Found at i= 4

Clause - BEFORE Instantiation
cup2(three) :- ( No Body ).

Clause - AFTER Instantiation
cup2(three) :- ( No Body ).

Trying to Establish the goals : cup2(two) , egg(two,Y002)
Substitutions: No Substitutions

Trying to Unify the Literals  : cup2(three) and cup2(two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : cup2(two) , egg(two,Y002)
Finding Clause starting with i= 5 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : cup(two)
Finding Clause starting with i= 2 ->Clause Found at i= 2

Clause - BEFORE Instantiation
cup(X) :- cup2(Y) , egg(X,Y).

Clause - AFTER Instantiation
cup(X005) :- cup2(Y005) , egg(X005,Y005).

Trying to Establish the goals : cup(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : cup(X005) and cup(two)
OK - Unification is Successfull

Goals : cup2(Y005) , egg(two,Y005)
Looking for Clause with Predicate: cup2
Finding Clause starting with i= 0 ->Clause Found at i= 3
Clause: cup2(one) :- ( No Body ).

Clause - BEFORE Instantiation
cup2(one) :- ( No Body ).

Clause - AFTER Instantiation
cup2(one) :- ( No Body ).

Trying to Establish the goals : cup2(Y005) , egg(two,Y005)
Substitutions: (X005, two)
Trying to Unify the Literals  : cup2(one) and cup2(Y005)
OK - Unification is Successfull

Goals : egg(two,one)
Looking for Clause with Predicate: egg
Finding Clause starting with i= 0 ->Clause Found at i= 5
Clause: egg(one,one) :- ( No Body ).
```

```
Clause - BEFORE Instantiation
egg(one,one) :- ( No Body ).

Clause - AFTER Instantiation
egg(one,one) :- ( No Body ).

Trying to Establish the goals : egg(two,one)
Substitutions: (X005, two) (Y005, one)
Trying to Unify the Literals  : egg(one,one) and egg(two,one)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : egg(two,one)
Finding Clause starting with i= 6 ->Clause Found at i= 6

Clause - BEFORE Instantiation
egg(two,three) :- ( No Body ).

Clause - AFTER Instantiation
egg(two,three) :- ( No Body ).

Trying to Establish the goals : egg(two,one)
Substitutions: No Substitutions

Trying to Unify the Literals  : egg(two,three) and egg(two,one)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : egg(two,one)
Finding Clause starting with i= 7 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : cup2(Y005) , egg(two,Y005)
Finding Clause starting with i= 4 ->Clause Found at i= 4

Clause - BEFORE Instantiation
cup2(three) :- ( No Body ).

Clause - AFTER Instantiation
cup2(three) :- ( No Body ).

Trying to Establish the goals : cup2(Y005) , egg(two,Y005)
Substitutions: No Substitutions

Trying to Unify the Literals  : cup2(three) and cup2(Y005)
OK - Unification is Successfull

Goals : egg(two,three)
Looking for Clause with Predicate: egg
Finding Clause starting with i= 0 ->Clause Found at i= 5
Clause: egg(one,one) :- ( No Body ).

Clause - BEFORE Instantiation
egg(one,one) :- ( No Body ).

Clause - AFTER Instantiation
egg(one,one) :- ( No Body ).

Trying to Establish the goals : egg(two,three)
Substitutions: (Y005, three)
```

```
Trying to Unify the Literals  : egg(one,one) and egg(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : egg(two,three)
Finding Clause starting with i= 6 ->Clause Found at i= 6

Clause - BEFORE Instantiation
egg(two,three) :- ( No Body ).

Clause - AFTER Instantiation
egg(two,three) :- ( No Body ).

Trying to Establish the goals : egg(two,three)
Substitutions: No Substitutions

Trying to Unify the Literals  : egg(two,three) and egg(two,three)
OK - Unification is Successfull

*** SUCCESS - User's Query is Established ***
```

```
pasta ( XX ) :- cupFlour ( XX ) , egg ( XX ) , tblspWater ( YY ) , twice ( XX ,
YY ) , tspSalt ( XX ) , tspOil ( XX ) .
cupFlour ( XX ) :- haveCupFlour ( XX ).
cupFlour ( XX ) :- haveCupFlour ( YY ) , less ( XX , YY ).
cupMilk ( XX ) :- haveCupMilk ( XX ).
cupMilk ( XX ) :- haveCupMilk ( YY ) , less ( XX , YY ).
egg ( XX ) :- haveEgg ( XX ).
egg ( XX ) :- haveEgg ( YY ) , less ( XX , YY ).
tspSalt ( XX ) :- haveTspSalt ( XX ).
tspSalt ( XX ) :- haveTspSalt ( YY ) , less ( XX , YY ).
tspOil ( XX ) :- haveTspOil ( XX ).
tspOil ( XX ) :- haveTspOil ( YY ) , less ( XX , YY ).
twice ( one , two ).
twice ( two , four ).
twice ( three , six ).
less ( one , two ).
less ( two , three ).
less ( three , four ).
tblspWater ( one ).
tblspWater ( two ).
tblspWater ( three ).
tblspWater ( four ).
haveCupFlour ( three ).
haveEgg ( two ).
haveTspSalt ( three ).
haveTspSalt ( four ).
haveTspOil ( three ).
```

```
*** Proceeding for Database creation ***
Input Clauses File: test3
Number of clauses: 26

Clause Errors = 0
Database is created Successfully

*** Proceeding to build the Query ***
Input Query File: qry
Query Errors = 0
User's Query : pasta(two)

*** Datalog Interpreter is Trying to Establish User's Query ***

Goals : pasta(two)
Looking for Clause with Predicate: pasta
Finding Clause starting with i= 0 ->Clause Found at i= 0
Clause: pasta(XX) :- cupFlour(XX) , egg(XX) , tblspWater(YY) , twice(XX,YY) , ts
pSalt(XX) , tspOil(XX).

Clause - BEFORE Instantiation
pasta(XX) :- cupFlour(XX) , egg(XX) , tblspWater(YY) , twice(XX,YY) , tspSalt(XX
) , tspOil(XX).

Clause - AFTER Instantiation
pasta(XX001) :- cupFlour(XX001) , egg(XX001) , tblspWater(YY001) , twice(XX001,Y
Y001) , tspSalt(XX001) , tspOil(XX001).

Trying to Establish the goals : pasta(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : pasta(XX001) and pasta(two)
OK - Unification is Successfull

Goals : cupFlour(two) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspSal
t(two) , tspOil(two)
Looking for Clause with Predicate: cupFlour
Finding Clause starting with i= 0 ->Clause Found at i= 1
Clause: cupFlour(XX) :- haveCupFlour(XX).

Clause - BEFORE Instantiation
cupFlour(XX) :- haveCupFlour(XX).

Clause - AFTER Instantiation
cupFlour(XX002) :- haveCupFlour(XX002).

Trying to Establish the goals : cupFlour(two) , egg(two) , tblspWater(YY001) , t
wice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: (XX001, two)
Trying to Unify the Literals  : cupFlour(XX002) and cupFlour(two)
OK - Unification is Successfull

Goals : haveCupFlour(two) , egg(two) , tblspWater(YY001) , twice(two,YY001) , ts
pSalt(two) , tspOil(two)
Looking for Clause with Predicate: haveCupFlour
Finding Clause starting with i= 0 ->Clause Found at i= 21
Clause: haveCupFlour(three) :- ( No Body ).

Clause - BEFORE Instantiation
haveCupFlour(three) :- ( No Body ).
```

```
Clause - AFTER Instantiation
haveCupFlour(three) :- ( No Body ).

Trying to Establish the goals : haveCupFlour(two) , egg(two) , tblspWater(YY001)
 , twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: (XX001, two) (XX002, two)
Trying to Unify the Literals  : haveCupFlour(three) and haveCupFlour(two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveCupFlour(two) , egg(two) , tblspWater(YY001) , twice(two,YY001) , ts
pSalt(two) , tspOil(two)
Finding Clause starting with i= 22 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : cupFlour(two) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspSal
t(two) , tspOil(two)
Finding Clause starting with i= 2 ->Clause Found at i= 2

Clause - BEFORE Instantiation
cupFlour(XX) :- haveCupFlour(YY) , less(XX,YY).

Clause - AFTER Instantiation
cupFlour(XX004) :- haveCupFlour(YY004) , less(XX004,YY004).

Trying to Establish the goals : cupFlour(two) , egg(two) , tblspWater(YY001) , t
wice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : cupFlour(XX004) and cupFlour(two)
OK - Unification is Successfull

Goals : haveCupFlour(YY004) , less(two,YY004) , egg(two) , tblspWater(YY001) , t
wice(two,YY001) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: haveCupFlour
Finding Clause starting with i= 0 ->Clause Found at i= 21
Clause: haveCupFlour(three) :- ( No Body ).

Clause - BEFORE Instantiation
haveCupFlour(three) :- ( No Body ).

Clause - AFTER Instantiation
haveCupFlour(three) :- ( No Body ).

Trying to Establish the goals : haveCupFlour(YY004) , less(two,YY004) , egg(two)
 , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: (XX004, two)
Trying to Unify the Literals  : haveCupFlour(three) and haveCupFlour(YY004)
OK - Unification is Successfull

Goals : less(two,three) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspS
alt(two) , tspOil(two)
Looking for Clause with Predicate: less
Finding Clause starting with i= 0 ->Clause Found at i= 14
Clause: less(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
less(one,two) :- ( No Body ).
```

```
Clause - AFTER Instantiation
less(one,two) :- ( No Body ).

Trying to Establish the goals : less(two,three) , egg(two) , tblspWater(YY001) ,
 twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: (XX004, two) (YY004, three)
Trying to Unify the Literals  : less(one,two) and less(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,three) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspS
alt(two) , tspOil(two)
Finding Clause starting with i= 15 ->Clause Found at i= 15

Clause - BEFORE Instantiation
less(two,three) :- ( No Body ).

Clause - AFTER Instantiation
less(two,three) :- ( No Body ).

Trying to Establish the goals : less(two,three) , egg(two) , tblspWater(YY001) ,
 twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : less(two,three) and less(two,three)
OK - Unification is Successfull

Goals : egg(two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(
two)
Looking for Clause with Predicate: egg
Finding Clause starting with i= 0 ->Clause Found at i= 5
Clause: egg(XX) :- haveEgg(XX).

Clause - BEFORE Instantiation
egg(XX) :- haveEgg(XX).

Clause - AFTER Instantiation
egg(XX008) :- haveEgg(XX008).

Trying to Establish the goals : egg(two) , tblspWater(YY001) , twice(two,YY001)
, tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : egg(XX008) and egg(two)
OK - Unification is Successfull

Goals : haveEgg(two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tsp
Oil(two)
Looking for Clause with Predicate: haveEgg
Finding Clause starting with i= 0 ->Clause Found at i= 22
Clause: haveEgg(two) :- ( No Body ).

Clause - BEFORE Instantiation
haveEgg(two) :- ( No Body ).

Clause - AFTER Instantiation
haveEgg(two) :- ( No Body ).

Trying to Establish the goals : haveEgg(two) , tblspWater(YY001) , twice(two,YY0
01) , tspSalt(two) , tspOil(two)
```

```
Substitutions: (XX008, two)
Trying to Unify the Literals  : haveEgg(two) and haveEgg(two)
OK - Unification is Successfull

Goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: tblspWater
Finding Clause starting with i= 0 ->Clause Found at i= 17
Clause: tblspWater(one) :- ( No Body ).

Clause - BEFORE Instantiation
tblspWater(one) :- ( No Body ).

Clause - AFTER Instantiation
tblspWater(one) :- ( No Body ).

Trying to Establish the goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(t
wo) , tspOil(two)
Substitutions: (XX008, two)
Trying to Unify the Literals  : tblspWater(one) and tblspWater(YY001)
OK - Unification is Successfull

Goals : twice(two,one) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: twice
Finding Clause starting with i= 0 ->Clause Found at i= 11
Clause: twice(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
twice(one,two) :- ( No Body ).

Clause - AFTER Instantiation
twice(one,two) :- ( No Body ).

Trying to Establish the goals : twice(two,one) , tspSalt(two) , tspOil(two)
Substitutions: (XX008, two) (YY001, one)
Trying to Unify the Literals  : twice(one,two) and twice(two,one)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,one) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 12 ->Clause Found at i= 12

Clause - BEFORE Instantiation
twice(two,four) :- ( No Body ).

Clause - AFTER Instantiation
twice(two,four) :- ( No Body ).

Trying to Establish the goals : twice(two,one) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(two,four) and twice(two,one)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,one) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 13 ->Clause Found at i= 13

Clause - BEFORE Instantiation
twice(three,six) :- ( No Body ).
```

```
Clause - AFTER Instantiation
twice(three,six) :- ( No Body ).

Trying to Establish the goals : twice(two,one) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(three,six) and twice(two,one)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,one) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 14 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 18 ->Clause Found at i= 18

Clause - BEFORE Instantiation
tblspWater(two) :- ( No Body ).

Clause - AFTER Instantiation
tblspWater(two) :- ( No Body ).

Trying to Establish the goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(t
wo) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tblspWater(two) and tblspWater(YY001)
OK - Unification is Successfull

Goals : twice(two,two) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: twice
Finding Clause starting with i= 0 ->Clause Found at i= 11
Clause: twice(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
twice(one,two) :- ( No Body ).

Clause - AFTER Instantiation
twice(one,two) :- ( No Body ).

Trying to Establish the goals : twice(two,two) , tspSalt(two) , tspOil(two)
Substitutions: (YY001, two)
Trying to Unify the Literals  : twice(one,two) and twice(two,two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,two) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 12 ->Clause Found at i= 12

Clause - BEFORE Instantiation
twice(two,four) :- ( No Body ).

Clause - AFTER Instantiation
twice(two,four) :- ( No Body ).

Trying to Establish the goals : twice(two,two) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(two,four) and twice(two,two)
```

```
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,two) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 13 ->Clause Found at i= 13

Clause - BEFORE Instantiation
twice(three,six) :- ( No Body ).

Clause - AFTER Instantiation
twice(three,six) :- ( No Body ).

Trying to Establish the goals : twice(two,two) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(three,six) and twice(two,two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,two) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 14 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 19 ->Clause Found at i= 19

Clause - BEFORE Instantiation
tblspWater(three) :- ( No Body ).

Clause - AFTER Instantiation
tblspWater(three) :- ( No Body ).

Trying to Establish the goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(t
wo) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tblspWater(three) and tblspWater(YY001)
OK - Unification is Successfull

Goals : twice(two,three) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: twice
Finding Clause starting with i= 0 ->Clause Found at i= 11
Clause: twice(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
twice(one,two) :- ( No Body ).

Clause - AFTER Instantiation
twice(one,two) :- ( No Body ).

Trying to Establish the goals : twice(two,three) , tspSalt(two) , tspOil(two)
Substitutions: (YY001, three)
Trying to Unify the Literals  : twice(one,two) and twice(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,three) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 12 ->Clause Found at i= 12

Clause - BEFORE Instantiation
```

```
twice(two,four) :- ( No Body ).

Clause - AFTER Instantiation
twice(two,four) :- ( No Body ).

Trying to Establish the goals : twice(two,three) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(two,four) and twice(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,three) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 13 ->Clause Found at i= 13

Clause - BEFORE Instantiation
twice(three,six) :- ( No Body ).

Clause - AFTER Instantiation
twice(three,six) :- ( No Body ).

Trying to Establish the goals : twice(two,three) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(three,six) and twice(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,three) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 14 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 20 ->Clause Found at i= 20

Clause - BEFORE Instantiation
tblspWater(four) :- ( No Body ).

Clause - AFTER Instantiation
tblspWater(four) :- ( No Body ).

Trying to Establish the goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(t
wo) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tblspWater(four) and tblspWater(YY001)
OK - Unification is Successfull

Goals : twice(two,four) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: twice
Finding Clause starting with i= 0 ->Clause Found at i= 11
Clause: twice(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
twice(one,two) :- ( No Body ).

Clause - AFTER Instantiation
twice(one,two) :- ( No Body ).

Trying to Establish the goals : twice(two,four) , tspSalt(two) , tspOil(two)
```

```
Substitutions: (YY001, four)
Trying to Unify the Literals  : twice(one,two) and twice(two,four)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,four) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 12 ->Clause Found at i= 12

Clause - BEFORE Instantiation
twice(two,four) :- ( No Body ).

Clause - AFTER Instantiation
twice(two,four) :- ( No Body ).

Trying to Establish the goals : twice(two,four) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(two,four) and twice(two,four)
OK - Unification is Successfull

Goals : tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: tspSalt
Finding Clause starting with i= 0 ->Clause Found at i= 7
Clause: tspSalt(XX) :- haveTspSalt(XX).

Clause - BEFORE Instantiation
tspSalt(XX) :- haveTspSalt(XX).

Clause - AFTER Instantiation
tspSalt(XX025) :- haveTspSalt(XX025).

Trying to Establish the goals : tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tspSalt(XX025) and tspSalt(two)
OK - Unification is Successfull

Goals : haveTspSalt(two) , tspOil(two)
Looking for Clause with Predicate: haveTspSalt
Finding Clause starting with i= 0 ->Clause Found at i= 23
Clause: haveTspSalt(three) :- ( No Body ).

Clause - BEFORE Instantiation
haveTspSalt(three) :- ( No Body ).

Clause - AFTER Instantiation
haveTspSalt(three) :- ( No Body ).

Trying to Establish the goals : haveTspSalt(two) , tspOil(two)
Substitutions: (XX025, two)
Trying to Unify the Literals  : haveTspSalt(three) and haveTspSalt(two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveTspSalt(two) , tspOil(two)
Finding Clause starting with i= 24 ->Clause Found at i= 24

Clause - BEFORE Instantiation
haveTspSalt(four) :- ( No Body ).
```

```
Clause - AFTER Instantiation
haveTspSalt(four) :- ( No Body ).

Trying to Establish the goals : haveTspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : haveTspSalt(four) and haveTspSalt(two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveTspSalt(two) , tspOil(two)
Finding Clause starting with i= 25 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tspSalt(two) , tspOil(two)
Finding Clause starting with i= 8 ->Clause Found at i= 8

Clause - BEFORE Instantiation
tspSalt(XX) :- haveTspSalt(YY) , less(XX,YY).

Clause - AFTER Instantiation
tspSalt(XX028) :- haveTspSalt(YY028) , less(XX028,YY028).

Trying to Establish the goals : tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tspSalt(XX028) and tspSalt(two)
OK - Unification is Successfull

Goals : haveTspSalt(YY028) , less(two,YY028) , tspOil(two)
Looking for Clause with Predicate: haveTspSalt
Finding Clause starting with i= 0 ->Clause Found at i= 23
Clause: haveTspSalt(three) :- ( No Body ).

Clause - BEFORE Instantiation
haveTspSalt(three) :- ( No Body ).

Clause - AFTER Instantiation
haveTspSalt(three) :- ( No Body ).

Trying to Establish the goals : haveTspSalt(YY028) , less(two,YY028) , tspOil(tw
o)
Substitutions: (XX028, two)
Trying to Unify the Literals  : haveTspSalt(three) and haveTspSalt(YY028)
OK - Unification is Successfull

Goals : less(two,three) , tspOil(two)
Looking for Clause with Predicate: less
Finding Clause starting with i= 0 ->Clause Found at i= 14
Clause: less(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
less(one,two) :- ( No Body ).

Clause - AFTER Instantiation
less(one,two) :- ( No Body ).

Trying to Establish the goals : less(two,three) , tspOil(two)
Substitutions: (XX028, two) (YY028, three)
Trying to Unify the Literals  : less(one,two) and less(two,three)
```

```
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,three) , tspOil(two)
Finding Clause starting with i= 15 ->Clause Found at i= 15

Clause - BEFORE Instantiation
less(two,three) :- ( No Body ).

Clause - AFTER Instantiation
less(two,three) :- ( No Body ).

Trying to Establish the goals : less(two,three) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : less(two,three) and less(two,three)
OK - Unification is Successfull

Goals : tspOil(two)
Looking for Clause with Predicate: tspOil
Finding Clause starting with i= 0 ->Clause Found at i= 9
Clause: tspOil(XX) :- haveTspOil(XX).

Clause - BEFORE Instantiation
tspOil(XX) :- haveTspOil(XX).

Clause - AFTER Instantiation
tspOil(XX032) :- haveTspOil(XX032).

Trying to Establish the goals : tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tspOil(XX032) and tspOil(two)
OK - Unification is Successfull

Goals : haveTspOil(two)
Looking for Clause with Predicate: haveTspOil
Finding Clause starting with i= 0 ->Clause Found at i= 25
Clause: haveTspOil(three) :- ( No Body ).

Clause - BEFORE Instantiation
haveTspOil(three) :- ( No Body ).

Clause - AFTER Instantiation
haveTspOil(three) :- ( No Body ).

Trying to Establish the goals : haveTspOil(two)
Substitutions: (XX032, two)
Trying to Unify the Literals  : haveTspOil(three) and haveTspOil(two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveTspOil(two)
Finding Clause starting with i= 26 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tspOil(two)
Finding Clause starting with i= 10 ->Clause Found at i= 10

Clause - BEFORE Instantiation
```

```
tspOil(XX) :- haveTspOil(YY) , less(XX,YY).

Clause - AFTER Instantiation
tspOil(XX034) :- haveTspOil(YY034) , less(XX034,YY034).

Trying to Establish the goals : tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tspOil(XX034) and tspOil(two)
OK - Unification is Successfull

Goals : haveTspOil(YY034) , less(two,YY034)
Looking for Clause with Predicate: haveTspOil
Finding Clause starting with i= 0 ->Clause Found at i= 25
Clause: haveTspOil(three) :- ( No Body ).

Clause - BEFORE Instantiation
haveTspOil(three) :- ( No Body ).

Clause - AFTER Instantiation
haveTspOil(three) :- ( No Body ).

Trying to Establish the goals : haveTspOil(YY034) , less(two,YY034)
Substitutions: (XX034, two)
Trying to Unify the Literals  : haveTspOil(three) and haveTspOil(YY034)
OK - Unification is Successfull

Goals : less(two,three)
Looking for Clause with Predicate: less
Finding Clause starting with i= 0 ->Clause Found at i= 14
Clause: less(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
less(one,two) :- ( No Body ).

Clause - AFTER Instantiation
less(one,two) :- ( No Body ).

Trying to Establish the goals : less(two,three)
Substitutions: (XX034, two) (YY034, three)
Trying to Unify the Literals  : less(one,two) and less(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,three)
Finding Clause starting with i= 15 ->Clause Found at i= 15

Clause - BEFORE Instantiation
less(two,three) :- ( No Body ).

Clause - AFTER Instantiation
less(two,three) :- ( No Body ).

Trying to Establish the goals : less(two,three)
Substitutions: No Substitutions

Trying to Unify the Literals  : less(two,three) and less(two,three)
OK - Unification is Successfull

*** SUCCESS - User's Query is Established ***
```

```
pasta ( XX ) :- cupFlour ( XX ) , egg ( XX ) , tblspWater ( YY ) , twice ( XX ,
YY ) , tspSalt ( XX ) , tspOil ( XX ) .
cupFlour ( XX ) :- haveCupFlour ( XX ).
cupFlour ( XX ) :- haveCupFlour ( YY ) , less ( XX , YY ).
cupMilk ( XX ) :- haveCupMilk ( XX ).
cupMilk ( XX ) :- haveCupMilk ( YY ) , less ( XX , YY ).
egg ( XX ) :- haveEgg ( XX ).
egg ( XX ) :- haveEgg ( YY ) , less ( XX , YY ).
tspSalt ( XX ) :- haveTspSalt ( XX ).
tspSalt ( XX ) :- haveTspSalt ( YY ) , less ( XX , YY ).
tspOil ( XX ) :- haveTspOil ( XX ).
tspOil ( XX ) :- haveTspOil ( YY ) , less ( XX , YY ).
twice ( one , two ).
twice ( two , four ).
twice ( three , six ).
less ( one , two ).
less ( two , three ).
less ( three , four ).
tblspWater ( one ).
tblspWater ( two ).
tblspWater ( three ).
tblspWater ( four ).
haveCupFlour ( three ).
haveEgg ( two ).
haveTspSalt ( four ).
haveTspOil ( three ).
```

```
*** Proceeding for Database creation ***
Input Clauses File: test4
Number of clauses: 25

Clause Errors = 0
Database is created Successfully

*** Proceeding to build the Query ***
Input Query File: qry
Query Errors = 0
User's Query : pasta(two)

*** Datalog Interpreter is Trying to Establish User's Query ***

Goals : pasta(two)
Looking for Clause with Predicate: pasta
Finding Clause starting with i= 0 ->Clause Found at i= 0
Clause: pasta(XX) :- cupFlour(XX) , egg(XX) , tblspWater(YY) , twice(XX,YY) , ts
pSalt(XX) , tspOil(XX).

Clause - BEFORE Instantiation
pasta(XX) :- cupFlour(XX) , egg(XX) , tblspWater(YY) , twice(XX,YY) , tspSalt(XX
) , tspOil(XX).

Clause - AFTER Instantiation
pasta(XX001) :- cupFlour(XX001) , egg(XX001) , tblspWater(YY001) , twice(XX001,Y
Y001) , tspSalt(XX001) , tspOil(XX001).

Trying to Establish the goals : pasta(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : pasta(XX001) and pasta(two)
OK - Unification is Successfull

Goals : cupFlour(two) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspSal
t(two) , tspOil(two)
Looking for Clause with Predicate: cupFlour
Finding Clause starting with i= 0 ->Clause Found at i= 1
Clause: cupFlour(XX) :- haveCupFlour(XX).

Clause - BEFORE Instantiation
cupFlour(XX) :- haveCupFlour(XX).

Clause - AFTER Instantiation
cupFlour(XX002) :- haveCupFlour(XX002).

Trying to Establish the goals : cupFlour(two) , egg(two) , tblspWater(YY001) , t
wice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: (XX001, two)
Trying to Unify the Literals  : cupFlour(XX002) and cupFlour(two)
OK - Unification is Successfull

Goals : haveCupFlour(two) , egg(two) , tblspWater(YY001) , twice(two,YY001) , ts
pSalt(two) , tspOil(two)
Looking for Clause with Predicate: haveCupFlour
Finding Clause starting with i= 0 ->Clause Found at i= 21
Clause: haveCupFlour(three) :- ( No Body ).

Clause - BEFORE Instantiation
haveCupFlour(three) :- ( No Body ).
```

```
Clause - AFTER Instantiation
haveCupFlour(three) :- ( No Body ).

Trying to Establish the goals : haveCupFlour(two) , egg(two) , tblspWater(YY001)
 , twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: (XX001, two) (XX002, two)
Trying to Unify the Literals  : haveCupFlour(three) and haveCupFlour(two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveCupFlour(two) , egg(two) , tblspWater(YY001) , twice(two,YY001) , ts
pSalt(two) , tspOil(two)
Finding Clause starting with i= 22 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : cupFlour(two) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspSal
t(two) , tspOil(two)
Finding Clause starting with i= 2 ->Clause Found at i= 2

Clause - BEFORE Instantiation
cupFlour(XX) :- haveCupFlour(YY) , less(XX,YY).

Clause - AFTER Instantiation
cupFlour(XX004) :- haveCupFlour(YY004) , less(XX004,YY004).

Trying to Establish the goals : cupFlour(two) , egg(two) , tblspWater(YY001) , t
wice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : cupFlour(XX004) and cupFlour(two)
OK - Unification is Successfull

Goals : haveCupFlour(YY004) , less(two,YY004) , egg(two) , tblspWater(YY001) , t
wice(two,YY001) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: haveCupFlour
Finding Clause starting with i= 0 ->Clause Found at i= 21
Clause: haveCupFlour(three) :- ( No Body ).

Clause - BEFORE Instantiation
haveCupFlour(three) :- ( No Body ).

Clause - AFTER Instantiation
haveCupFlour(three) :- ( No Body ).

Trying to Establish the goals : haveCupFlour(YY004) , less(two,YY004) , egg(two)
 , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: (XX004, two)
Trying to Unify the Literals  : haveCupFlour(three) and haveCupFlour(YY004)
OK - Unification is Successfull

Goals : less(two,three) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspS
alt(two) , tspOil(two)
Looking for Clause with Predicate: less
Finding Clause starting with i= 0 ->Clause Found at i= 14
Clause: less(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
less(one,two) :- ( No Body ).
```

```
Clause - AFTER Instantiation
less(one,two) :- ( No Body ).

Trying to Establish the goals : less(two,three) , egg(two) , tblspWater(YY001) ,
 twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: (XX004, two) (YY004, three)
Trying to Unify the Literals  : less(one,two) and less(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,three) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspS
alt(two) , tspOil(two)
Finding Clause starting with i= 15 ->Clause Found at i= 15

Clause - BEFORE Instantiation
less(two,three) :- ( No Body ).

Clause - AFTER Instantiation
less(two,three) :- ( No Body ).

Trying to Establish the goals : less(two,three) , egg(two) , tblspWater(YY001) ,
 twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : less(two,three) and less(two,three)
OK - Unification is Successfull

Goals : egg(two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(
two)
Looking for Clause with Predicate: egg
Finding Clause starting with i= 0 ->Clause Found at i= 5
Clause: egg(XX) :- haveEgg(XX).

Clause - BEFORE Instantiation
egg(XX) :- haveEgg(XX).

Clause - AFTER Instantiation
egg(XX008) :- haveEgg(XX008).

Trying to Establish the goals : egg(two) , tblspWater(YY001) , twice(two,YY001)
, tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : egg(XX008) and egg(two)
OK - Unification is Successfull

Goals : haveEgg(two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tsp
Oil(two)
Looking for Clause with Predicate: haveEgg
Finding Clause starting with i= 0 ->Clause Found at i= 22
Clause: haveEgg(two) :- ( No Body ).

Clause - BEFORE Instantiation
haveEgg(two) :- ( No Body ).

Clause - AFTER Instantiation
haveEgg(two) :- ( No Body ).

Trying to Establish the goals : haveEgg(two) , tblspWater(YY001) , twice(two,YY0
01) , tspSalt(two) , tspOil(two)
```

```
Substitutions: (XX008, two)
Trying to Unify the Literals  : haveEgg(two) and haveEgg(two)
OK - Unification is Successfull

Goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: tblspWater
Finding Clause starting with i= 0 ->Clause Found at i= 17
Clause: tblspWater(one) :- ( No Body ).

Clause - BEFORE Instantiation
tblspWater(one) :- ( No Body ).

Clause - AFTER Instantiation
tblspWater(one) :- ( No Body ).

Trying to Establish the goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(t
wo) , tspOil(two)
Substitutions: (XX008, two)
Trying to Unify the Literals  : tblspWater(one) and tblspWater(YY001)
OK - Unification is Successfull

Goals : twice(two,one) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: twice
Finding Clause starting with i= 0 ->Clause Found at i= 11
Clause: twice(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
twice(one,two) :- ( No Body ).

Clause - AFTER Instantiation
twice(one,two) :- ( No Body ).

Trying to Establish the goals : twice(two,one) , tspSalt(two) , tspOil(two)
Substitutions: (XX008, two) (YY001, one)
Trying to Unify the Literals  : twice(one,two) and twice(two,one)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,one) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 12 ->Clause Found at i= 12

Clause - BEFORE Instantiation
twice(two,four) :- ( No Body ).

Clause - AFTER Instantiation
twice(two,four) :- ( No Body ).

Trying to Establish the goals : twice(two,one) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(two,four) and twice(two,one)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,one) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 13 ->Clause Found at i= 13

Clause - BEFORE Instantiation
twice(three,six) :- ( No Body ).
```

```
Clause - AFTER Instantiation
twice(three,six) :- ( No Body ).

Trying to Establish the goals : twice(two,one) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(three,six) and twice(two,one)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,one) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 14 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 18 ->Clause Found at i= 18

Clause - BEFORE Instantiation
tblspWater(two) :- ( No Body ).

Clause - AFTER Instantiation
tblspWater(two) :- ( No Body ).

Trying to Establish the goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(t
wo) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tblspWater(two) and tblspWater(YY001)
OK - Unification is Successfull

Goals : twice(two,two) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: twice
Finding Clause starting with i= 0 ->Clause Found at i= 11
Clause: twice(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
twice(one,two) :- ( No Body ).

Clause - AFTER Instantiation
twice(one,two) :- ( No Body ).

Trying to Establish the goals : twice(two,two) , tspSalt(two) , tspOil(two)
Substitutions: (YY001, two)
Trying to Unify the Literals  : twice(one,two) and twice(two,two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,two) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 12 ->Clause Found at i= 12

Clause - BEFORE Instantiation
twice(two,four) :- ( No Body ).

Clause - AFTER Instantiation
twice(two,four) :- ( No Body ).

Trying to Establish the goals : twice(two,two) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(two,four) and twice(two,two)
```

```
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,two) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 13 ->Clause Found at i= 13

Clause - BEFORE Instantiation
twice(three,six) :- ( No Body ).

Clause - AFTER Instantiation
twice(three,six) :- ( No Body ).

Trying to Establish the goals : twice(two,two) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(three,six) and twice(two,two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,two) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 14 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 19 ->Clause Found at i= 19

Clause - BEFORE Instantiation
tblspWater(three) :- ( No Body ).

Clause - AFTER Instantiation
tblspWater(three) :- ( No Body ).

Trying to Establish the goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(t
wo) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tblspWater(three) and tblspWater(YY001)
OK - Unification is Successfull

Goals : twice(two,three) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: twice
Finding Clause starting with i= 0 ->Clause Found at i= 11
Clause: twice(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
twice(one,two) :- ( No Body ).

Clause - AFTER Instantiation
twice(one,two) :- ( No Body ).

Trying to Establish the goals : twice(two,three) , tspSalt(two) , tspOil(two)
Substitutions: (YY001, three)
Trying to Unify the Literals  : twice(one,two) and twice(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,three) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 12 ->Clause Found at i= 12

Clause - BEFORE Instantiation
```

```
twice(two,four) :- ( No Body ).

Clause - AFTER Instantiation
twice(two,four) :- ( No Body ).

Trying to Establish the goals : twice(two,three) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(two,four) and twice(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,three) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 13 ->Clause Found at i= 13

Clause - BEFORE Instantiation
twice(three,six) :- ( No Body ).

Clause - AFTER Instantiation
twice(three,six) :- ( No Body ).

Trying to Establish the goals : twice(two,three) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(three,six) and twice(two,three)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,three) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 14 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 20 ->Clause Found at i= 20

Clause - BEFORE Instantiation
tblspWater(four) :- ( No Body ).

Clause - AFTER Instantiation
tblspWater(four) :- ( No Body ).

Trying to Establish the goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(t
wo) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tblspWater(four) and tblspWater(YY001)
OK - Unification is Successfull

Goals : twice(two,four) , tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: twice
Finding Clause starting with i= 0 ->Clause Found at i= 11
Clause: twice(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
twice(one,two) :- ( No Body ).

Clause - AFTER Instantiation
twice(one,two) :- ( No Body ).

Trying to Establish the goals : twice(two,four) , tspSalt(two) , tspOil(two)
```

```
Substitutions: (YY001, four)
Trying to Unify the Literals  : twice(one,two) and twice(two,four)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,four) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 12 ->Clause Found at i= 12

Clause - BEFORE Instantiation
twice(two,four) :- ( No Body ).

Clause - AFTER Instantiation
twice(two,four) :- ( No Body ).

Trying to Establish the goals : twice(two,four) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(two,four) and twice(two,four)
OK - Unification is Successfull

Goals : tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: tspSalt
Finding Clause starting with i= 0 ->Clause Found at i= 7
Clause: tspSalt(XX) :- haveTspSalt(XX).

Clause - BEFORE Instantiation
tspSalt(XX) :- haveTspSalt(XX).

Clause - AFTER Instantiation
tspSalt(XX025) :- haveTspSalt(XX025).

Trying to Establish the goals : tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tspSalt(XX025) and tspSalt(two)
OK - Unification is Successfull

Goals : haveTspSalt(two) , tspOil(two)
Looking for Clause with Predicate: haveTspSalt
Finding Clause starting with i= 0 ->Clause Found at i= 23
Clause: haveTspSalt(four) :- ( No Body ).

Clause - BEFORE Instantiation
haveTspSalt(four) :- ( No Body ).

Clause - AFTER Instantiation
haveTspSalt(four) :- ( No Body ).

Trying to Establish the goals : haveTspSalt(two) , tspOil(two)
Substitutions: (XX025, two)
Trying to Unify the Literals  : haveTspSalt(four) and haveTspSalt(two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveTspSalt(two) , tspOil(two)
Finding Clause starting with i= 24 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tspSalt(two) , tspOil(two)
Finding Clause starting with i= 8 ->Clause Found at i= 8
```

```
Clause - BEFORE Instantiation
tspSalt(XX) :- haveTspSalt(YY) , less(XX,YY).

Clause - AFTER Instantiation
tspSalt(XX027) :- haveTspSalt(YY027) , less(XX027,YY027).

Trying to Establish the goals : tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : tspSalt(XX027) and tspSalt(two)
OK - Unification is Successfull

Goals : haveTspSalt(YY027) , less(two,YY027) , tspOil(two)
Looking for Clause with Predicate: haveTspSalt
Finding Clause starting with i= 0 ->Clause Found at i= 23
Clause: haveTspSalt(four) :- ( No Body ).

Clause - BEFORE Instantiation
haveTspSalt(four) :- ( No Body ).

Clause - AFTER Instantiation
haveTspSalt(four) :- ( No Body ).

Trying to Establish the goals : haveTspSalt(YY027) , less(two,YY027) , tspOil(tw
o)
Substitutions: (XX027, two)
Trying to Unify the Literals  : haveTspSalt(four) and haveTspSalt(YY027)
OK - Unification is Successfull

Goals : less(two,four) , tspOil(two)
Looking for Clause with Predicate: less
Finding Clause starting with i= 0 ->Clause Found at i= 14
Clause: less(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
less(one,two) :- ( No Body ).

Clause - AFTER Instantiation
less(one,two) :- ( No Body ).

Trying to Establish the goals : less(two,four) , tspOil(two)
Substitutions: (XX027, two) (YY027, four)
Trying to Unify the Literals  : less(one,two) and less(two,four)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,four) , tspOil(two)
Finding Clause starting with i= 15 ->Clause Found at i= 15

Clause - BEFORE Instantiation
less(two,three) :- ( No Body ).

Clause - AFTER Instantiation
less(two,three) :- ( No Body ).

Trying to Establish the goals : less(two,four) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : less(two,three) and less(two,four)
```

```
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,four) , tspOil(two)
Finding Clause starting with i= 16 ->Clause Found at i= 16

Clause - BEFORE Instantiation
less(three,four) :- ( No Body ).

Clause - AFTER Instantiation
less(three,four) :- ( No Body ).

Trying to Establish the goals : less(two,four) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : less(three,four) and less(two,four)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,four) , tspOil(two)
Finding Clause starting with i= 17 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveTspSalt(YY027) , less(two,YY027) , tspOil(two)
Finding Clause starting with i= 24 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tspSalt(two) , tspOil(two)
Finding Clause starting with i= 9 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,four) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 13 ->Clause Found at i= 13

Clause - BEFORE Instantiation
twice(three,six) :- ( No Body ).

Clause - AFTER Instantiation
twice(three,six) :- ( No Body ).

Trying to Establish the goals : twice(two,four) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : twice(three,six) and twice(two,four)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : twice(two,four) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 14 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 21 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveEgg(two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tsp
Oil(two)
Finding Clause starting with i= 23 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
```

```
Goals : egg(two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(
two)
Finding Clause starting with i= 6 ->Clause Found at i= 6

Clause - BEFORE Instantiation
egg(XX) :- haveEgg(YY) , less(XX,YY).

Clause - AFTER Instantiation
egg(XX033) :- haveEgg(YY033) , less(XX033,YY033).

Trying to Establish the goals : egg(two) , tblspWater(YY001) , twice(two,YY001)
, tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : egg(XX033) and egg(two)
OK - Unification is Successfull

Goals : haveEgg(YY033) , less(two,YY033) , tblspWater(YY001) , twice(two,YY001)
, tspSalt(two) , tspOil(two)
Looking for Clause with Predicate: haveEgg
Finding Clause starting with i= 0 ->Clause Found at i= 22
Clause: haveEgg(two) :- ( No Body ).

Clause - BEFORE Instantiation
haveEgg(two) :- ( No Body ).

Clause - AFTER Instantiation
haveEgg(two) :- ( No Body ).

Trying to Establish the goals : haveEgg(YY033) , less(two,YY033) , tblspWater(YY
001) , twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: (XX033, two)
Trying to Unify the Literals  : haveEgg(two) and haveEgg(YY033)
OK - Unification is Successfull

Goals : less(two,two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , ts
pOil(two)
Looking for Clause with Predicate: less
Finding Clause starting with i= 0 ->Clause Found at i= 14
Clause: less(one,two) :- ( No Body ).

Clause - BEFORE Instantiation
less(one,two) :- ( No Body ).

Clause - AFTER Instantiation
less(one,two) :- ( No Body ).

Trying to Establish the goals : less(two,two) , tblspWater(YY001) , twice(two,YY
001) , tspSalt(two) , tspOil(two)
Substitutions: (XX033, two) (YY033, two)
Trying to Unify the Literals  : less(one,two) and less(two,two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , ts
pOil(two)
Finding Clause starting with i= 15 ->Clause Found at i= 15

Clause - BEFORE Instantiation
less(two,three) :- ( No Body ).
```

```
Clause - AFTER Instantiation
less(two,three) :- ( No Body ).

Trying to Establish the goals : less(two,two) , tblspWater(YY001) , twice(two,YY
001) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : less(two,three) and less(two,two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , ts
pOil(two)
Finding Clause starting with i= 16 ->Clause Found at i= 16

Clause - BEFORE Instantiation
less(three,four) :- ( No Body ).

Clause - AFTER Instantiation
less(three,four) :- ( No Body ).

Trying to Establish the goals : less(two,two) , tblspWater(YY001) , twice(two,YY
001) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : less(three,four) and less(two,two)
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , ts
pOil(two)
Finding Clause starting with i= 17 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveEgg(YY033) , less(two,YY033) , tblspWater(YY001) , twice(two,YY001)
, tspSalt(two) , tspOil(two)
Finding Clause starting with i= 23 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : egg(two) , tblspWater(YY001) , twice(two,YY001) , tspSalt(two) , tspOil(
two)
Finding Clause starting with i= 7 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,three) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspS
alt(two) , tspOil(two)
Finding Clause starting with i= 16 ->Clause Found at i= 16

Clause - BEFORE Instantiation
less(three,four) :- ( No Body ).

Clause - AFTER Instantiation
less(three,four) :- ( No Body ).

Trying to Establish the goals : less(two,three) , egg(two) , tblspWater(YY001) ,
 twice(two,YY001) , tspSalt(two) , tspOil(two)
Substitutions: No Substitutions

Trying to Unify the Literals  : less(three,four) and less(two,three)
```

```
Unification Failed : Constants not same

*** Unable to Establish the goals - Back Tracking to ->
Goals : less(two,three) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspS
alt(two) , tspOil(two)
Finding Clause starting with i= 17 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : haveCupFlour(YY004) , less(two,YY004) , egg(two) , tblspWater(YY001) , t
wice(two,YY001) , tspSalt(two) , tspOil(two)
Finding Clause starting with i= 22 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : cupFlour(two) , egg(two) , tblspWater(YY001) , twice(two,YY001) , tspSal
t(two) , tspOil(two)
Finding Clause starting with i= 3 ->Clause Not found

*** Unable to Establish the goals - Back Tracking to ->
Goals : pasta(two)
Finding Clause starting with i= 1 ->Clause Not found

*** FAILURE - User's Query CAN'T BE Established ***
```