A Natural Language User Interface to Database Systems

by

Chunsheng Yi

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 1996

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

ISBN 0-612-16379-2

Canada

Name _____

Dissertation Abstracts International and *Masters Abstracts International* are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

Computer Science
SUBJECT TERM

0984
SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology ... 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology .. 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics ... 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

A NATURAL LANGUAGE USER INTERFACE TO DATABASE SYSTEMS

BY

CHUNSHENG YI

A Thesis/Practicum submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Natural language user interfaces to database systems(NLIDB) attempt to formulate queries to database systems in user's native language. This thesis presents a NLIDB which uses *abductive reasoning* to understand natural language (e.g. English) and translates queries into the formal Structural Query Language(SQL).

Abduction is an inference to achieve the best explanation. The task of understanding natural language is viewed as finding the structural relationships between unstructured inputs. That is, to understand is to seek the best explanation of how the inputs are coherently related. *Obvious abduction* uses a network to represent the domain knowledge. Observations correspond to nodes in the network annotated with a set of attribute-value pairs. An explanation of the observations is a generalized subtree of the network that connects all the observations. This connection is a coherent set of relationships between the observations, and therefore, explains how they are related. To understand queries to a database, we use Entity-Relationship(E-R) diagram of the database as domain knowledge. When querying a database, the user need only list the interested items. These items are mapped onto the nodes of the E-R diagram. The access paths are obtained by finding a connection among these nodes in the E-R diagram. We choose the shortest connection, which has the minimum total length, if there are several alternative connections. The shortest connections, or database *query graphs* are translated to SQL statements based on the mapping between E-R diagram and database schema. Those SQL statements can be submitted to SQL supported database management systems.

Acknowledgements

I am deeply grateful to my supervisor, Dr. Dekang Lin, for his immense support and encouragement during this research. He was easy to work with because he gave me the freedom to explore my own ideas, reviewed my ideas thoroughly and critically, and he recognized meaningful contributions with compliments. His motivating style and clear directions have helped me to develop my confidence and abilities in research and writing.

Thanks to all my classmates at the University of Manitoba for all of the hot and interesting discussions that contributed to the quality of this thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Database system and *user interface* are the two key components of many computer applications. In order to access the computer applications, the user must either rely on his/her programming skills, or the modes of the interaction provided by the interface designer. While the databases grow larger and the user population increases, Data Base Management Systems(DBMS) are still built for retrieval efficiency and not user effectiveness[10]. There are several kinds of database interfaces: Structural Query Language(SQL), form based interfaces, Graphical User Interfaces(GUI), and Natural Language Interfaces(NLI). Compared with other interfaces, NLIs provide the capability for a wide audience of computer neophytes to access computer databases but do not want to find the time to learn all the intricate details.

This thesis presents a NLI to database systems(NLIDB) which uses *abductive reasoning* to understand queries in natural language(e.g. English) and translates queries into SQL. This chapter reviews the concepts of database system and the

key issues to understand queries by abductions. At the end of this chapter, two simple database schemas are given which are the underline databases through this thesis.

1.2 Database and Relational Database

Databases are large collections of structured data stored in one or several computers in order to serve several data processing applications. Databases model some parts of the real world using a series of models. A model represents a set of concepts that describe information pertinent to these applications.

According to the data model on which the database is based, database systems can be categorized as *relational, network, hierarchical*, and *object-oriented.* The Entity-Relationship model, a high-level conceptual data model, is often used in database design. The *network* model, also known as the CODASYL DBTG model, represents data as *record types* and *set types.* Each set type defines a 1:N relationship between an owner record and a member record. A record type can participate as owner or member in any number of set types. A network model uses data definition language(DDL) to define record types and set types. Also, the model uses data manipulation language(DML) to navigate, retrieve, and update database. The *hierarchical* model represents data as hierarchical tree structures. Each hierarchy represents a number of related records. The main structures used by the hierarchical model are record types and parent-child relationship(PCR) types. Each PCR type defines a hierarchical 1:N relationship between a parent record type and a child record type. Relationships are strictly hierarchical in that a record type can participate as child in at most one PCR type. This restriction makes it difficult to represent a world where numerous relationships exist. Similar to the network

model, the hierarchical model needs separate DDL and DML to define and manipulate the databases. The *object-oriented* model defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a *class*, and classes are organized into hierarchies or acyclic graphs. The operations of each class are specified in terms of predefined procedures called *methods*.

The relational model of data, first proposed by Codd[7], has been widely accepted mainly because of its conceptual simplicity, symmetry, and data independence. The basic idea of relational model is to consider a database as a collection of *relations*. A common representation of a given relation is a *table* where the columns represent the domains, called *attribute*, and the rows represent the elements in the relation, called *tuple*. In addition, the relational DBMS has been extending their models to incorporate object-oriented concepts and other capabilities; they are referred to as *extended relational systems*.

The relational algebra, which consists of a set of operators, is used to manipulate relations. The operators are used to select tuples from individual relations and to combine related tuples from several relations for the purpose of answering a query. The result of each operation defines a new relation, so relational algebra can then be used to manipulate and query the database in a homogeneous manner. Those most important operations are PROJECT, SELECT and JOIN. SELECT chooses tuples and PROJECT selects columns or attributes. JOIN constructs a new relation by taking tuples from relation a R and the corresponding tuples from relation a S. The correspondence is expressed by comparison between common attributes. This operation is often necessary when the answer to the query must be obtained from more than one table.The relational algebra itself, however, suffers from the disadvantage of being too procedural. That is, the users need to specify not only

what but also *how* (i.e. in what order). Mathematical maturity is demanded to master relational algebra.

High-level query languages are proposed to alleviate the problem. The basic operations in these languages are usually composition of operators in relational algebra or calculus. The two best known of these languages are SQL and QUEL.

1.2.1 SQL

SQL is a relational query language developed at IBM as the interface for an experimental relational database system called SYSTEM R. It is neither a pure algebraic language nor a predicated language. It has emerged as the most commonly used relational language, and is now used in a large number of products. A joint effort under way by ANSI(the American National Standards Institute) and ISO(the International Standards Organization) has led to a series of standard versions of SQL[10].

SQL is a comprehensive database language; it has statements for data definition, query, and update. Hence, it is both a DDL and a DML. In addition, it has facilities for defining views on the database, for embedding SQL statements into a general-purpose programming language such as C or PASCAL.

The basic SQL statement is the qualification block which gives as its result a new relation:

select <attribute list>
from <relation(s)>
where <condition(s)>

The JOIN operation can be expressed by either nesting the qualification blocks or by putting several relations in the FROM clause. For example, the query:

what are the costs of blue parts? (A)

can be express in SQL as follows:

```
select   cost, part
from     part, supply
where    part.color="blue" and part.part=supply.part
```

A *view* in SQL terminology is a single table that is usually derived from other tables. Those tables could be base tables or previously defined views. A view does not necessarily exist in physical form. It is considered a *virtual table*, in contrast to base tables whose tuples are actually stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view. We can also think of a view as a way of specifying a table referenced frequently, even though it may not exist physically.

1.2.2 QUEL

QUEL is the query language of INGRES, a relational DBMS developed at the University of California, Berkeley. QUEL can be used as an interactive query language or it can be embedded within a host programming language. It includes a range of functionality similar to that provided by early SQL versions. Basic QUEL retrieval queries of the select-project-join type are very similar to tuple relational calculus. Two clause, RETRIEVE and WHERE, are used; RETRIEVE specifies the attributes to be retrieved (the projection attributes) and WHERE specifies the select and join conditions. QUEL does not have a FROM clause as SQL does; rather, all attributes in a QUEL query must be explicitly qualified, either by their relation name or by a tuple variable declared to range over their relation. Tuple

variables are declared explicitly in the RANGE statement of QUEL. Here is the generic query format in QUEL:

range of t_1 is R_1
$\vdots$
range of t_k is R_k
retrieve($t_{i_1}.A_1, \ldots, t_{i_r}.A_r$) where ϕ

In the above, A_m is an attribute of relation R_{i_m}, for m=1,2, ..., r, and ϕ is a tuple calculus formula with no quantifiers, which specifies the joining conditions. The query (A) can be expressed in QUEL as follows:

range of p is part
range of s is supply
retrieve(p.part, s.cost) where p.color="blue" and p.part=s.part

1.3 The Navigation Problem

In database, the term *navigation* originally refers to the process of following the physical access links in network or hierarchical databases[33]. Users of relational database are relieved from physical navigation. However, they still have to perform logical navigation. That is, the users are responsible for specifying which tables to access and the join conditions. Although this is substantially simpler than following the physical links, it has turned out to be a major difficulty in using database. To perform the logical navigation, the user must not only be familiar with the concepts in relational database in general and the structure of that particular database, but also remember the table names and attribute names, which are, unfortunately, often cryptic abbreviations.

Controlled experimental research has shown that queries requiring linking between relations or complex traversal of hierarchies or networks substantially increase the probability of user errors [32, 30, 4]. In the relational model linkage between relations requires a join or specification of linking domains which is confusing and tedious step.

One way to relieve the user of the logical navigation is to provide *logical views* of the database which are different from the actual database. Since the user can use only the predefined relations that belong to his/her logical view, he/she must have available a large set of predefined logical views if he/she is to access the data in a truly flexible manner.

1.4 Steiner Trees

Several efforts have been made to relieve the user of the responsibility for logical navigation. The approach taken by Wald[34] and Motro[28] employs a graph to represent the database conceptual schema. The graph is similar to various conceptual modeling graphs used in database designs, such as E-R diagrams. We refer to this graph as **the database graph**. When querying a database, the user need only list the items of interest. These items are mapped onto the nodes of the database graph. The access paths are obtained by finding a connection between these nodes in the graph. Since, in general, several alternative connections may be possible, the shortest connection, which has the minimum total length, is used to resolve the ambiguity.

The graph-theoretic definition of the problem of finding the shortest connection between a given set of nodes in a graph is known as the Steiner Problem in Graphs and the shortest connection, which must obviously be a tree, is called the Steiner

Tree. We use the terms *the Steiner Tree* and *the shortest connection* interchangeably in this thesis.

Unfortunately, the Steiner Problem in Graphs is NP-Complete [11]. Therefore, a key issue in this approach is how to cope with this complexity. Wald handled this problem by restricting the database graph to a special kind: the partial 2-trees[1]. An algorithm was then introduced, which can solve Steiner Problems in partial 2-trees in linear time. Figure 1.1 shows some 2-tree examples. This method is not applicable when the database graph is not a partial 2-tree.

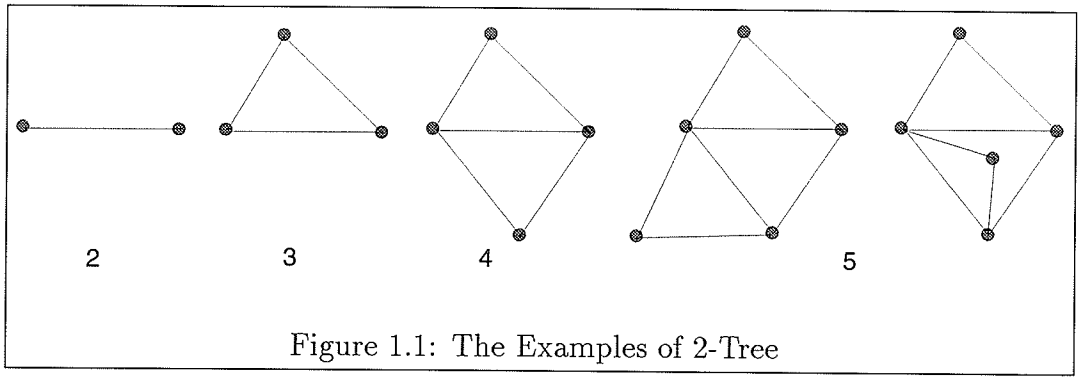

Figure 1.1: The Examples of 2-Tree

Motro [28] on the other hand, limited the search to the connections where the nodes to be connected are at most 2 or 3 links away from each other. The drawback of the algorithm is its *brittleness* due to the constant limit.

The abduction algorithm in [22] is a generalization of a message passing algorithm and can be adopted for the Steiner Problem.

[1]A 2-tree can be defined recursively as follows: an edge <x,y> is a 2-tree; given an edge <x,y> of a 2-tree, adding a new node z adjacent to x and y yields a 2-tree. A partial 2-tree is a spanning subgraph of 2-tree.

1.5 Obvious Abduction

Obvious abduction, detailed in Chapter 3, uses a network to represent the domain knowledge. Observations correspond to nodes in the network annotated with a set of attribute-value pairs. An explanation of the observations is a generalized subtree of the network that connects all the observations. This connection is a coherent set of relationships between the observations, and therefore, explains how they are related.

To find the Steiner Trees, we employ E-R diagram as the domain knowledge. The given interested nodes in E-R diagram are observations; the explanations are the the connections of those nodes; the best explanations are the shortest connections of those nodes, namely Steiner Trees.

Although the database graph tends to contain hundreds or even thousands of nodes, the nodes explicitly mentioned in a query tend be close to each other, since they are semantically related. The distributed algorithm is particularly well suited for this application because it is able to exploit this locality. Most of the nodes that are far away from the nodes to be connected will not be involved in the computation at all.

1.6 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 reviews several previous approaches to database interface, including formal query language, form-based interface, graphical user interface, and natural language user interface. Chapter 3 presents the overview of obvious abduction, and framework of the system architecture of the system. Chapter 4 gives more details of the system implementation.

It is also concerned with the internal representation of domain knowledge network and a SQL translator which transforms the found Steiner Trees to SQL statements. Chapter 5 will summarize the overall contribution and suggest further research.

1.7 Example Databases

For the purposes of this thesis, we will take the examples from two databases: one is called the "company business database", the other is the "employee database". The semantic interpretation of the databases is fairly self-explanatory. The "company database" has the information on what products a customer has ordered, what parts a product uses, and what parts a supplier provides. The "employee database" stores the information of employee, such as an employee's personal data, the project he/she works on, and so on.

The table **company** contains information about the companies who do business with this company (the owner of the database). The **customer** and **supplier** are subclasses of **company**. The suppliers are the companies from whom this company buys parts. The customers are the companies to whom this company sells products. A company may be both a customer and a supplier. The table **order** contains the incoming orders of products sent to the customers. The table **supply** contains the outgoing orders of parts from the suppliers. The table **product** contains all the products. The table **part** contains all the parts. The table **use** specifies which part is used in which product.

The schema of the "business" database are as follows (the fields in bold font are key fields):

company(**cno**, name, address, phone);

customer(**cno**, credit);

supplier(**cno**, reliability);

product(**pno**, name, cost);

part(**part_no**, name, color, description);

order(**cno, pno**, quantity);

supply(**part_no, cno**, cost);

use(**part_no, pno**);

The schema of the "employee" database are as follows:

employee(**SIN**, Name, Bdate, Sex, Address, Salary, supervision, dno);

department(**dnumber**, name, mgrsin, mgrstartdate);

project(**pnumber**, name, dnum);

works_on(**SIN, pnumber**, hours);

manage(**SIN, dnumber**);

works_for(**SIN, dnumber**);

Chapter 2

Previous Approaches to Database Interfaces

This chapter reviews several previous approaches to database interfaces. In 1.2.1 and 1.2.2 we have discussed the two most widely used *formal query languages*. The other interfaces are *form-based* interface, *graphical user interface* (GUI) and *natural language interface* (NLI).

2.1 Form-based Interface

In form-based interfaces, pre-defined forms consisting of fields are used. The user fills the known fields, and the system completes the remaining fields by querying the underlying database. A user wishing to learn the manager of an employee named "Smith" could fill in the pre-defined form "**Employee Information Form**" as shown in Figure 2.1 on the left. The system should respond by filling the remaining fields as shown in Figure 2.1 on the right.

Employee Information Form			
Employee:	Smith	Employee	Smith
Department:		Department	sales
Phone:		Phone	1516
Manager:		Manager	Baker

Figure 2.1: An Example of the Pre-defined Form

EMPLOYEE	Name	Ssn	Bdata	Address	Sex	Salary	SuperSsn	Dno

Figure 2.2: Table skeleton

A more powerful method is called *Query By Example*(QBE)[10]. QBE is a language developed at IBM Research. It is intended to provide a convenient and unified way to query, update, define and control a relational database. The general level of the language resembles that of SQL but has been designed for use with a visual display terminal. It has a two-dimensional syntax. Each operation in QBE is specified using one or more tables; each such table is built up on the display screen, with some parts being supplied by the system and others by the user.

To query the database using QBE the user first select one or more relations by filling in the relation names in the table skeletons. Each of the table skeleton is a graphical representation of a relation, as shown in Figure 2.2. The user does not have to remember the names of attributes or relations, because they are displayed as part of these skeletons. The user then fills in the blanks with domain variables and attributes, using the screen editor. The variables are the same as used in relational calculus except that they are usually typical elements in the domain preceded by underscores. The prefix "P." is used to indicate that the values of a

EMPLOYEE	Name	Ssn	Bdata	Address	Sex	Salary	SuperSsn	Dno
	_Na			_Addr				_Dn

DEPARTMENT	DName	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	_Dn		

RESULTS		
P.	_Na	_Addr

Figure 2.3: An example query in QBE

particular column are to be retrieved, where P stands for Print. Figure 2.3 shows what a typical QBE query looks like. The query is "List the name and address of all employees who work for the 'Research' department".

This approach is appealing in that forms are a common communication medium. Paper forms have been used extensively in offices. Most people can easily fill out forms and can also design forms of their own. Greenblatt and Waxman [13] compared the ease of using QBE, SEQUEL (SQL), and relational algebra. The study favors QBE over the other two. However, as far as logical navigation is concerned, users of QBE are still responsible for specifying the access paths. The graphic tools facilitate browsing the relations and attributes, and make the navigation easier.

2.2 Graphical User Interface

Graphical user interfaces provide visual and direct operations on tables and relations. The interface is intuitive and easy to use. End-users do not have to under-

stand SQL in order to be productive. The user first selects the database tables to be used in the query. Each selected table appears on the screen as a frame consisting of attribute-slots. The user can then fill in attribute slots, and join attributes across the various frames. Figure 2.4 shows a GUI in GQL(Graphical Query Language) [9]. As in GQL, the GUI adopts a click-and-drag style. By clicking, a user chooses the data objects corresponding to the table he/she wishes to retrieve, specifies the information he/she wishes to retrieve from that table. By dialog box he/she refines the queries by adding join conditions, qualifications, or restrictions. In Figure 2.4, the bottom frame shows how to define join condition; the right frame shows how to choose interested attributes; and the top frame shows the overall tables and their relationships.

2.3 Natural Language Front-end

Natural Language Processing(NLP) has long been a major research goal in Artificial Intelligence. The Natural Language Interface(NLI) is one of the most important applications of natural language processing. The advantage of natural language interface is obvious. A sophisticated natural language system would require little or no learning on the part of the user. He or she would simply sit down at a terminal and "ask" the database various questions.

Natural language interfaces to database usually consist of a natural language component and a database access component. Questions entered in natural language would be translated into a statement in formal query language. Once the statement is unambiguously formulated, the query would be processed by the database management system to produce the required data. Figure 2.5 shows the basic components of a typical natural language interface. The bottom box is the database

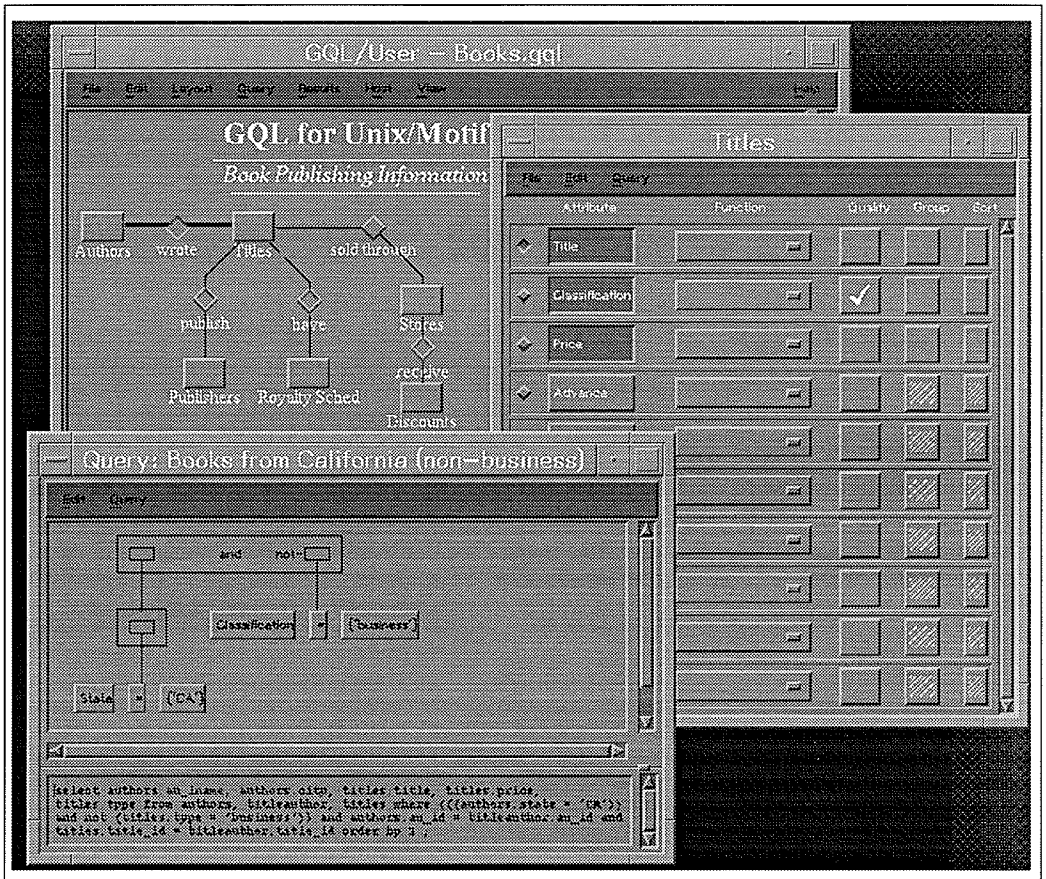

Figure 2.4: An example of GUI

access component; the top box is the natural language component. In the top box, there are three sub-components: understanding component, generation component (most system takes database output directly), and a knowledge base.

The approaches to natural language interfaces can roughly be categorized as follows:

Language through multiple choices In this approach, whenever there is ambiguity in user's input the system displays the available options to the user, who chooses from among those options. RENDEZVOUS [8] is a typical example. It

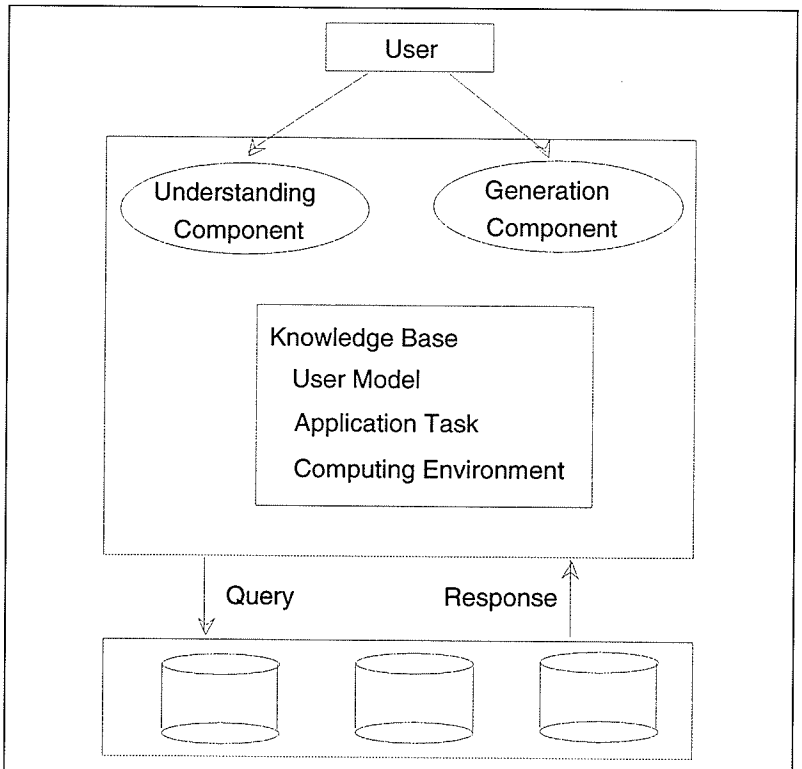

Figure 2.5: Components of a Natural Language Interface

engages the user in a "clarification dialogue" and gradually constructs the user's request. Such dialogues help to extend the linguistic coverage of a natural language system in that they allow the system to enhance its linguistic capabilities in consultation with the user. Unfortunately, they can turn a simple request (from the user's point of view) into a tedious and lengthy discussion with the system.

Pattern-matching Some of the early NLIs relied on pattern-matching techniques to answer the user's queries. A pattern-matching system usually has some rules, and actions under each rule. For example, assuming Figure 2.6 holds information about companies, we could have these rules:

Company	Address	Phone
IBM	111 One IBM Plaza, MA 01851	(900)123-4567
MicroSoft	981 One MicroSoft Way, WA 91851	(900)967-4321
...	...	...

Figure 2.6: An Table Example

pattern: ... "address" ... < company_name >
action: submit the SQL statement

```
select   address
from     company_table
where    company_table.name = <company_name>
```

pattern: ... "address" ... "company"
action: submit the SQL statement

```
select   company, address
from     company_table
```

The first rule says if a user's request contains the word "address" followed by a company's name, the system will locate the row which contains the company name.

For example, if user typed "what is the address of IBM?", the system would use the first rule. The same rule works for "Print the address of IBM", "Could you tell me the address of MicroSoft?", etc.

The second rule should handle all the cases that the user's request contains word "address" followed by word "company". For example, it will report all companies and their addresses for "what is the address of each company?", "List the addresses of all companies", etc.

The main advantage of the pattern-matching approach is its simplicity: no parsing and interpretation modules are needed, and the systems are easy to implement. However, the shallowness of the pattern-matching approach would often lead to

S	⇒	QUERY SHIP-PROPERTY of SHIP
QUERY	⇒	what is | tell me
SHIP-PROPERTY	⇒	the SHIP-PROP| SHIP-PROP
SHIP-PROP	⇒	speed | length | type
SHIP	⇒	the SHIP-NAME | the fastest SHIP2
SHIP-NAME	⇒	Kennedy | Kitty Hawk | Constellation | ...
SHIP2	⇒	COUNTRYS SHIP3 | SHIP3
SHIP3	⇒	SHIPTYPE LOCATION | SHIPTYPE
SHIPTYPE	⇒	carrier | submarine | ...
COUNTRYS	⇒	American | British | Russian | ...
LOCATION	⇒	in the Mediterranean | in the Atlantic | ...

Figure 2.7: Semantic grammar in LADDER system

bad failures. For example, if some words have different meanings in some contexts, the system will give strange answers.

Semantic grammars The above approach works when only a restricted number of statements makes sense to the system, but as the number of things that users may need to say increases, it becomes less and less practical.

A semantic grammar, like any other grammar, consists of a set of rewrite rules, where the nonterminal symbols are semantic entities such as *ship, ship-property, location etc.*, rather than syntactic entities such as verb, noun, noun phrase, *etc.* Figure 2.7 show a simplified fragment of a semantic grammar used in the LADDER system [17]. PLANES [35], and SOPHIE [5] are also well-known semantic grammar based natural language interfaces.

In semantic grammar systems, the question-answering is done by parsing the input and mapping the parse tree to a database query. The grammar's categories do not necessarily correspond to syntactic concepts. Since the semantic information about the knowledge domain is hard-wired into the semantic grammar, its categories are usually chosen to enforce semantic constraints. They can also be chosen to

```
  S  ⇒  NP VP
 NP  ⇒  Det N
Det  ⇒  "what" | "which"
  N  ⇒  "rock"| "specimen"| "magnesium"| "radiation" | "light"| ...
 VP  ⇒  V N
  V  ⇒  "contains" | "emits"| ...
```

Figure 2.8: A Simple Syntactic Grammar

facilitate the mapping from the parse tree to database objects. Semantic grammars are useful when only a relatively small subset of a whole language needs to be recognized, but they do not capture much of the syntactic regularity of the language. As they are expanded to deal with larger and larger pieces of the language, they tend to get much larger and much more complex. Also a new semantic grammar has to be written whenever the NIL is configured for a new domain since semantic grammars contain hard-wired grammars about a specific domain.

Syntactic grammars To capture as much of the regularity of the language as possible in the rules used to understand it, the syntactic regularity of the language used must be captured. Several techniques are available for conducting the syntactic parsing. In 1970s, the most often used method was ATN (Augmented Transition Network), as in LUNAR [37] and ROBOT [16].

Before 1980, it was widely accepted that human language cannot be effectively captured by context-free rules alone. Gazdar in his Generalized Phrase Structure Grammar[12] has shown that unbounded dependencies, which were believed unexpressible by context-free rules, can be elegantly treated by context-free grammars.

Figure 2.8 shows an over-simplistic grammar in a LUNAR-like system. It says that a sentence (S) consists of a noun phrase (NP) followed by a verb phrase (VP). A noun phrase consists of a determiner (Det) followed by a noun (N). A determiner

may be "what" or "which".

Syntax-based NLI usually interface to application-specific database systems, that provide database query languages carefully designed to facilitate the mapping from the parse tree to the database query. It is usually difficult to devise mapping rules that will transform directly the parse tree into some expression in a real-life database query language(e.g SQL).

Therefore the problem with syntax based approach is that it is fragile and inflexible. It is not easily adaptable to quasi-grammatical or abbreviated utterances. It is also unclear whether a satisfactory domain-independent semantics can be supplied. On the other hand semantic grammars and their like are much less fragile, and much more flexible. But, they are significantly more expensive to produce and to use. Also, they must be recreated for each new domain.

Principle-Based Parsing Both syntactic and semantic parsing are so-called *rule based* parsing. As mentioned above, a rule-based parser has restrictions: it is too inflexible, too specific, too fragile, too hard to maintain; and furthermore, adding a few more rules are inadequate for a real-world problem. Many research groups have demonstrated that thousands of rules are not enough for a real-world problem, evoking doubts about whether traditional rules are really the right representation for expressing linguistic competence [3].

Principle-based parsing replaces large set of rules used to parse sentence with a much smaller, fixed set of parameterized, universal principles. It is not like traditional parsing that relies on many of individual, language-particular rules, exemplified by augmented transition network(ATN) or most systems based on context free grammars(CFG). It repairs the following defects:

- a partially well-formed sentence,
- pre-program rules for dialect variant,
- rewriting rules for other languages, and
- a rule set which is too large.

In contrast to rule-based parsing, a principle-based approach aims to reconstruct the vocabulary of grammatical theory in such a way that surface and language-particular constructions from the interactions of a small set of primitive elements. Figure 2.9 illustrates the difference. The top half of the figure shows a conventional rule-based approach. Each sentence type is described by a different rule. The bottom half of the figure shows a principle-based approach. Sentence types are derived from a small set of more fundamental principles that deductively interact to yield the effect of constructions. Naturally, no single principle accounts for all variations in a language. It is the interaction that matters.

Government-Binding(GB) Theory [14] provides the principle set for natural language parsing. This theory includes:

- the **X-Bar Theory** which describes the bar shapes allowed in a language.
- the **Theta Criterion** which dictates that every verb must discharge its thematic arguments - its placeholders that flesh out who did what to whom.
- the **Case Filter** which says that pronounced noun phrase must receive a case. A case is an attribute of a noun that indicates the type of position occupied by the noun.
- the **Binding Theory** which spells out how pronouns may be related to their antecedents in certain configurations.

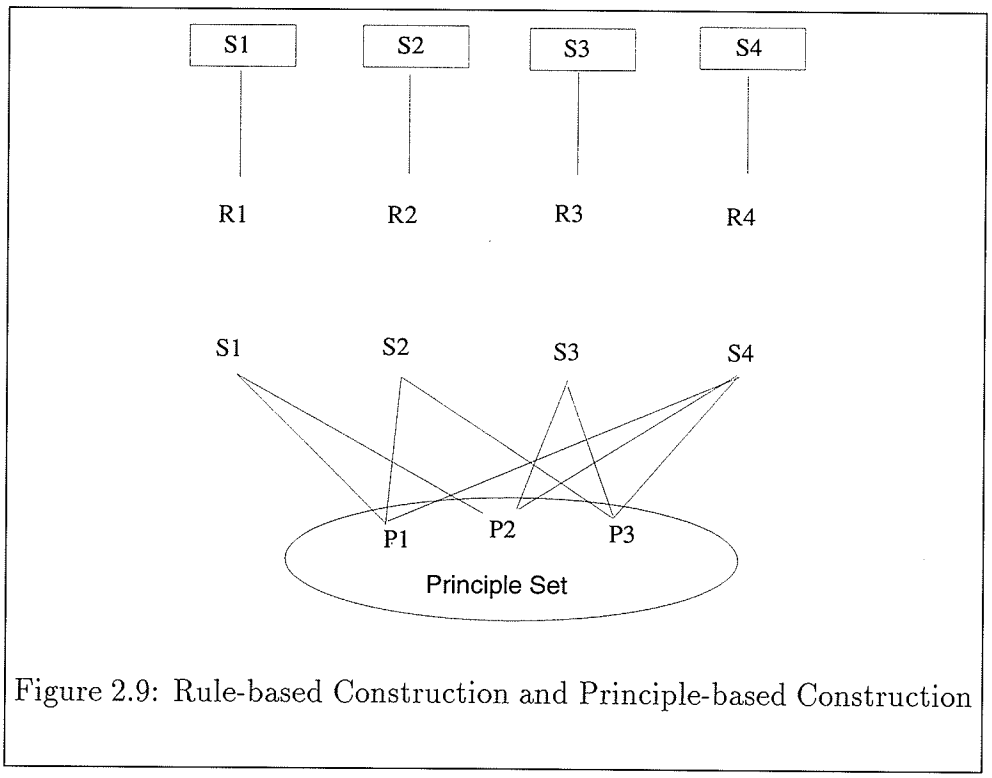

Figure 2.9: Rule-based Construction and Principle-based Construction

- the **Movement Principle** which says basically that any phrase can be moved anywhere.

Given a candidate structure of a sentence, principles can be used to determine whether it is an illicit structure. If a structure satisfies all the principles, the sentence is a grammatical one. If there are multiple candidate structures that all satisfy the principles, then that sentence is syntactically ambiguous.

Intermediate representation languages Some NLIs first transform the natural language question into an intermediate logical query, expressed in some internal meaning representation languages. The intermediate logical query expresses the meaning of the user's question in terms of high level world concepts, which are independent of the database structure. The logical query is then translated into an

expression in the database's query language, and evaluated against the database.

MASQUE [20] is a natural language query interface for database. The system, a descendant of Qarren and Pereira's CHAT-80 [36], transforms written English questions into Prolog-like queries, called *logical query language*. It is roughly a subset of Horn logic that can be used to express the meaning of a large set of English. MASQUE/SQL then extends the system with the translation component which can map those intermediate queries to SQL. For instance, the question "what is the salary of each technician" will generate the MASQUE logical query:

```
answer([Salary, Person]):-
is_technician(Person),
salary_of(Salary, Person).
```

Difficulties in NLI Although there has been tremendous progress in recent years, the ultimate goal of creating machines that can interact in a facile manner with people remains far off, awaiting both improved information-processing algorithms and alternative computing architectures. Here we discuss a number of issues in the design of natural language systems.

Habitability Having adequate habitability means allowing user to present the request in natural language sentences that come readily and comfortably to mind. A highly restricted subset of English is merely another formal language to learn.

Natural languages feature rich syntactical structures, which render a great amount of difficulty for computers to understand. A partial list of hundreds of variations of a simple query: "What are the salaries of female employees?" is shown in [18]. A syntax based approach has to cover all the sentence patterns for the

interface to be habitable. Providing access to a breadth of functionality through a wide variety of linguistic constructions requires that the natural language front-end be considerably sophisticated.

Adaptability It is widely accepted that any natural language interface to database must contain and use the knowledge about the domain with which the database is concerned. This domain knowledge serves a wide variety of functions throughout parsing, interpretation, and translation of natural language questions into database queries. Transferring this knowledge from one domain to another is one of the greatest difficulties in natural language system design. It is very difficult for non-linguists to provide linguistic information about a new domain. Hundreds or even thousands of new database-specific words must be identified and represented before the interface becomes useful. The structural information about the database must also be provided. This, unfortunately, often creates what is known as the *NLP Wall.* The essence of the problem is this: the very purpose of a natural language interface is to access the database without worrying about syntax of query language and the underlying database management system, but building this interface incurs the need of expertise in computational linguistics which is even more expensive and difficult to obtain.

2.4 Universal Relations

The universal relation approach [21, 27] provides a user view that the database is a universal relation which consists of all the attributes in any of the relation of the database. The user is thus spared memorizing details concerning which attributes are grouped with which to form relation schemes. For example, query

(A) on page 5 is formulated in System/U, a universal relation system developed at Standford University, as follows:

retrieve(part, partcost) where partcolour="blue"

The universal relation systems implicitly make the following assumptions that must be satisfied by the application for the universal relation model to be adequate for the application.

Unique Role Assumption Each attribute in the universal relation plays only one role. For example, an attribute *name* cannot stand for names of employees, customers, and suppliers in the same universal relation scheme. Essentially, this requirement forces the database designer to embed access path in attribute names.

Unique Relationship Assumption For every set X of attributes there is a basic relationship that the user has in mind. The user's queries refer to these basic relationships rather than to the underlying database. We refer to this basic relationship as the connection on X, denoted by $[X]$.

Query answering in Universal Relation consists of two steps:

1. *Binding:* Construct the connection $[X]$ for the set X of attributes mentioned in the query.

2. *Evaluation:* Whatever operations must be applied to answer the query are then applied to $[X]$.

The evaluation phase is independent of the binding phase and generally straight forward. The key problem is how to compute the connection $[X]$. In earlier implementations of Universal Relation [1], the connection $[X]$ is computed by taking

the natural join of all the relations in the database (The relations can be joined completely by introducing tuples with null values), and then project them onto X. This approach requires the database scheme be acyclic.

Maier and Ullman [26] demonstrated the deficiency of the acyclicity requirement. They further explained how the concept of maximal objects will allow for cyclic relational schemes. In their approach, an object is a computed relation representing a meaningful (acyclic) connection between all its attributes. A maximal object is an object contained in no other object. They are predefined by the database designer and remain static throughout.

Given a query, with X being the set of attributes in it, the connection $[X]$ is computed as follows:

1. find all the maximal objects that contain all the attributes in X.

2. If there are no such X, then $[X]$ is empty. If there is one such maximal objects, then $[X]$ is the projection of the natural join of all the objects that make it up. If there are several such maximal objects, we first take the natural join for each maximal object, project onto the attributes in X, and then take the union of the results.

Although Universal Relation is believed by many researches to be the most promising approach of providing easier access to database systems [33], it suffers from a number of problems:

- The Unique Role Assumption often leads to a proliferation of unintuitive attribute names.

- Although it is reasonable to assume that the user has the basic connection in mind when posing a query, the *basic* connection between two or more

attributes often depends on the context. In the universal relation approach, the connections are predefined through maximal objects.

Chapter 3

Understanding Query by Abduction

3.1 Abductive reasoning

Finding explanations for properties, events and actions is one of the most important aspects of natural language understanding. We make inferences from the observed states, events and actions to the goals of the characters described in the text. Recently, abduction has been glowingly recognized as one of the promising inferences to explanation.

The philosopher C.S. Peirce [29] defined abduction as the process of finding the best explanations for a set of observations; that is, inferring causes from effects. It is a kind of non-deductive inference that falls into the following pattern:

let α and β be two sentences,

Given	$\alpha \rightarrow \beta$
and	β
Infer	α, because, if α is true, β would be explained.

Abduction, together with deduction and induction, are the three basic forms of inference. Although it is not known as widely as the others, it has shown its promising prospects in many different AI research areas, such as natural language understanding, diagnosis, and plan recognition. To understand natural language by abduction is to perform abductive reasoning in syntactic, semantic, and discourse analysis. The task of understanding is viewed as finding the structural relationships between unstructured inputs. That is, to understand is to seek the best explanation of how the inputs are coherently related. From this abductive perspective, the goal of the parser is to explain how the words in a sentence are structured according to syntactic relationships; the goal of the semantic interpretation is to find the semantic relationships among the content words in a sentence; the goal of discourse analysis is to show how the events mentioned in the sentences fit together to form a coherent plan.

3.2 Obvious abduction

Although the abductive formulation of natural language understanding tasks results in significant simplifications [19], the computational complexity of abductive inference presents a serious problem. Lin proposed and prototyped a solution to this problem, called *obvious abduction*[22].

Obvious abduction is a model of abductive inference that covers the kinds of abductive inferences people perform without apparent effort, such as parsing, plan

recognition, and diagnosis. Obvious abduction uses a network to represent the domain knowledge. In the proposed abductive diagnosis, the nodes in the causal network represent events, which are the presence of symptoms, disorders or intermediate states; the links represents causal relationships between events. In plan recognition, the nodes in a plan hierarchy represent event types, subgoals as well as an ultimate goal. The links represent structural relationship among the plan types such as decomposition. Many tasks in natural language understanding can be viewed as abductive reasoning. These include both low-level tasks such as work sense disambiguation and high-level tasks such as discourse understanding. In syntactical parsing, the nodes represents the syntactical units, such as subjects and verbs; while the links represent the dominance relationships between the words.

The common theme in obvious abduction applications can be summarized as follows: the domain knowledge is represented by a network, whose nodes represent the objects and the links in the network represent the structural and taxonomic relationship between the objects. Observations correspond to nodes in the network annotated with a set of attributes. An explanation of the observations is a generalized subtree of the network that connects all the observations. This connection is a coherent set of relationships between the observations; therefore, it explains how they are related.

To find the all the explanations of given observations, Lin presented a generic message passing algorithm. In this algorithm, the nodes in the network are also computing agents which communicate by sending messages. Each message contains an item. An item consists of the subset of observations, an attribute assignment, and a set of features.

Definition 3.2.1 (Item) *An item is a triple:* $<O, V, F>$, *where* O *is the subset*

of the observations to be explained, V is an attribute assignment, and F is a set of features. We call O, V, and F the O-component, V-component, and F-component of the item respectively.

The features represent the relations between nodes, such as the causal relationships in diagnosis and the decomposition in plan recognition. The messages are passed in the reverse direction of the links in the network. An item at a node means that the observation in the item can be explained as features of an object that satisfied the description of that node with assigned attributes. As a result, the items represent partial explanations of the observations. The partial explanations are combined at the nodes to generate explanations for larger subset of observation. The node then sends out combined new messages until at some nodes the observation set covers all of the observations. The message passing process stops when no further messages are being sent and the explanations can then be retrieved from the network by tracing the origins of the messages whose observation set contains all the observations. The consistency of the explanations is guaranteed by the enforcement of constraints during the passing process.

To apply obvious abduction to database query understanding, we use the E-R diagram or database graph as the knowledge base. When querying the database, we submit a query in English. There are some words or phrases that relate the values or entities in the database. We isolate these words/phrases and map them onto the nodes in the database graph. The abductive inference algorithm can be used to find a a generalized subtree of the network that connects all the observations. The query access paths are obtained from the connection tree. Since, in general, several alternative connections may be possible, the shortest connection, which has the minimum total length, is used to resolve the ambiguity. The understanding of

queries in natural language is well-fitted for the obvious abduction.

3.3 Knowledge Base and E-R Diagram

There are many formalisms to represent database schemes by graphs, such as Entity-Relationship Model [6], Semantic Database Model (SDM) [15], Semantic Association Model (SAM*) [31], *etc.* These models serve primarily as database schema design tools. In this thesis, we take advantage of these models that encourage a more navigational view of data relationships.

The Entity-Relationship(E-R) diagram is a high-level conceptual data model. It is used mainly as a tool during the process of database design [10]. It is one of the best domain knowledge sources for us to understand the database.

The basic object in E-R diagram is an *entity*, which is a "thing" in the real world. An entity may exist physically —an employee— or conceptually — a job. A database usually contains a group of entities which are similar. Such similar entities define an *entity type*. Each entity is an *instance* of the entity type. For convenience, we use the term "entity" to refer entity type and "entity instance" to refer entity in this thesis.

Each entity is described by an entity name and a list of properties, called *attributes*. For example, an entity with name employee may have attributes "name", "age", "address", and so on. A particular entity instance will have a *value* for each of its attributes. For example, an instance of entity employee could have the following values: the name is John Smith; the age is 55; the address is 5500 Portage Ave.

An entity usually has an attribute whose values are distinct from other individ-

ual instances. Such an attribute is called the *key attribute* or simply *key* and its values can be used to uniquely identify an instance. The "Social Insurance Number" can be used as the key of the entity "employee" because each person has a unique value for this attribute.

Each attribute is associated with a *value range* or *domain*, which specifies the set of values that may be assigned to that attribute for an instance. For example, the range of "Age" of an employee could be between 18 and 70. The range of Name can be a set of strings of alphabetic characters.

In addition to entities, a databases typically also contains relationships among the entities. A *relationship type* R among n entity types $E_1, E_2, ..., E_n$ is a set of associations $r_1, r, ..., r_m$ among entity instances from these types. Each of the entity $E_1, E_2, ..., E_n$ is said to *participate* in the relationship R. Similarly, we use "relationship" to refer the relationship type and "relationship instance" to refer each r_i in R. For example, consider a relationship WORKS_FOR between the two entity EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works. Each relationship instance associates one employee instance and one department entity instance.

The *degree* of a relationship is the number of participating entities. Hence the WORKS_ FOR relationship is of degree two. Such degree relationship is called binary relationship. An example of relationship of degree greater than 2 is supplier-supply-part-project, representing the relationship where a supplier can supply different parts for different projects.

Each entity that participates in a relationship plays a particular *role* in the relationship. We use the term *role name* to signifies the role that a participating entity plays. Usually the role name is highly associated with the participating entity but it is not necessarily the same. Sometimes it is hard to come up with a

role name. The role name is not necessary in most cases, except when the entity participates in a relationship more than once in different roles. Such a case is also called *recursive*. For example, a SUPERVISION relationship relates a employee and a supervisor where both employee and supervisor are the same entity EMPLOYEE. Thus we give EMPLOYEE two different role names for its different participation, supervisor and supervisee.

Relationships usually have certain constraints that limit the possible combinations of entity instances participating. The most frequently occurring constraint is called *cardinality ratio* that specifies the number of relationship instances that an entity can participate. The cardinality ratios are annotated as 1:1, 1:N, M:N. For example, if we know an employee can only manage one department, then the relationship MANAGE is 1:1; many employees may work for one department, so WORKS_FOR is N:1; the relationship WORK_ON is M:N if we know an employee can work on several PROJECTs and that several employees can work on one project.

In E-R diagrams, the graphical representation of E-R models, entities are represented by rectangles, relationships are represented by diamonds and attributes are represented by ovals. The entities that participate in a relationship set are indicated by arcs which connect the entities to the relationship. Arcs are also used to indicate an attribute belongs to which entity set or relationship set. The E-R diagram of the "employee" database is shown in Figure 3.1.

An *Enhanced E-R diagram*(EE-R) is an E-R diagram with additional concepts of specialization, generalization, and inheritance. In many cases, an entity has numerous additional subgroupings of its entities that are meaningful and need to be represented explicitly because of their significance to the database application. For example, in the "company business" database the member of "company" may

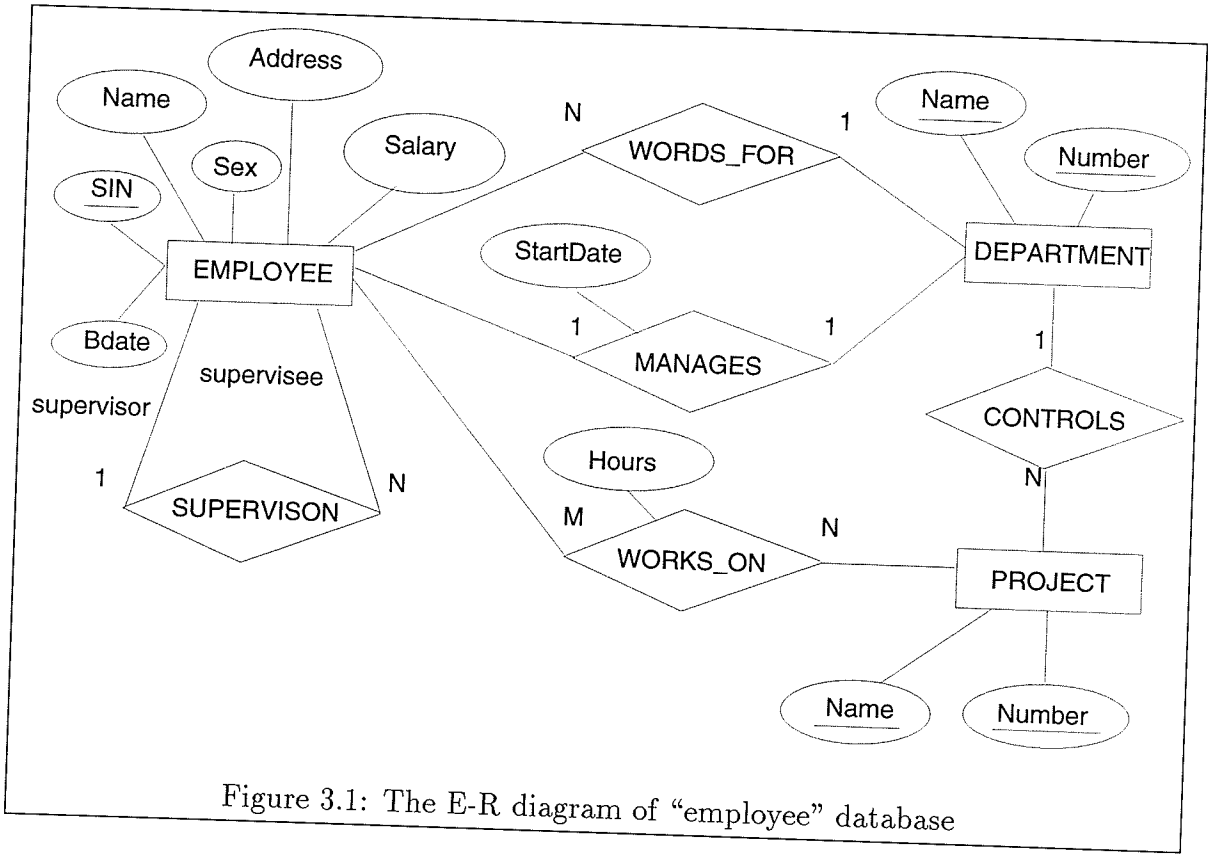

Figure 3.1: The E-R diagram of "employee" database

be further grouped into "customer" and "supplier". The set of entities in each of the latter groupings is a subset of the entities that belong to the "company" , meaning that every entity instance that is "customer" or "supplier" is also a "company". We call each of the subgroupings a *subclass* of "company", and "company" is called the *superclass* for each of these subclasses. We call the relationship between a superclass and any one of its subclasses a *class/subclass relationship*. An important concept associated with subclasses is that of *attribute inheritance*. Because an entity instance in the subclass represents the same real-world entity instance as the superclass, it should possess values for its specific attributes as well as values of its attributes as a member of the superclass. We say that a subclass *inherits* all the attributes of the superclass. The process of defining a set of subclasses is called

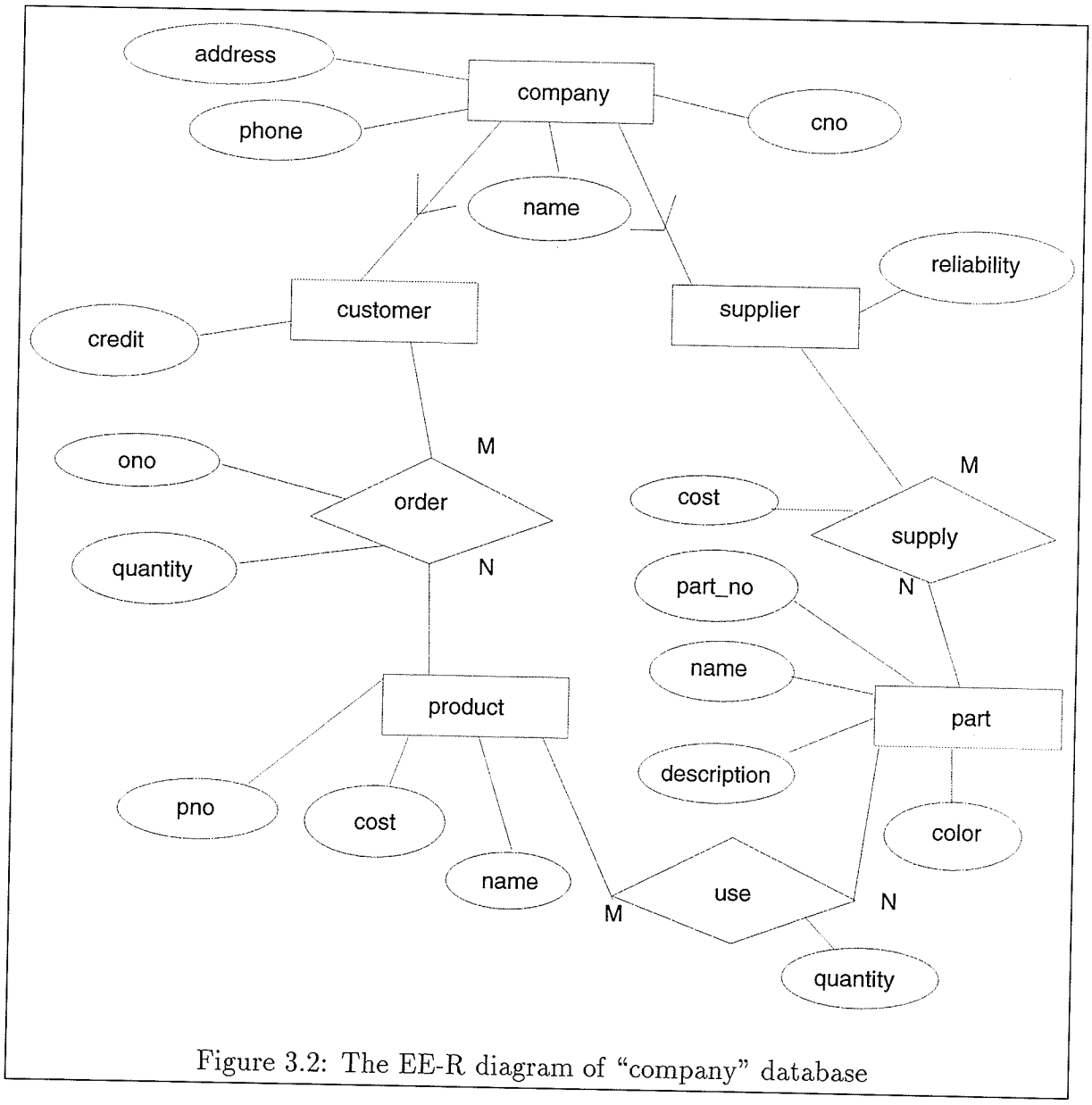

Figure 3.2: The EE-R diagram of "company" database

specialization. In the reverse process of abstraction, we suppress the differences among several entities, identify their common features, and *generalize* them into a single superclass. This process is referred to *generalization*. Figure 3.2 shows the EE-R diagram of the "business" database. The superclass/class relationship is

labeled with a "greater than" symbol. There are many other notations to present the detailed relationships between superclass and subclass in [10].

3.4 Queries and Query Graphs

For our purpose, we restrict ourselves to acyclic queries that only involve natural joins and correspond to subtrees in the database graph.

Bernstein and Chiu[2] have shown how a query in the relational model can be represented by a query graph. In this thesis, a *query graph* is a connected subgraph of a database graph. While not all proper queries correspond to a single connected subgraph, a query that corresponds to several disjoint subgraphs could safely be regarded as a collection of independent queries that were submitted together. Based on this observation, we may assume that only queries that correspond to a single connected subgraph are permitted.

The query corresponding to a query graph is a relational expression in which the natural join of all the entities and relationships adjacent in the query graph are projected onto the attributes in the query graph.

For example, consider the query:

display the reliability of the suppliers of blue parts.

Its corresponding query graph is shown in Figure 3.3. The query in SQL is:

```
Select supplier.reliability, company.name
From   supplier, supply, part, company
Where  company.cno = supplier.cno AND supplier.cno = supply.cno
       AND supply.pno = part.pno AND part.color = ''blue''
```

Note that different queries may correspond to the same connected subgraph.

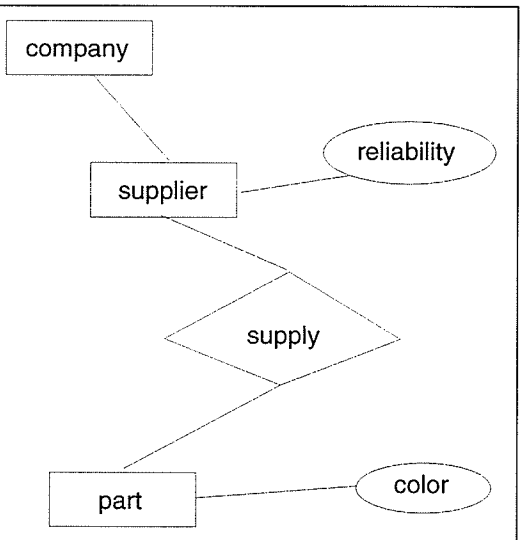

Figure 3.3: An Query Graph Example

3.5 A Framework for Understanding Queries in Natural Language

There are four major components in the system architecture: lexical analyzer, parser, semantic analyzer, and SQL translator. Figure 3.4 shows the system architecture.

3.5.1 Lexicon

The lexicon contains the mapping from the words to their semantic and/or syntactic representations. It holds almost the same contents as the grammar part of a ordinary dictionary, such as word categories, and phrases. In addition, it provides the mapping between the words and the nodes in the E-R diagram.

We have constructed two lexicons in the system. One is the generic lexicon,

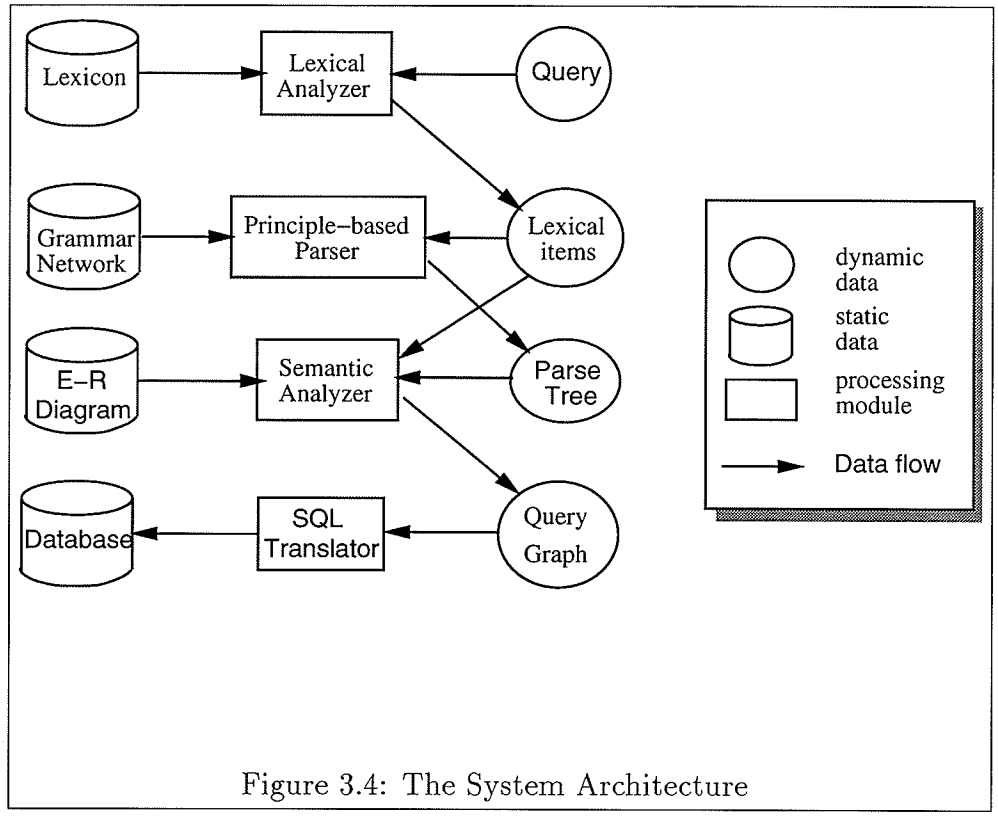

Figure 3.4: The System Architecture

which is the computational representation of syntactic definitions of each single word; the other is domain dependent lexicon, which holds the correspondence between the words used in queries and the nodes in the E-R diagram. We have to reconstruct the domain dependent lexicon for each new database scheme. The reasons for two separate lexicons is to make the system easy to configure for different domain application.

The domain dependent lexicon consists of two kinds of entries: metadata entries and data value entries. Metadata entries refer to attributes, entities or relationships in the database. Data values, on the other hand, appear as values in the database. For instance, **company**, and **order** are metadata values, whereas **acme**, and **red** are data values. Metadata values are mapped onto the nodes which represent them

...	...	...
marco	customer	data value
marco	supplier	data value
ged	customer	data value
...	...	...
...	...	...
colour	color	metadata
color	color	metadata
order	order	metadata
red	color	data value
...	...	...

Figure 3.5: An Excerpt from the Lexicon

in the graph. The data values are mapped onto the nodes which represent the domain to which they belong. For example, the corresponding node of **company** is *company* and the corresponding nodes of *acme* are *customer* and *supplier*.

The lexicon is simply a list of triples composed of an entry word, the identifier of the corresponding node and the data type. The data type specifies whether the word is a data value or a metadata value. We note that a data value is considered ambiguous if it may belong to more than one domain. An excerpt from the lexicon of the sample database is shown in Figure 3.5

3.5.2 Lexical analyzer

The lexical analyzer takes English input from users and recognizes the sentence boundaries by creating a set of lexical items which are the internal representations of words in a sentence. The power of the lexical analyzer depends on the system lexicons.

Lexical rules are used to recognize numbers, locations, names of persons and organizations that consist of multiple words. Only non-numerical data values appear in the lexicon. We have one restriction on numeric values in queries: the domain

of numerical values must be specified in the query. For example, *quantity* = 3, *cost* < 15.00, *etc.*

The lexical analyzer recognizes the variations of the English words, such as -ing forms, -ed forms, and plural forms. Although the domain dependent lexicon holds only the root forms, the lexicon can also be accessed by variations of a word.

3.5.3 Principle based parser

The efficient parsing of natural language is one of the key issues of natural language understanding. In Chapter 2.3, we have briefly compared the *rule based* parsing and *principle-based* parsing. Although there are many advantages of principle-based grammars, such as the universality of principles and modularity of components in the grammar, principle-based parsers have not achieved the coverage and speed of rule-based and unification-based parsers. This is largely due to the mismatch between linguistic devices used in principle-based grammars and computational devices available for the parser. In contrast, both rules and unification of structures are well understood computational devices. Lin presented a principle-based parser in [25]. The key idea of this parser is to provide computational models of linguistic devices such as government, case-marking, barrier, *etc.* These computation models enables procedural interpretation of declarative principles more efficient than previous principle-based parsers. In fact, the parser is even more efficient than rule and unification-based grammars with similar coverage[24].

Lin and Goebel [23] presented a message passing algorithm for parsing. The algorithm constructs parse trees from the bottom-up. The grammar is represented by a network. The nodes in the network represent grammatical categories (e.g., NP, Nbar, N) and subcategories, such as V:NP (transitive verb that takes an NP as

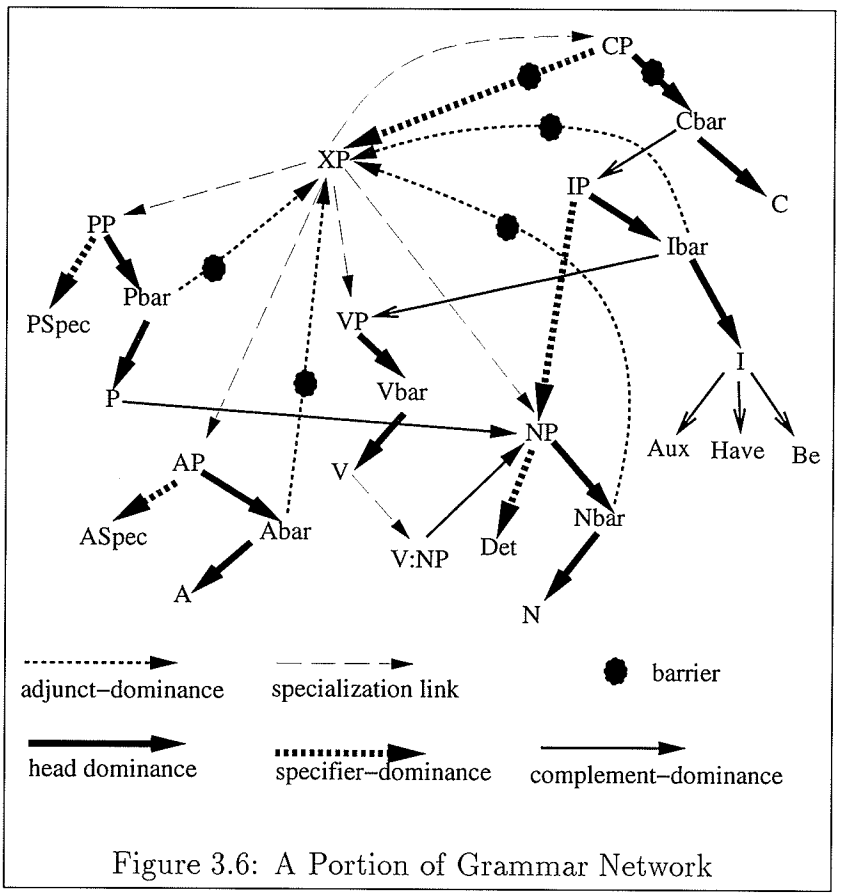

Figure 3.6: A Portion of Grammar Network

complement). The links in the network represent dominance relationships between the categories.

Figure 3.6 showed a portion of grammar network for Government-Binding(GB) Theory. Input sentences are parsed by passing messages in the grammar network. The nodes in the network are computing agents. They communicate with each other by passing messages in the reverse direction of the links in the network. Each node in the network has a local memory which stores a set of items. GB principles are implemented as local and percolation constraints on the items. Local constraints are attached to nodes in the network. All items at a node must satisfy the local constraint of the node. Percolation constraints are attached to the links in the

network. When an message can be sent across a link only if the item satisfy the percolation constraint of the node.

The result of parsing is the parse trees. The nodes in the grammar network represent categories. The nodes in a parse tree, on the other hand, are instances of grammatical categories. Obviously not all parse trees are subtrees of the grammar network, since a parse tree may contain multiple instances of the same category. However, a subtree that all the links that are adjacent to a single node in the parse tree must be a subtree of grammar network. Figure 3.7 shows the simplified parse tree of sentence "display the reliability of suppliers of blue parts". The complete parse tree of this sentence by the parse is listed in Appendix B.

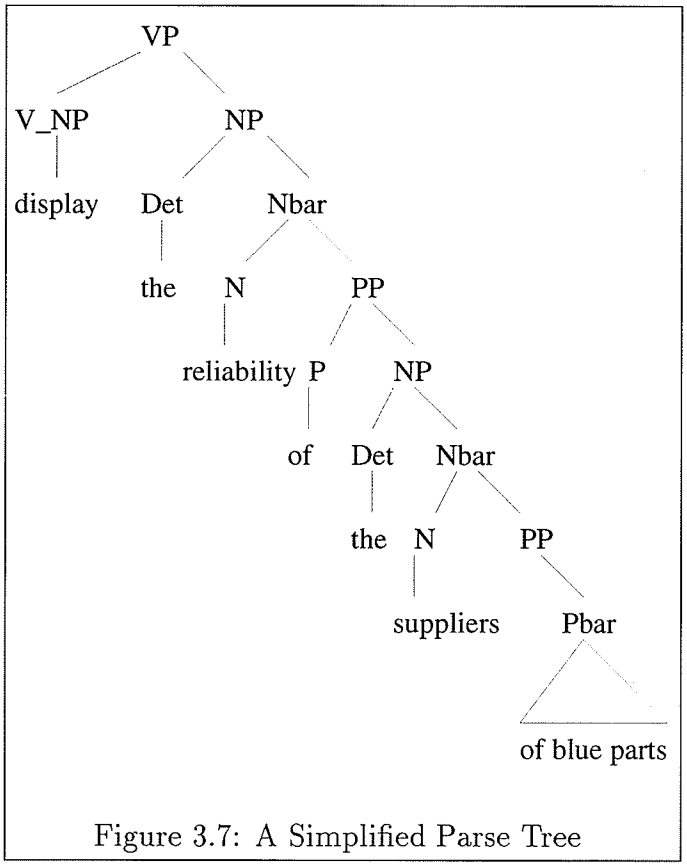

Figure 3.7: A Simplified Parse Tree

3.5.4 Construct Query Graph by Abduction

From the lexical items, we have constructed parser trees. We use the same message passing algorithm to construct query graph, i.e Steiner Tree or the shortest connection in graph theory terminology.

A lexical item corresponds to a node in the E-R diagram, annotated with a set of attribute values. The nodes in the network are computing agents. They communicate with each other by passing messages in the reverse direction of the links in the network. Each node in the network has a local memory which stores a set of items. An item is a triple that represents an explanation of a subset of the observations:

<observation, attribute-values, features>, where

`observation` is an integer interval [i,j] denoting the i'th to j'th word in the input sentence.

`attribute-values` specify attributes, including `meta, datavalue`, of the root word.

`features` represent a relationship between nodes. In our case they are the links via which the item is passed. The initial feature set are empty.

When a node receives an item, it attempts to combine the item with items from other nodes to form new items. That is, the node combines several smaller explanations into a larger one. Two items

$$<[i_1,j_1], A_1, F_1> \text{ and } <[i_2,j_2], A_2, F_2>$$

can be combined if

1. their observations are adjacent to each other: $i_2 = j_1+1$.
2. their attribute values A_1 and A_2 are unifiable.

The result of the combination is a new item:

$$<[i_1,j_2], \text{unify}(A_1, A_2), F_1 \cup F_2>.$$

The new items represent larger explanation resulted from combining two smaller ones. They are then propagated further to other nodes if they satisfy the completion predicate.

Each node has a **completion predicate**. If the attribute values of a message satisfy the completion predicate, the message is sent further to other messages. Otherwise, the message waits to be combined with other messages at the node.

Filters can be attached to the links in the semantic network. A filter is a set of conditions. A message can only pass through the filter if it satisfy those conditions.

When the message passing process stops, we can find the best explanation by tracing the origins of the messages that explain the largest number of observations, i.e it covers all of the given lexical items in a query.

We summarize the process as following steps.

Step 1: Lexical Look-up Look up all words in the lexicon and create a lexical item for each word. The lexical item

$$<[i,j], A >$$

where

[i,j] is an interval denoting the position of the word in the sentence;

A is the attribute values of the word ;

Step 2: Message Passing For each lexical item

$$<[i,j], A>$$

create an initial message <[i,j], A, Ø>, the feature set is Ø. The message is sent to the node that represents the lexical item. When the node receives

initial messages, it may forward the message further to other nodes or combine the message with messages from other nodes and send the results of the combination further. A message passing process is then initiated. The message passing process stops when there are no more messages being passed around. At that point, the initial message for the next lexical item is fed into the network.

Step 3: Construct Explanation The explanation of the input words are represented by messages at the node, whose `observations` component covers all the words. These messages are descriptions of explanation, not explanation themselves. However, the explanations can be retrieved by tracing the origins of these messages.

Semantic disambiguity is achieved as a side effect of the search for the best explanation, i.e. we search the tree with minimum total length. Consider the query sentence:

"display the reliability of suppliers of blue part"

By the lexicon mapping we get the following lexical items:

- <[0,0], reliability (metadata)> , node reliability
- <[1,1], supplier (metadata)>, node supplier
- <[2,2], blue (datavalue)>, node color
- <[3,3], part (metadata)>, node part

Four initial messages are created based on the above lexical items. They are very similar to the lexical items:

- <[0,0], (reliability—metadata), Ø>
- <[1,1], (supplier–metadata), Ø>
- <[2,2], (blue–datavalue), Ø>
- <[3,3], (part—metadata), Ø>

They are sent to their corresponding nodes as shown in Figure 3.8.

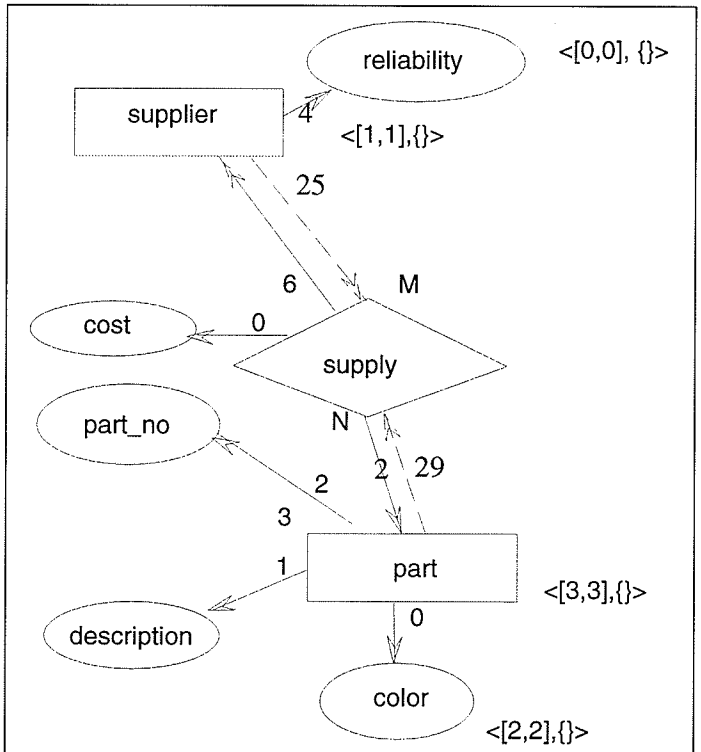

Figure 3.8: The Nodes Receiving Initial Messages

Some of the messages the passing algorithm creates are listed below:

1. item <[0,0] {∅}> is sent to node *reliability* and forwarded to node *supplier* as item <[0,0] {4}>, *supply*, *part*, and other nodes;

2. item <[1,1] {∅}> is sent to node *supplier* and forward to other nodes. At node *supplier*, item <[0,0] {4}> combines item <[1,1] {∅}> and forms item <[0,1] {4}>. Item <[0,1] {4}> is forwarded to other nodes. At node *supply*, item <[0,0] {6}> combines item <[1,1] {6}> and forms <[0,1] {6}>. Figure 3.9 shows the messages and their combinations.

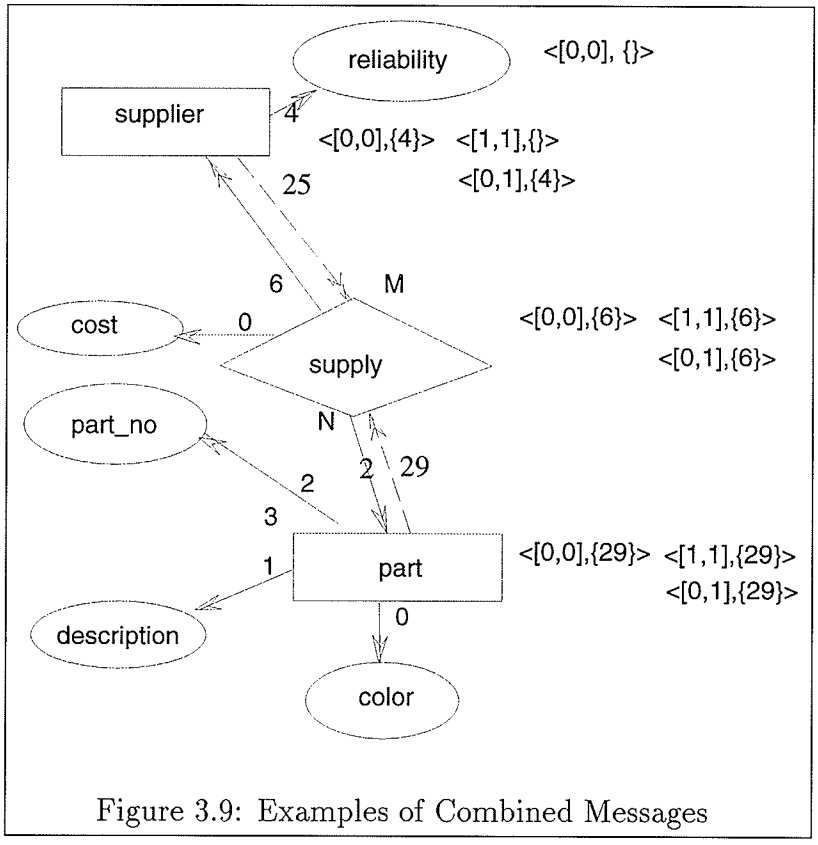

Figure 3.9: Examples of Combined Messages

3. item <[2,2] {∅}> is sent to node *color* and forwarded to *supply* as item <[2,2] {2}>, node *part* and other nodes. At node *supply*, item <[0,1] {6}> and item <[2,2] {2}> are combined to item <[0,2] {2, 6}>.

4. item <[3,3] {∅}> is sent to node *part* and forwarded to node *supply* as item <[3,3] {2}> and others. At node *supply*, item <[0,2] {2, 6}> and item <[3,3] {2}> are combined to item <[0,3] {2,6}>. And this item covers all observations and will be traced and lead to an explanation. The nodes and their stored messages are shown in Figure 3.10.

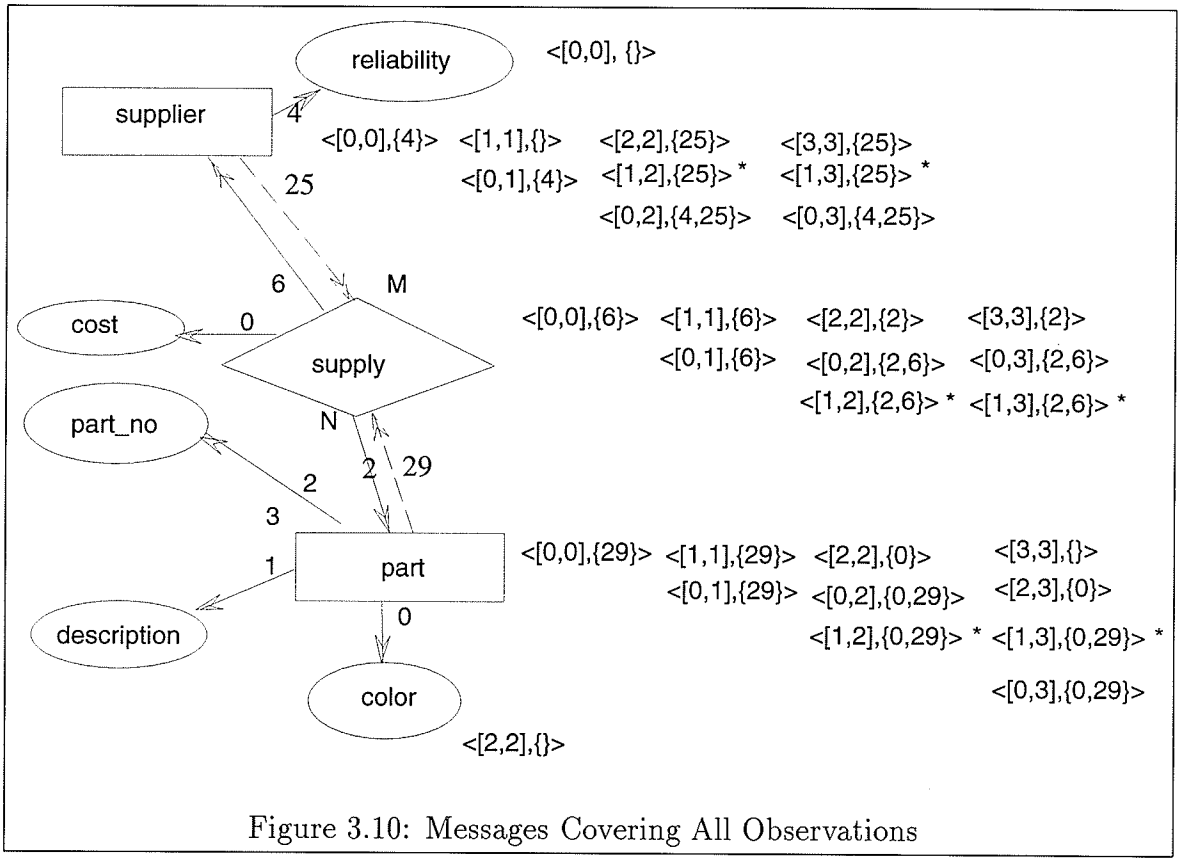

Figure 3.10: Messages Covering All Observations

There are many other messages we do not list above. Also there are many other nodes at which message are passed. Since they are disqualified by the shortest length threshold and we do not list them either.

Integration with Syntax Previous approaches to semantic interpretation can be classified as either one of the following:

Syntax-guided: Semantic structures are derived from a parse tree or parse tree fragments. The disadvantage of this is that the semantic analysis is critically dependent upon the output of syntactic analysis and syntactic ambiguities have to be resolved before semantic analysis.

Frame-guided: Semantic interpretation is driven by instantiation of frames that are triggered by keywords. The problem with this approach is that there is no principled method for controlling the interaction of multiple frames that may be triggered by the same word or the same set of words. Furthermore, this approach often results in complex frame definitions that are difficult to port to another domain.

We have not mentioned the syntactic constraints in the above discussion. There is one syntactic constraint enforced in the message passing process: the connection trees obtained by the semantic analyzer are not arbitrary; they must be convex with respect to the sequence of words in the sentence.

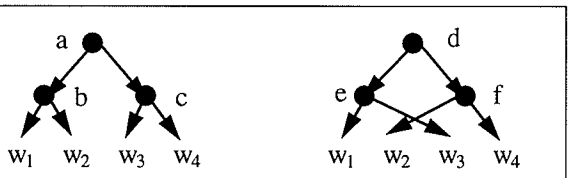

Figure 3.11: Convex and Non-convex Trees

Definition 3.5.1 (Convexity) *A tree connecting a sequence of elements is convex with respect to the sequence iff for any two elements w_i, w_j $(i < j)$ in the sequence,*

any node in the tree that dominates both w_i and w_j must also dominate all the element between w_i and w_j.

Figure 3.11 shows examples of a convex and a non-convex tree. Although the syntax of different languages may be very different, they all seem to satisfy the convexity constraint.

The semantic interpretation we implemented is **syntax-constrained** in the sense that the semantic structure must be consistent with syntactic structure. The notion of structural-consistency between semantic and syntactic structures is similar to the structural-consistency between parse trees.

Definition 3.5.2 (Span) *An integer interval $[i, j]$ is said to be a span of a sentence if there exists a parse tree and a node n in the parse tree such that the i'th to j'th word in a sentence is dominated exactly by a consecutive subtrees of n.*

Figure 3.12 shows the spans in a parse tree. The spans in a parse forest is the unions of the spans in the trees in the forest.

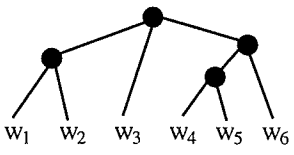

Figure 3.12: The spans in a sentence

Semantic interpretation is based on:

Definition 3.5.3 (Structural-consistency Hypothesis) *The spans in semantic dependency structure are a subset of the spans in the parse forest.*

From a packed shared parse forest, we can derive all the allowable spans in a sentence. During the message passing process, the messages are combined by a node only if the span of their combination is an allowable span according to the parse forest.

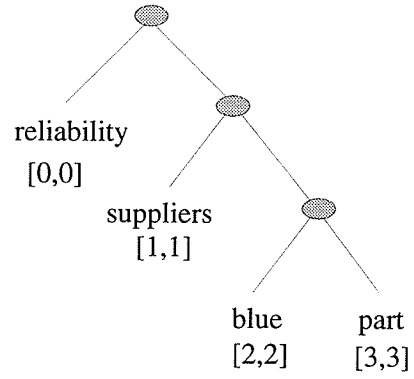

Figure 3.13: The Syntactic Constraints in Message Passing

Let us examine the example of query "display the reliability of suppliers of blue parts" again. If we get rid of those words unrelated to database items and the details of intermediate non-terminals, the parse tree shown in Figure 3.7 can be simplified as Figure 3.13. The final query graph is shown in Figure 3.3. The spans in the Figure 3.13 are [0,3], [1,3], [2,3]. During the message passing process, item [2,2] will be forwarded by node *part* to node *supply*, item [1,1] will reach node *supply* as well. Since [1,2] is not a span, item [1,1] and item [2,2] will not be unified to form a new item [1,2]. However, the stored item [2,3] will unify with item [1,1] to form the new item [1,3] since it is a span. The syntax constraints will eliminate a large number of incompatible items during the message passing process. The items marked with "*" in Figure 3.10 are not created.

Note the convexity constraint is a special case of the structural-consistency hypothesis, i.e. the parse tree is assumed to be a flat structure, where the category S (sentence) immediately dominates all the words(Figure 3.14). This means that any interval $[i, j]$ is an allowable span. The parse tree imposes no constraints on the semantic structure.

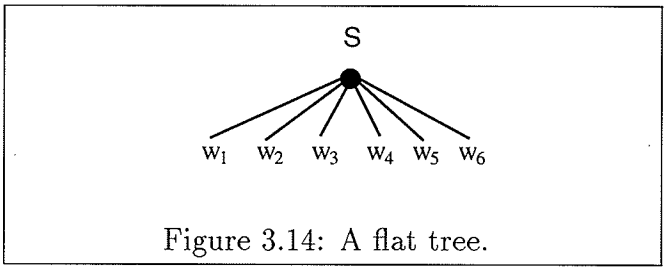

Figure 3.14: A flat tree.

3.5.5 SQL Translator

The query graphs can be translated into SQL statements. The translation is based on the mapping from E-R diagram to database schema (i.e relations or tables). Some theories have been developed to choose "good" schemas, that is, to measure formally why one set of grouping of attributes into a table may be better than another. The theory is called *functional dependency.* The criteria include normal form, update anomalies, spurious tuples, and so on. Here we assume that the E-R diagram we used is adopted from the database design expert; that is, the grouping of attributes and the design of entity relations are appropriate. Therefore, we can derive all of the tables of a database from its E-R diagram.

Here we present a mapping from E-R diagram to tables adopted from [10]. The author also gives the mappings to other implementation models such as network, and hierarchical models. Many CASE (Computer Aided Software Engineering) tools are based on E-R model to automatically convert the E-R diagram into a

relational database schema.

1. For each regular entity type E in the E-R schema, we create a relation R that includes all the attributes of E. We choose one of the key attributes of E as primary key for R.

2. For each binary 1:1 relationship type R in the E-R schema, we identify the relations S and T that correspond to the entity types participating in R. We choose one of the relations, say S, and include as foreign key in S the primary key of T. It is better to choose an entity type with total participation in R in the role of S. We include all the attributes of the 1:1 relationship type R as attributes of S.

3. For each regular binary 1:N relationship type R, we identify the relation S that represents the participating entity type at the N-side of the relationship type. We includes as foreign key in S the primary key of the relation T that represents the other entity type participating in R; this is because each entity instance on the N-side is related to at most, one entity instance on the 1-side of the relationship type.

4. For each binary M:N relationship type R, we create a new relation S to represent R. We include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. We also include any attributes of the M:N relationship type as attributes of S.

5. Convert each specialization with m subclasses $S_1, S_2, ..., S_m$ and (generalized) superclass C, where attributes of C are $k, a_1, ..., a_n$ and k is the (primary) key, into relation schemas using the following procedure: Create a relation L for C

with attributes $k, a_1, ..., a_n$ and primary key pk(L) = k. Also create a relation L_i for each subclass $S_i, 1 \leq i \leq m$, with the attributes k $\cup$ attributes of S_i and pk(L) = k.

From the above steps we learn that the some of relationship types (1:1 and 1:N) are not represented explicitly(i.e. they are not directly mapped to tables); instead, they are represented by having two attributes A and B, one a primary key and the other a foreign key included in two relations S and T. Two tuples in S and T are related when they have the same value for A and B. By using the EQUIJOIN (or NATRUALJOIN) operation over S.A and T.B, we can combine all pairs of related tuples from S and T and materialized the relationship. For a M:N binary relationship type, two join operations are needed.

To make the translation algorithm consistent and simple, we provide those missing 1:1 and 1:N relationship by creating temporary relation schema. This temporary relations are called *views* in SQL. Here is how we create them: For each binary 1:1 and 1:N relationship type R in the E-R diagram, S and T are the participating entities, create relation R with both primary keys of S and T as its attributes.

There are several types of entity, relationship, and attribute that we do not include in the above steps: they are n-ary relationship types, weak entity type and multi-valued attributes. Note n-ary relationship can be decomposed into 1:N, 1:1 or M:N relationship. With those temporary relation schemas, we treat weak entity types the same as the regular entity type. The multivalued attributes can be avoided by adding an extra relationship(1:N) and an extra entity.

The rules of translation are shown in Figure 3.15. The different cases of node connections are listed on the left; the translator's actions are listed on the right.

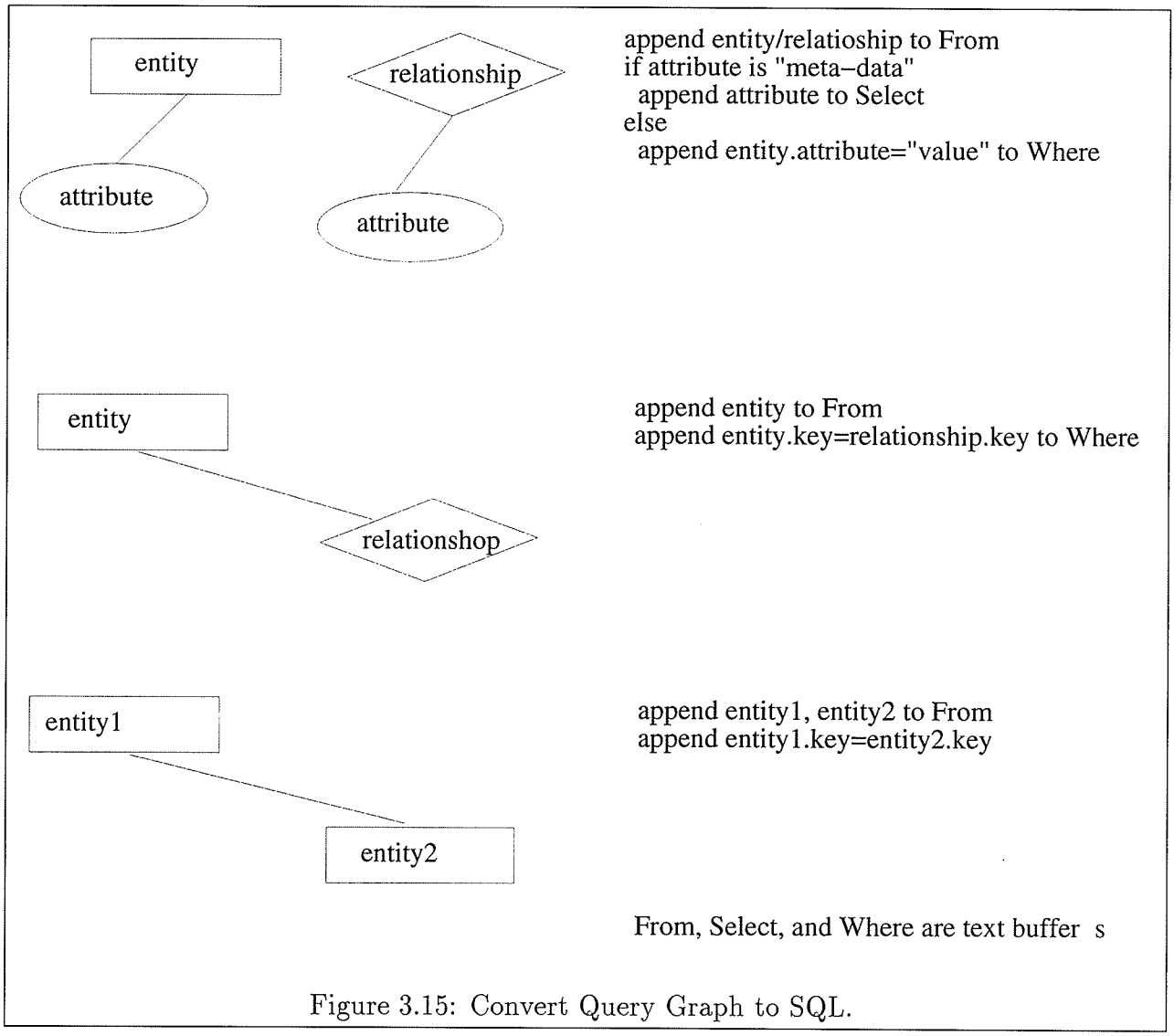

Figure 3.15: Convert Query Graph to SQL.

3.6 Query Examples

Recall the "company" E-R diagram. The following are some simple queries executed on the system.

Example 3.6.1 *List the products ordered by marco?*

node mapping : product, order, company

query graph : company ← customer ← order → product

SQL statement :

```
Select c.name p.name
From   product p, order o, customer cu, company c
Where  p.pno = o.pno AND c.cno = cu.cno AND c.name = ''marco''
```

Example 3.6.2 *Which company supplies green nut?*

node mapping : company, supply, color, description

query graph : company ← supplier → supply ← part → color, description

SQL statement :

```
Select c.name pa.name, pa.color, pa.description
From   company c, supply su, supplier s, part pa
Where  c.cno = s.cno AND s.cno = su.con AND
       pa.part_no = su.part_no AND
       pa.color = ''green'' AND pa.description= ''nut''
```

Chapter 4

An Object-oriented Implementation

4.1 Class hierarchy

In the object-oriented paradigm, we organize software as a collection of objects with data members and a set of operations. To represent the E-R diagram, we have specialized three basic object types:

- ERDiagram – abstraction of the network;
- ERDiagramNode – abstraction of all kinds of nodes;
- ERDiagramLink – abstraction of all kinds of links.

In an E-R diagram, we have three kinds of nodes, *entity, relationship* and *attribute*. In our view, there are few differences between an entity and a relationship. We further specialized *ERDErNode* for entity and relationship nodes and *ERDAt-*

trNode for attribute node. Figure 4.1 shows the hierarchy of the node classes. The shaded nodes are the ones I have implemented.

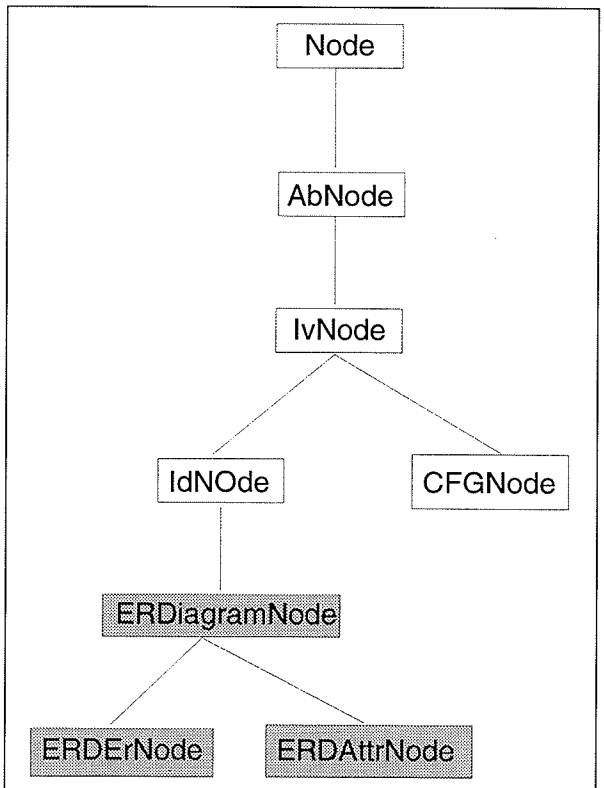

Figure 4.1: The Hierarchy of Node Classes

In Figure 4.1, *Node* is a generic node class. *AbNode* is the generic node with abductive inference specific behaviors. An *IvNode* is equipped with a *completion predicate*. *IdNode* has *item* storage. *CFGNode* is used to represent context free grammar in the principle-based parser.

There are four kinds of links in an E-R diagram. We specialize the following classes:

- ERDErLink – connect entity node and relationship node;

- ERDEraLink – connect entity/relationship node and attribute node;
- ERDEeLink – connect two entity nodes representing the relationship of a subclass "is" a superclass; and
- ERDSpecLink – connect two entity nodes representing the relationship that a superclass can be "specialized" as a subclass.

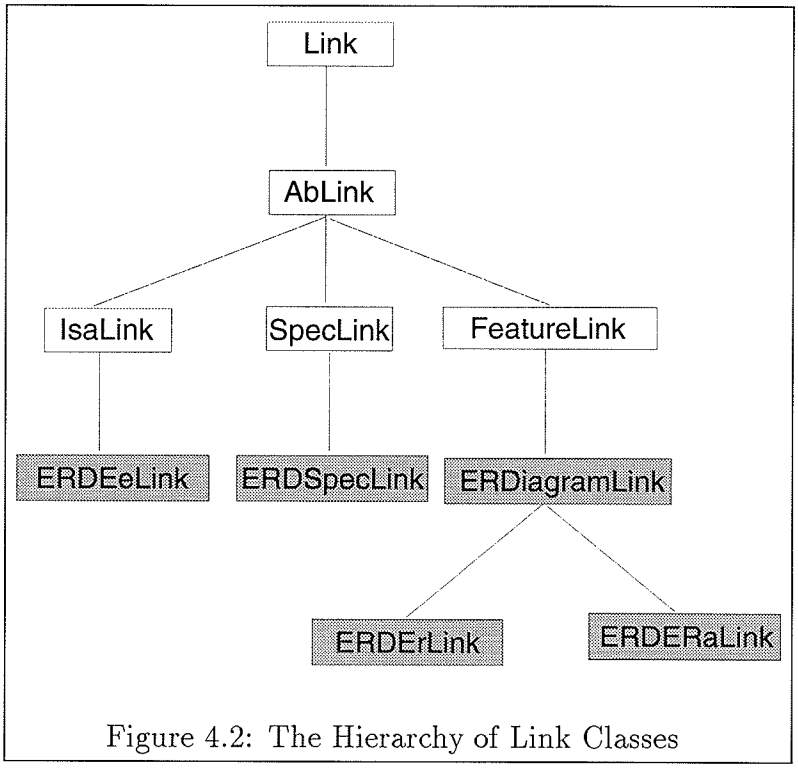

Figure 4.2: The Hierarchy of Link Classes

Figure 4.2 shows the hierarchy of the link classes. In Figure 4.2, *Link* is the top and generic class; *AbLink* contains abductive inference specific information; The *SpecLink* and *IsaLink* represent the generalization and specialization among super/sub nodes. *FeatureLink* represents other relationship between nodes. The links in the obvious abductive reasoning are directed links. The message is sent via the reversed direction of link. The IsaLink is from sub-class to super-class

and the SpecLink is from super-class to sub-class. *ERDEeLink* and *ERDSpecLink* are SpecLink and IsaLink with further E-R diagram details. For instance, entity "customer" is linked to entity "company" via an ERDEelink and it means *customer is-a company.* For each ERDEelink, we automatically attach a ERDSpecLink in the reversed direction, e.g. the company can be "specialized" as a customer. ERDErLink and ERDEraLink are FeatureLink and represent the other relation between entity and attribute, entity and relationship node. We have another hidden but the same type link between entity and relationship nodes. Observations are mostly attributes in database, namely the initial nodes are mostly attribute nodes. Therefore we choose the direction of ERDEraLink to be from the entity to the attribute, and from the relationship to the attribute. The messages sent to the attribute nodes will be forwarded to entity or relationship nodes. However, whatever the direction of ERDErLink(the regular ERDErLink) we choose, the message will either stop at an entity node or a relationship node and will not be forwarded to other entity or relationship nodes. To avoid this, we automatically attach a reversed ERDErLink(the auto-attached ERDErLink) between entity and relationship. This enables messages to be passed in both directions but may cause the message sending never to stop. We can resolve this problem by assigning a filter on the auto-attached ERDErLink. A filter is an examiner who checks the messages. The message itself stores the link via which it came from. When a message is sent via the auto-attached ERDErLink, the filter checks if it was sent via the regular ERDErLink. If it was, the sending is blocked and the message is discarded.

The last class is ERDiagram. It is a specialization of AbNet. Figure 4.3 shows the hierarchy of our graph classes. A *Graph* organizes all nodes and links. It provides node adding, deleting, and searching functionality. *AbNet* has abductive specific members and methods such as retrieving trace. The *lexicon, lexical item*

are held by *ERDiagram*.

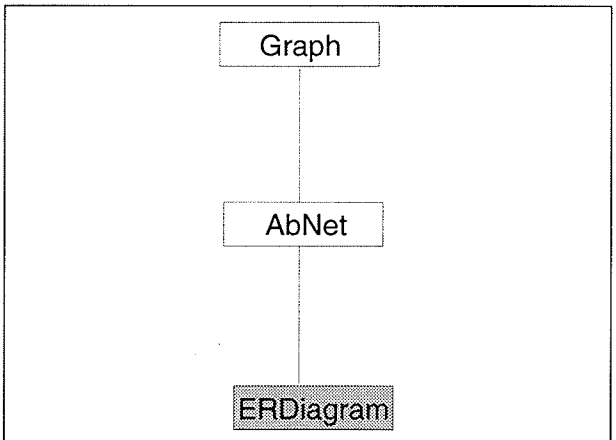

Figure 4.3: The Hierarchy of Graph Classes

Using the above objects, we create the ERDiagram object shown in Figure 4.4 which represents the E-R diagram in Figure 3.2. It becomes a directed graph. We have added those automatically attached links and they are shown with dash lines. Besides the E-R diagram specific labels (e.g. the relationship label 1:N, M:N), each link is assigned a link-id (with exception of ERDEeLink and ERDSpecLink).

4.2 Command Interpreter

In the obvious abductive engine implementation, we consistently use a lisp-like language as the interactive interface, called the *command interpreter*.

Each command is like a list in LISP. The first element in the list is interpreted as a function name, the function is retrieved from the evaluation environment by that name. The rest of the list is passed to the function as arguments. The following classes are implemented to support the command interpreter:

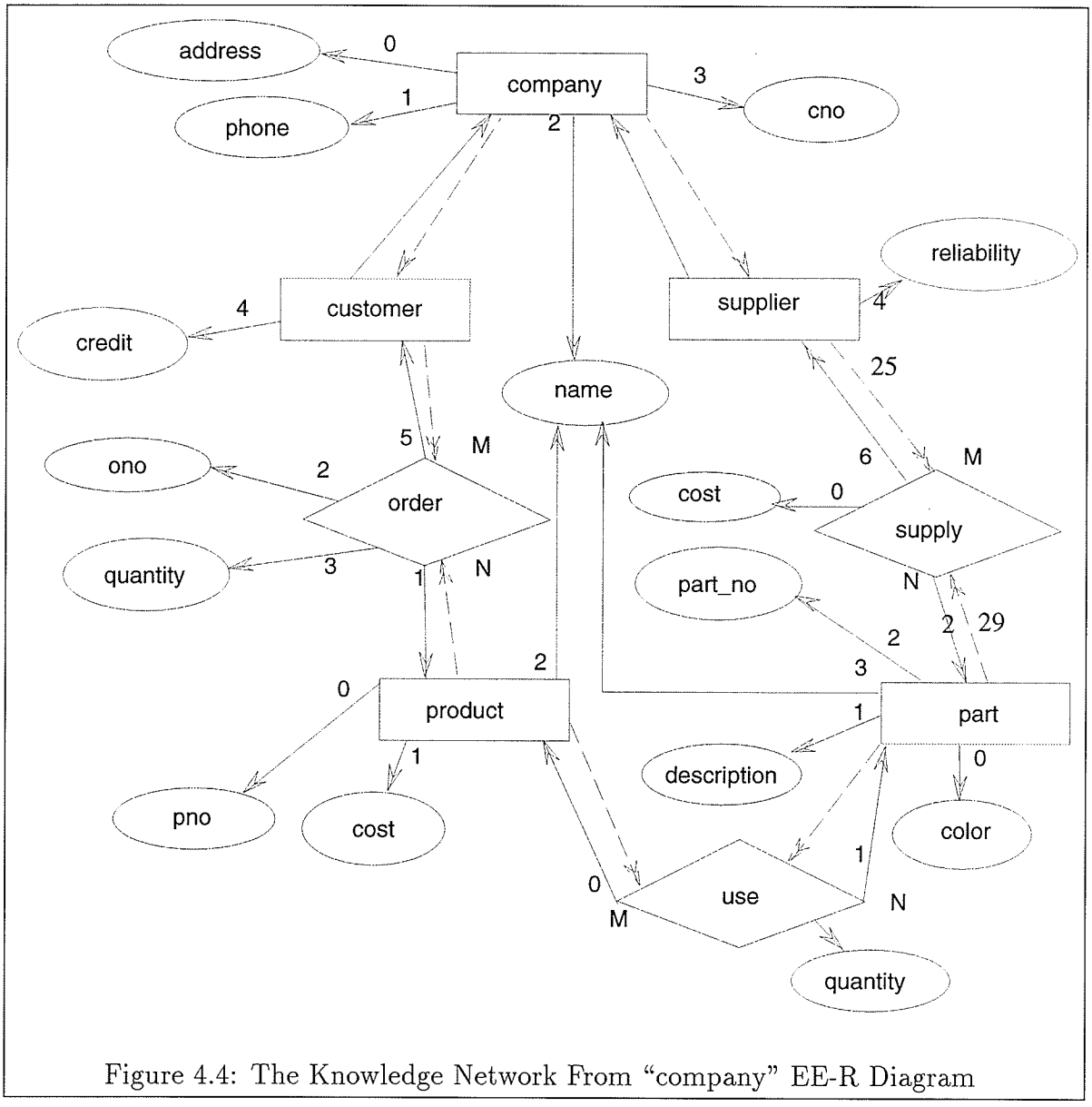

Figure 4.4: The Knowledge Network From "company" EE-R Diagram

- IRSlist stands for Interactive Recursive Slist (Slist is a circular single linked list). IRSlist can be constructed by reading in a lisp-like list. For example, reading the list

(A B (C (D E) F))

will construct the IRSlist shown in Figure 4.5.

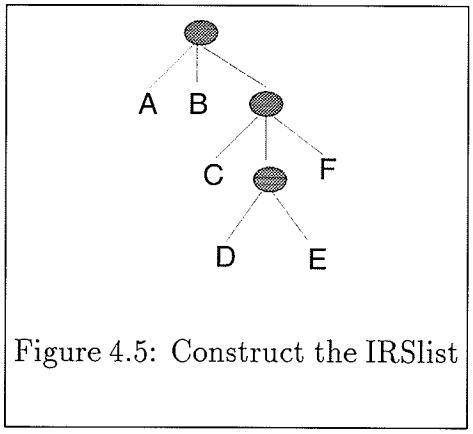

Figure 4.5: Construct the IRSlist

- Once the IRSlist has been read in, it can be evaluated by CmdInterpreter. To evaluate a non-empty list, the first element in the list is treated as a function name. The function is looked up from the CmdInterpreter's evaluation environment. The rest of the list is passed to the function as an argument.

Let variable *shell* be a CmdInterpreter, *args* be an IRSlist*. Following the steps below allows a user to add a new customized command to the system.

1. Create a function to do whatever you want, e.g

```
static Ent def_graph(IRSlist* args, CmdInterpreter& shell)
{
  /* args->get_val() will get and remove an list/atom from args list */
  const char* name = args->get_val();
  const char* type = args->get_val();
  if (name==0) {
    Cerr << "(def-graph <name> <type>): missing name" << endl;
    return 0;
  }
  if (type==0) {
    Cerr << "(def-graph <name> <type>): missing type" << endl;
    return 0;
```

```
    }
    Graph* graph = Graph::make_graph(type);
    ...
}
```

2. Add the function to the evaluation environment:

```
shell.enter("def-graph", FUNC2ENT def_graph);
```

Now function "def_graph" can be evoked with the name "def-graph" in command interpreter.

4.3 Class Prototype

The "def-graph" in the example above will create an instance of class *Graph*. We want to create an instance of class *ERDiagram*. We use a class prototype dictionary to reuse the "def" codes. Class Graph maintains a dictionary which contains the class prototypes. The *key* in each entry is the class type name; the *value* is its default constructor (i.e. without parameter). Class Graph provides method to add a prototype. For example, we add the E-R diagram classes to the dictionary as following:

```
Graph::node_proto(new ERDErNode());
Graph::node_proto(new ERDAttrNode());
Graph::link_proto(new ERDEeLink());
Graph::link_proto(new ERDEraLink());
Graph::link_proto(new ERDErLink());
Graph::graph_proto(new ERDiagram());
```

Each of these classes provides a name-fixed method to create its instance when needed. The names are called *new_node()* for subclass of Node, *new_link()* for subclass of Link, and *new_graph()* for subclass of Graph. We can retrieve the class

prototype by its name in the dictionary and call *new_<whatever>* to create the instance.

4.4 Knowledge network

Using the classes, the class prototype dictionary, and the command interpreter, we can build the E-R diagram knowledge network. The following script demonstrates how to create a network. It is a partial description of the E-R diagram network in Figure 4.4. The complete script is listed in Appendix A.

```
(def-graph customer-supplier ERDiagram
  (graph-size 100)
  (mylexicon-size 500)
  (myweight-threshold 2)
  (max-explanations 100)
)
(def-node company ERDErNode
  (node-alias)
)
(def-node cno ERDAttrNode)
(def-node address ERDAttrNode)
(def-node supplier ERDErNode
  (isa company)
  (node-alias)
)
(def-node supply ERDErNode
  (node-alias)
)
(def-link company
 (address ERDEraLink (link-id 0) (optional-feature))
 (phone ERDEraLink (link-id 1) (optional-feature))
 (name ERDEraLink (link-id 2) (optional-feature))
 (cno ERDEraLink (link-id 3) (optional-feature) (pk))
)
(def-link supply
 (cost ERDEraLink (link-id 0) (optional-feature))
 (part ERDErLink (link-id 2) (optional-feature) (reverse-link))
 (supplier ERDErLink (link-id 6) (optional-feature) (reverse-link))
```

```
)
```

The graph definition takes the following format:

```
(def-graph <graph name> ERDiagram
  <graph attribute list ...>
)
```

As we described above, ERDiagram here is the class type name. According to the class prototype, def-graph will create an instance of class ERDiagram. We have implemented some system limits such as graph size, lexicon size, and so on. Users can set these limits to match their application domain. They are listed as below:

- (graph-size 100) : limit the node number in the network to 100;
- (mylexicon-size 500) : limit the lexicon size we associated with this network to 500;
- (myweight-threshold 2) : set the threshold which we will take to choose or discard the explanations;
- (max-explanations 100) : limit the explanation number we will trace to 100;

Each node appeared in the E-R diagram has to be defined explicitly with name and its type. Node definition are similar to graph definition as shown in the following fragment:

```
(def-node <node name> <node type>
 <node property list ... >
)
```

The parameters listed define the type and properties of a node. The valid parameters are listed below:

- <node name> is node name in an E-R diagram;
- <node type> is the class type name; it can be either ERDErNode or ERDAttrNode;
- <node attribute list > defines properties of a node. Available properties are:
  - (isa <node name>) specifies the super/sub class relationship(also defines an ERDEeLink and an ERDSpecLink).
  - (node-alias) will generate a shortest alias of that node which will be used in SQL statement.
  - (table <table name>) specifies the table name in database if it is different from the entity name in the E-R diagram.

The connections between nodes are defined by "def-link". As mentioned, we have defined "IsA" link and "Specialize" link in node definition and do not need to consider those two kinds of links here. Link definition takes the following format:

```
(def-link <start node>
  (<end node 1> <link type 1> <link attribute list ...>)
  (<end node 2> <link type 2> <link attribute list ...>)
  ...
)
```

Each link starts from <start node> and ends at <end node>. The <start node> and <end node> are the node names in E-R diagram. Each link definition groups all links starting from the same node. <link type> is class type name; It can be either ERDErLink or ERDEraLink. The link attribute list is a list of the following attributes:

- (link-id < *id* >) will assign each link an identifier. Since we use link-id to distinguish each feature link connecting to the same node, each link must have a unique <id>.

- (reverse-link) will add a reversed direction link. This will enable the message to be passed the other way. Each entity and relationship node pair should have (reverse-link) in addition to the normal one.

- (optional-features) will mark the message coming via this link to be *optional*; otherwise it will be a *necessary* feature. In this case the node will wait until all of the necessary links have passed by a message and then to begin to send new message.

- (pk) means the attribute node connected by this link is a *primary key*. Only ERDEraLink should have this feature. Each entity node can have one and only have one primary key.

- (field-name <name>) is the real field name used in the table if it differs from the name in E-R diagram. This is only applicable to ERDEraLink.

4.5 Lexicon

The purpose of lexicons is to map words into their semantic and/or syntactic representations. A lexicon file contains a list of lexical entries, they are written in LISP-like format. The following entries are excerpt from the generic dictionary.

```
(addition
 (syn (N))
)
(additional
 (syn (Adj -cmp))
```

```
)
(additionally
 (syn (Adv))
)
(additive
 (syn (C))
)
(addle
 (syn (Adj -cmp))
 (syn (Tn :opt))
)
(addle-brained
 (syn (Adj -cmp))
)
(bomb
 (syn (C))
 (syn (Tn))
 (phrases (bomb out) (bomb, atom) (bomb, buzz) (bomb, cobalt)
 (bomb, thermonuclear) (bomb, time))
)
```

Each entry has a *key* or a *head word* and a series of meanings attached to it. The meanings of a key can be defined in different ways. For different formats, we just create a different prototype. The lexicon interprets the meaning according to its prototype format. Usually, it is defined by an *attribute vector*, which holds a set of attributes, as shown in the above dictionary. Each of these attributes in an attribute vector can be binary, text string, enumerate, or another attribute vector. In the above example, we have predefined attribute vector called *syn* and made it as the meaning prototype. The system will interpret each key's meaning by *syn* prototype. Since many keys will share some same meanings, we use *alias* in each definition and that will reduce the dictionary size and make it easy to read. In the above lexicon excerpt, *N, Adj, Adv* are all alias. They are defined in a resource file and will be loaded when it is needed. The followings are the definitions of some most frequently used alias. The symbols refer the linguistic concepts. All of them

Attribute	Type	Values	Examples
cat	Enumerate	n, v, a, p, i, c	want: (cat v); good: (cat a); in: (cat p)
plu	Binary	+,-	children: +plu; child: -plu
per	Integer	1,2,3	she: (per 3); you: (per 2); me (per 1)

Figure 4.6: Lexicon Attribute Values

are described in [25].

```
(alias verb (make-av ((cat v) +pg -wh -cap)))
(alias N
       (make-av ((cat n) (nform norm) (theta ag th exp ben goal src instr))))
(alias U ;;
  (make-av (N -ct -plu))
  )
(alias Adj ;;
  (make-av (A -adv)))
(alias Adv ;;
  (make-av ((cat a) +adv -prd)))
(alias C ;;
  (make-av (N +ct))
  )
```

The attribute values hold the syntactic meaning of the key word. Figure 4.6 shows some attributes and their values.

A meaning can also be a key word of a series of phrases and these phrase are in turn the lexical entries. There is a list of phrases in the above entries for *bomb*. The word *bomb* is said to be the head of the phrases. Generally speaking, the head word of a phrase should be the least frequent word in the phrase or one that may undergo morphological changes. When the head word is found to be present in a sentence, the lexicon then checks its neighboring words in the sentence to see whether the phrase is present or not. If it is, one or more lexical items for the phrase are created.

We also have a domain-dependent dictionary. The major advantages of a separate dictionary is the flexibility to configure the interface for different database. We take the same dictionary format as the generic one but the meanings are different and the number of entries is much smaller. We define another attribute vector called *ERDAtts*. The lexicon maps the words in queries to nodes in E-R diagram. The mapping between words and nodes in E-R diagram is a key issue in our system. There are two kinds of entries: *metadata entry* and *value entry*. Metadata refer to entities, attributes, and relationships in E-R diagram. Values, on the other hand, refer to the values in the database. There is a difficulty in value type, namely numeric value. In a database, there are numerous numeric value entries. It is hard to decide the numeric value's domain without considering its neighbor words. It will generate a lot of ambiguity. To avoid this, we restrict the query's domain for numeric values must be explicitly specified. For example, quantity=3, cost<15, etc.

The ERDAtts attribute vector specifies a word meta type or date value type by a binary attribute, called *meta*. For example, in entry "customer", its *erdnode* attribute has value *customer*, which is the node name in the E-R diagram; and it is a meta data denoted by +meta. We list part of the dictionary as follows:

```
(color
 (ERDAtts ((erdnode color) +meta))
)
(colour
 (ERDAtts ((erdnode color) +meta))
)
(company
 (ERDAtts ((erdnode company) +meta))
)
(cost
 (ERDAtts ((erdnode cost) +meta))
)
(credit
 (ERDAtts ((erdnode credit) +meta))
)
```

```
(customer
 (ERDAtts ((erdnode customer) +meta))
)
```

The lexicon file is not loaded into the memory. The system automatically creates a file which contains the hash index to the lexicon file. The lexicon in memory only has the names of all associated lexicon file. The hash index is used to locate a lexicon entry.

A dictionary configuration file is created to make the dictionary more flexible. The dictionary configuration file is a text file. It has a file header which contains the following items:

- dictionary-name : specify the real dictionary file
- index-name : specify the index file
- index-size 70 : the size of dictionary file
- open_bkt (: the symbol used in the dictionary file.
- close_bkt)

We summarize the steps to construct and configure a customized dictionary as following:

1. create an attribute vector;
2. compose a dictionary using the attribute vector;
3. create the dictionary configuration file;
4. use built-in function (make-meaning-proto) to register the attribute vector to the prototype dictionary.

5. use built-in function (link-lex-file <configuration-file>) to install the dictionary.

Obviously, the system can load a few lexicons (limited by the memory resources). Built-in function (link-lex-file) can be used to switch to one of another lexicons.

4.6 Lexical analysis

4.6.1 Attribute Vector

An attribute vector is a set of attributes. Each of the attributes may have different type, such as binary, string, enumerate, or even another attribute vector (recursively).

An attribute vector is implemented as class *AttVec*, which holds all the value and related information. Some operations can be applied to a vector, such as *print, unify, copy, and alias.* The logical comparison operators are also defined so attribute vectors can be compared using equal to (==), greater than (>), and so on.

Each attribute in an attribute vector must have an *attribute interpreter.* The following lists some of the attribute interpreter classes implemented:

- BinAtt: binary attribute.
- EntAtt: enumerate attribute.
- StrAtt: string attribute.
- ListAtt: list attribute.

- HierAtt: Hierarchy attribute.
- VecAtt: attribute vector attribute.

Attribute interpreters are organized in an *attribute interpreter list.* The access and manipulation (e.g. changing, retrieving) are done through the attribute interpreter list. It is implemented as class *AttInterpreterList*, which holds pointers to all attribute interpreters. We can have as many AttInterpreterList as we need but all operations are applied to *current AttInterpreterList*, indicated by class static variable *AttVec::the_itp_list.* We may switch to any AttInterpreterList by setting AttVec::the_itp_list, or by using lisp function (set-interpreter-list). When an attribute vector is processed, we get each of the attribute interpreters through current attribute interpreter list; and then each attribute value can be interpreted by its own interpreter. An example of defining attribute vector interpreter is shown below. It defines an attribute interpreter list *ERDAtts.* The attribute vector it can interpret is an StrAtt, *erdnode* and a BinAtt, *meta.* It is the attribute vector we use in our domain-dependent lexicon.

```
(def-att-list ERDAtts
    (defattribute erdnode StrAtt)
    (defattribute meta BinAtt)
)
(generate-code ERDAtts erdatt.h erdatt.c)
```

The attribute vector defined above has to be accessed by its name. This will slow down the system performance somewhat. An alternative way to access an attribute vector is to generate a C++ source code and bind it in the executable. We then do not have to retrieve the attributes by a name and simply just by referring to a C++ variable. The line (generate-code ERDAtts erdatt.h erdatt.c) in the above example will generate two C files, erdatt.h and erdatt.c. All variables are defined and initialized in those two files.

4.6.2 Lexical Item

The lexical analyzer takes a sentence as input, recognizes the word boundaries, and creates a set of lexical items, which are organized in a *lexical item list.* We maintain one lexical item list to keep all lexical items of a sentence. The lexical item list is cleared to empty before each sentence is processed. Once a sentence is read in and stored as a IRSlist, member function lexical_search() will look up the lexicon and create lexical item for each word.

An original lexical item is similar to its lexical entry in the lexicon and with an additional interval [i,j] denoting the i'th to j'th word in the sentence (in the original item, the i is equal to j). Its meanings are read in from the lexicon as well. If a word has multiple meanings, all of its meanings are read in and corresponding lexical items are created. For example, given the sentence *"Which company supply green nut"*, the lexical analyzer will create the following lexical item list:

[0,0] which (ERDAtts ((erdnode "internode") +meta)) (affixes bare) (root which)

[1,1] company (ERDAtts ((erdnode "company") +meta)) (affixes bare) (root company)

[2,2] supply (ERDAtts ((erdnode "supply") +meta)) (affixes bare) (root supply)

[3,3] green (ERDAtts ((erdnode "color") -meta)) (affixes bare) (root green)

[4,4] nut (ERDAtts ((erdnode "description") -meta)) (affixes bare) (root nut)

During the lexical analysis, abbreviation will be expanded and some lexical rules are applied to recognize numbers, locations, names of persons and organizations that consist of multiple words.

4.6.3 Lexical Rules and Pre-Processor

In the process of member function lexical_search, there is a flag indicating whether to run the pre-processor or not. The flag also can be set by a lisp function (run-pre-process). The preprocess functions is generated by UNIX tool *lex*. It will recognize words like money, date, number, phone number, and so on. It replaces the original word(s) with a list, for example, in format (date Sep., 6, 1994) for an input of date.

Lexical rules can be defined to tell system how to handle those special lists. A lexical rule is defined in the following format:

```
(defrule <rule name>
  <pattern>
  <action>
)
```

The <pattern> is similar to generic regular expression but in LISP format. It may include logical operators such as "and", "or", and "not". <action> may be "assert" to generate a specified lexical item, or "delete" to ignore the word(s). Each defined lexical rule is stored in the lexical item list and applied to the sentence after the pre-process. Under most circumstances the special list is treated as a noun. The following lexical rules will treat <money>, <time>, <phone number> as noun.

```
(defrule money
  "<money>.*"
  (assert (low) (high) (const (avm ((cat n) (nform norm))))))

(defrule time
  "<time>.*"
  (assert (low) (high) (const (avm ((cat n) +adv (nform norm))))))

(defrule number
  "<number>.*"
  (assert (low) (high) (const (avm ((cat n) (nform norm))))))
```

Corporation name, personal name usually contain multiple words and match some patterns, for example, capitalization, title, corporation designator. They may also be handled by appropriate lexical rules. The following code fragment give the rules to recognize corporation names, personal names, and location names.

```
(defrule corp-name-recognition
  (seq
   (+ [A-Z])
   (? &)
   (? and)
   (* [A-Z])
   (? ,)
   (att (contain +corpdesig)))
  (prog
   (assert (low ($ 1)) (high ($ 6)) (const (avm ((cat n) +pn (nform norm)))))
   (delete-smaller (low ($ 1)) (high ($ 6)))))

(defrule person-name-recognition1
  (seq
   (att (contain +title))
   (+ [A-Z])
   (? (root Jr.)))
  (if (not (is-dict-entry (low ($ 1)) (high ($ 3))))
      (prog
       (assert
(low ($ 1)) (high ($ 3)) (const (avm ((cat n) +pn (nform norm)))))
       (delete-smaller (low ($ 1)) (high ($ 3))))))

(defrule location-recognition
  (seq
   (+ [A-Z])
   (? ,)
   (or (att (contain +province))
       (att (contain +country))))
  (prog
   (assert (low ($ 1)) (high ($ 3)) (const (avm ((cat n) +pn (nform norm)))))
   (delete-smaller (low ($ 1)) (high ($ 3)))))
```

The $ sign used in the example is like in UNIX shell script to indicate the argument position. The Lexical rules can also handle word variations. The following

rules are examples that handle words ended with -ed or -er or -est.

```
(defrule ending-ed
  (and (or (ending -ed)
   (ending -ing))
       (not (att (contain (cat v)))))
  (delete (litem)))

(defrule ending-er-est
  (and (or (ending -est)
   (ending -er))
       (not (att (contain (cat a)))))
  (delete (litem)))
```

4.7 Message passing

4.7.1 Item and Initial Message

After lexical analysis, all lexical item stored in the lexical item list. There are two kinds of words, one of which is not important to database access, such as "what is/are", "Display/list", which can be simply ignored in finding query graph; the other is the names or values of entities in database, which are corresponding to node in E-R diagram. Class ERDiagram's member function find_node() takes each lexical item as parameter and maps it to the nodes in E-R Diagram according to its meanings. If a NULL node is returned, this item is ignored. If a real node is returned, the initial messages are prepared based on its lexical items.

A message is an instance of class ERDItem. Figure 4.7 shows the class hierarchy of ERDItem. The ERDItem is the entity passed by abductive engine. Class Item is the superclass of all types of items used in the message passing algorithm for obvious abduction. F-component is a set of features; O-component is the subset of the observations to be explained; Class Item does not contain any information

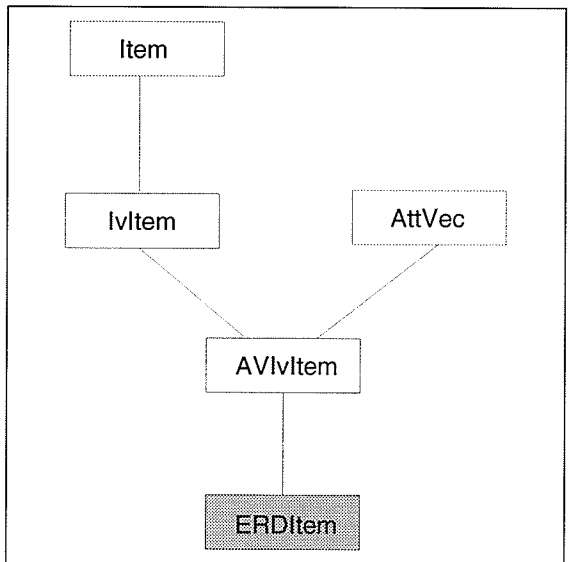

Figure 4.7: The Hierarchy of Class Item

about how the F-component, O-component and attribute values in an item should be represented. Rather, this abstract class defines the structure between items and a set of flag bits (by class SBitSet) that marks the type of the item. There are 5 types of items.

- COMPOSITE_ITEM is one created by unifying two other items.
- FEATURE_ITEM is an item that is received by the node via a feature link. The F-component of a feature item is guaranteed to be a singleton set.
- SPEC_ITEM and ISA_ITEM are items that arrived at a node via a specialization and isa link respectively.
- INITIAL_ITEM is sent to a node externally to initiate a message passing process.

- UNKNOWN_ITEM is the item not falling into the above categories. However, there should never be any UNKNOWN_ITEM.

An item contains two pointers. Depending on the type of the item, these pointers are interpreted differently as: head(), tail(), prev(), channnel();

A composite item is a unification of an item whose F-component is a singleton set and another item. The former is called head(), and the latter is called tail(), head() and tail() are not defined (nil) for other types of items.

The functions prev() and channel() are defined only for items that are received by a node via an link, i.e., items of type FEATURE_ITEM, ISA_ITEM, or SPEC_ITEM. Function prev() means previous item. Function channel() is the link via which this item arrived.

The SBitSet that is used to store item type also provides marker bit during the reclamation of storage used by the items. Because an item store does not necessarily keeps a list of all the items at a node and an item may be used to unify with multiple other items, this marker bit is necessary to avoid reclaiming the memory more than once.

One of the most important member function class Item provides is *combination_of()*, which will combine two items and set the COMPOSITE_ITEM mark.

Class IvItem is an Item with the following additional fields:

- Low
- High
- Source

When *low* and *high* in an item are set, they denote the *low'th* word to *high'th* word in the sentence. An initial item has value *low* equal to *high*. *Source* is used to record where the item came from; it records the link's $< id >$ via which the item is passed. This is only true for feature link, not for IsA and or SpecLink. The *source* is implemented as a 32-bit mask; that means it can only record link-id from 0 to 31. The link-id assigned to links which are connecting super/subclass nodes should be taken into special consideration depending on the application. Generally, the links connecting to a subclass are also considered connected to its superclass. For applications which distinguish the links connecting to node from those to its super-node, different link-ids should be given to them. For example, link-ids 0-5 are given to links connecting to node A. If node B is A's sub-node, link-ids 0-5 should not be used to label links connecting to node B. If node A and node B do not have super/sub-node relationship, there is no restrictions at all.

Class AvIvItem is both IvItem and an AttVec. It has features of both AvIvItem and AttVec. By assigning an attribute vector to an AvIvItem, the message is able to carry any kind of information to meet the needs in different applications.

Class ERDItem has two new member:

- root: copies the root word from the lexical item
- via_link: keeps the link-id via which the item is passed. It is different from the *source* member in class IvItem. *Source* is a set of links which remember how the item is formed. *via_link* remembers only one link-id via which the item was passed. The link-id is used in the filter to block the same item passed back.

The initial messages are the items whose attribute vector's values are copied from the meanings of the lexical items. They are sent to the corresponding nodes

in E-R diagram. In some cases, we may send different items to the same node. For example, in query:

List the product names using p1?

The word "name" and part name "p1" will both be mapped to node name. The first item being sent to name is an item with attribute metavalue; the second item being sent to name is an item with attribute datavalue (p1).

4.7.2 Message Passing

When a node receives an item, it checks the item against a set of constraints. The following pseudo code fragment demonstrates how it works.

```
void AbNode::receive(Item* item, FeatureLink* l)
{
  if (item not valid)
  {
    return;
  }
  store item;

  if (item is combinable && is complete)
  {
    store item as completed item list;
  }
  send item over all isa links;
  combine item with items in store;
  for each item in competed item list
  {
    send item over feature links;
    send item over spec links;
  }
}
```

- Validity check. An item is a valid one if it satisfies the local constraints at the node; Note that the contents in item may be changed as a result of enforcing

local constraints. if the item is invalid, it will be discarded.

- Store item. The item is stored into the node's memory. Only the new item will be stored. Class IdNode allocates the storage to store all the eligible items. The storage is called item *buckets*. For efficient access, items are organized according to their *low* and *high* values. For example, buckets[0] stores all items with high = 0, buckets[2] stores items with high = 2, and so on.

- Completeness Check. An item is complete if its F-component contains all the necessary features of the node. Class IvNode defines necessary features. Necessary feature indicates that there have to be items coming from those feature links. If one of them is missing, the item will not be sent out.

- Combinability check. An item is not combinable if it is incomplete and will not be able to be combined with items received by the node in the future.

- Item Combination. It will looks up the item store and combine the newly arrived item with all the items in the store which are compatible with it.

  The newly arrived item (named B) must be a singleton item. Item A is an item picked from the item store. There are three possible return values that member function *compatible(A, B)* will produce:

  EQUAL_TO, DISJOINT, INTERSECT.

  Basically, the returned value are calculated based on the *low* and *high* member of item A and B. When the two items are adjacent, i.e. the *low* of one item is equal to the *high + 1* of the other item, it returns EQUAL_TO. If the two ranges are intersected, it returns INTERSECT. The DISJOINT is the otherwise case. If the value is EQUAL_TO, then A and B are compatible. Otherwise, A and B are incompatible. However, if the value is DISJOINT,

then A and B may become compatible by adding more elements in their o-components or f-components. If compatible(A, B) is INTERSECT, then A and B can only become compatible by removing some elements in their o-components or f-components.

To integrate syntax constraints into the message passing process, we implemented an additional compatible criteria. That means item adjacency is no longer the only criteria. In addition, the two items must represent a span in the parse tree.

4.7.3 Message Filter

Each link in the network has a filter attached. A filter will do the following checks during the passing process:

- Block the items which are not allowed to pass. For ERDErLink, the items must be able to be passed two ways between an entity node and a relationship node. We add a reversed ERDErLink to the pair of nodes besides the normal ERDErlink. We label the invisible link with reverse-id = 32 - id; id is the normal ERDErlink id. (If the a node has more than 32 links attached to it, we would have a problem. In that case, the node should be decomposed into separated nodes) The filter will block items which pass this link and are sent back via the reversed link or an infinite loop occurs. If an item is allowed to pass, then the item is duplicated and sent to the destinations.

- If an item is a FEATURE_ITEM and is passed across a feature link, it is filtered by the link. The f-component of the duplicated item after filtering is set to the link-id.

- Recalculate the weights of items. Since we are finding the shortest connections, the items will carry the connection length as its weight. When an item passes across a link, its weight will increase by the weight of the link (normally the weight is 1).

4.8 Message Trace

The message passing will stop when there is no message sent out. Each node has a bucket of items in store. Some of them contain items whose O-component cover all of the observations. Since we stored the items with their sources, we can trace back all nodes from which the items came. The process is called *message trace.*

The tracing steps are described below:

1. Find the items which cover all of the observations. The ERDiagram member function *max_items()* will search each nodes in *end node* list. A end node is a node at which the tracing is started. Entity and relationship nodes should be end nodes. An attribute node is not a worthy starting point because there is only one initial message. We put all entity and relationship nodes in the *end node* list. Note we can put any kinds of node into the list. A small end node list will speed up the searching. The lisp function (end-nodes) can be used to add nodes into the list.

   Each node has member function *max_item()* which extracts the *complete* items from its item store and stores them to a max_item list. Since the items are stored in the item buckets by their *high boundary*, we need only check the items in the highest bucket. The comparison is done by member function *consider_as_max_items(max_items)*. To decide if an item is a max_item, it

compares the O-component of the item with the items already in max_items. There are three possible results:

- EQUAL_TO: find the matching node-item pair (all of the end nodes' max items are stored in max_items so we need node-item pair to distinguish them) and adjust its weight. If the item is new then insert it into max_items list.
- LESS_THAN: forget the new item and the max_items remains unchanged.
- GREATER_THAN: insert the new item and discard the old item from the max_items list.

When max_item is returned, we get a list which holds all the end nodes with their maximum items. Figure 4.8 shows the maximum item list and its structure.

2. Find the the items with minimum weight in the max_items list and create a *trace point* for each of them.

   We go through the maximum item list and calculate the minimum weight. The weight will be the length (link numbers) of the final shortest connection. We might use a threshold here to determine whether or not to create the trace point for an item. The threshold can be defined by a lisp function (threshold <threshold>). All the items which has weight less than or equal to (minimum weight + <threshold>) will lead to a trace point.

   A *trace point* (class TracePoint) provides information about how an item is combined from a set of singleton items at a node. Trace points are organized as a *trace bush*. A hash table is built to speed up the search of trace point.

   Trace points are created by tracing items on each node. The member function of a node:

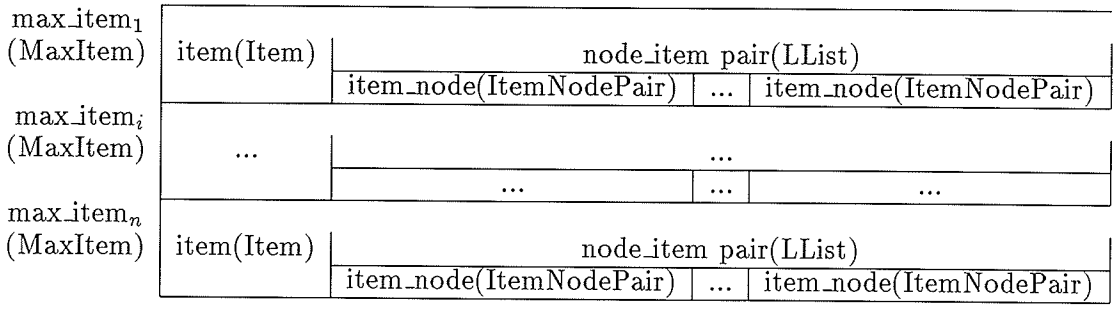

Figure 4.8: Maximum Item(Slist)

AbNode::trace(item, trace_bush)

will create trace points for item. The item was sent out by this node. It traces how the item was formed at this node and creates one or more trace points to represent the composition of the item and then enters the trace points into trace_bush.

The tracing function first searches for the trace point with the pair of item and node. If the trace point does not exist, it first calls *matching_item()* to find all of the items with the same observations and same weight; and then creates a trace point with matching items as arguments. The trace point is inserted in the trace bush. During creation of a trace point, the trace function goes through all of the matching items; for each matching item, extract its singleton items; if the singleton item is passed by another node over a feature link, then the node is traced recursively. The trace function saves all of the singleton item and corresponding trace point in a storage called item sources(class ItemSource).

Figure 4.9 shows the structure of trace bush and trace point.

tp_1(TracePoint)	node(AbNode)	item(Item)	mathing items(Slist)		
			item_source(ItemSource)	...	item_source
tp_i(TracePoint)	...	...	...		
			...	...	...
tp_n(TracePoint)	node(AbNode)	item(Item)	mathing items(Slist)		
			item_source(ItemSource)	...	item_source

$item\_source_1$ (ItemSource)	item(Item)	link(AbLink)	tp, single item pair(Slist)		
			tp_singleton(AssocPair)	...	tp_singleton
$item\_source_i$ (ItemSource)	...	...	...		
			...	...	...
$item\_source_n$ (ItemSource)	item(Item)	link(AbLink)	tp, single item pair(Slist)		
			tp_singleton(AssocPair)	...	tp_singleton

Figure 4.9: Trace Bush(TraceBush) and Trace Point (TracePoint)

4.9 Explanation

After constructing the trace bush filled with trace points, we say all interested items are traced. The next step is to explain the trace points. The idea of explanation is that when we explain a trace point on a node, we will reach a series of nodes. These nodes are connected by the links (channel). A set of nodes and links which connect the nodes is called an *explanation.* We will use class ERDExplanation to represent each explanation. Since we traced the items with minimum weights, the connections are the shortest connections, i.e Steiner tree for the given E-R diagram. It is also the query graph for given observations. The class hierarchy of ERDExplanation is shown in Figure 4.10.

The tree structure in this thesis is based on basic classes Slist and RSlist, called RSTree. It provides the generic tree operations such as pre-order, post-order, and breadth-first traversal. Other tree operations are provided by RSlist, such as insertion and deletion.

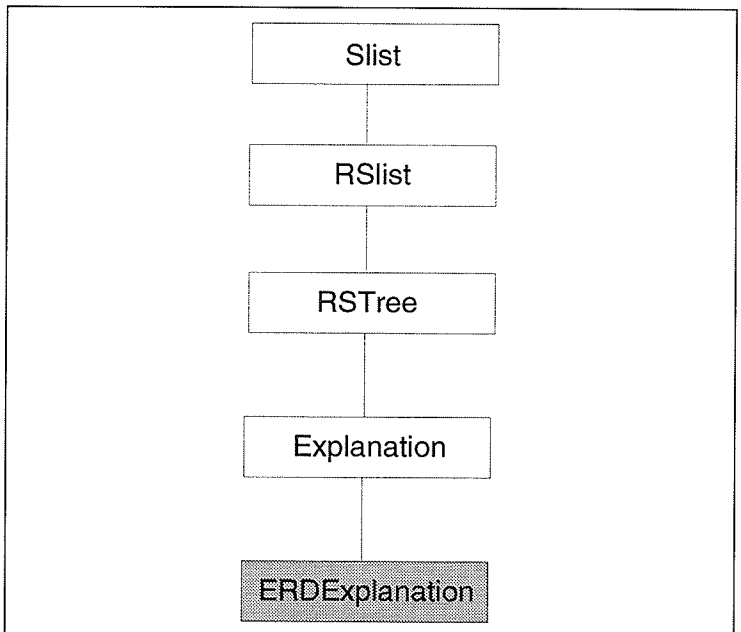

Figure 4.10: The Hierarchy of Class ERDExplanation

An Explanation is a tree. A node in the tree has pointers to the nodes, links in E-R diagram the item, and the trace point. It also records the length of the tree. An explanation tree is not exactly a subgraph of the E-R diagram. Its node is an instance of the node in E-R diagram. The word "node" we are using in explanation tree and in E-R diagram is different. In E-R diagram it is a user defined object but in explanation it is just a logical unit to organize data. We refer the nodes in explanation tree to *E-Node* to distinguish them from nodes in E-R diagram.

The explaining process starts from function *TraceBush::explanation*. Each trace point in the bush is explained and an explanation is created. A set of explanations are created after all trace points are explained. The trace point is explained in the steps below:

TracePoint::explain()

1. Create an initial explanation E-Node with current item in item source.
2. if the item is an INITIAL_ITEM, return the E-Node.
3. for each trace point under current item, recursively call TracePoint::explain. Insert the returned E-Node into the explanation tree.
4. for each singleton item under current item, if it is INITIAL_ITEM, create a new E-Node and insert in current tree.

During the explaining process, we must adjust the length of the tree at each stage. The last step above is not necessary for some domain applications. It is there to maintain consistency. We will trim the tree to fit the application domain.

There are two differences between the tree we want and the generic tree abductive inference engine generates. The first is in the generic tree each E-Node has itself as its first child. For example, E-Nodes "company" and "customer" in top portion of Figure 4.11 have themselves as their first child respectively. The second one is that, E-Node "customer" is connected to "company" via an ERDEeLink (i.e IsA link), then the nodes connecting to "company" such as "address" and "phone" are treated as connecting to "customer" in passing algorithms. These assumptions make sense under most circumstances. To simplify the SQL translating process, we want to find the original node (i.e the superclass node). In the above example, we will move "address" and "phone" from "customer" to "company".

The trimming process is done by ERDExplantion::trim(). It checks the child and its parent E-Node; if they are both pointing to the same node, then the child is deleted; if the child-parent relationship is not the same as they are in E-R diagram, it goes up along its ancestors until an ancestor is found who takes the child as its child; the child is moved to its "real" parent. Everything is copied during the

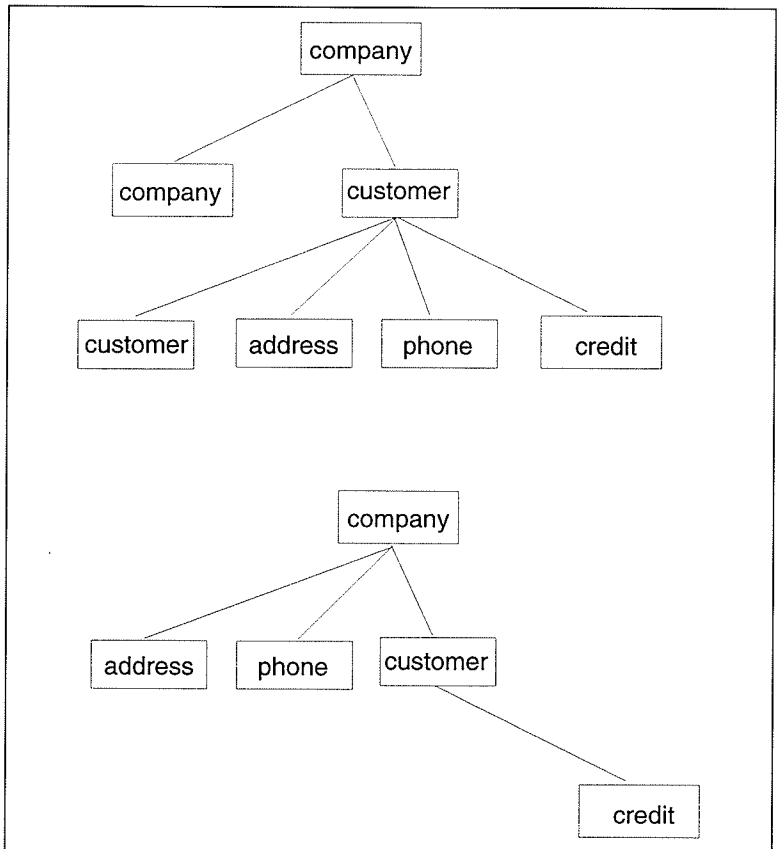

Figure 4.11: The Trimming of Generic Explanation Tree

moving process to ensure that the explanation is still correct. The bottom portion of Figure 4.11 is the trimmed explanation tree.

4.10 SQL Translator

Once explanations are generated and trimmed, we are ready to construct the queries in formal query language – SQL. One explanation is one query graph. Each node in the query graph has some pointers associating itself with the E-R diagram. They

are: a node pointer to the node in E-R diagram; a link pointer to the link in E-R diagram via which the nodes are connected; and an item pointer the lexical item from user, either metadata or data-value.

Different E-Nodes are likely instantiated from the same node in E-R diagram. For instance, every E-Node relating to "name" is instantiated from node Name in E-R diagram. Another relatively complicated circumstance of multiply instantiation is that one entity node in E-R diagram participates in a query more than once but as different roles. Consider the query:

What is the name of John's supervisor?

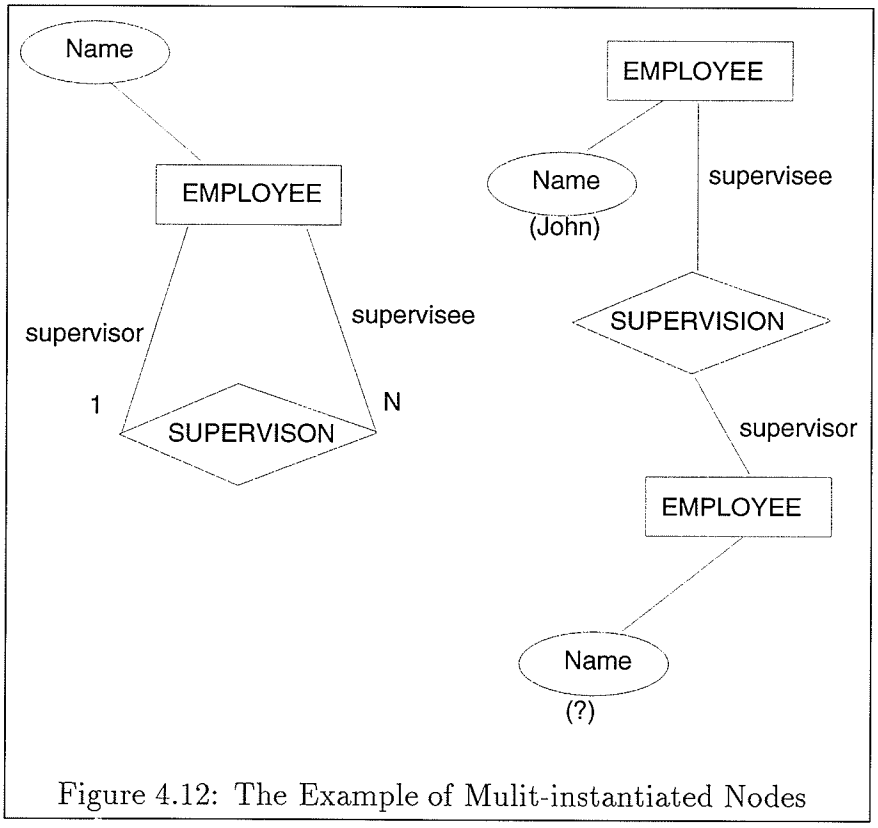

Figure 4.12: The Example of Mulit-instantiated Nodes

Its query graph is shown in Figure 4.12. The left part in Figure 4.12 is the related portion in E-R diagram and the right part is the query graph. We use

the same notations in the query graph as in E-R diagram and the labels are their corresponding node names in E-R diagram. Both node Name and node Employee are instantiated twice. In the query, both John and his supervisor are an Employee so that it is instantiated twice, one with attribute Name "John", one with the name being a variable.

However, when an entity node participates in a relationship more than once, the links their E-Nodes refer to are different. There must be more than one links and it is ambiguous to pick. There is another query graph which is the same as the one in Figure 4.12 but with the label "supervisee" and "supervisor" switched. The system lists both query graphs to user and lets the user resolve the ambiguity.

To simplify the translating algorithm, we give each E-Node an alias. These alias are derived from its corresponding table name. In the case of multiple entity instances, we give them different aliases. This work is done in the explanation tree trimming procedure.

The translator first allocates three buffers with name SELECT, FROM, and WHERE respectively. Semantic strings will be appended to those buffers during the translating procedure. Some functions used in the procedure are:

- *Alias* returns an entity's alias;
- *FieldName* returns an attribute node's field name in database schema;
- *TableName* returns an entity or a relationship's relation (table, view) name in database schema;
- *FieldValue* returns the value in its lexical item if it is a DATAVALUE type;
- *FieldOp* returns the logical operation in lexical item if it is a DATAVALUE though in most cases it is "equal to";

- *Key* returns the primary key of a given node.

Here is the translation algorithm:

```
clear SELECT, FROM, and WHERE buffers
for (each E-Node in the explanation tree)
{
  let Entity = current E-Node;
  let Link = link in E-R diagram;
  if (Link is NULL)
     shift to next node; /* must be the root of explanation tree */
   let Node = the node the E-Node instantiated from;
   let HeadNode = the head node of Link;
   let TailNode = the tail node of Link;
   switch (Link)
   {
    case ERDEraLink:
    /* find the corresponding node of current E-Node to be entity
       of attribute */
       if (Node = HeadNode)
       {
          Entity = parent of E-Node;
          Attribute = E-Node;
       }
       else
       {
          Entity = E-Node;
          Attribute = parent of E-Node;
       }
       append Alias(Entity).FieldName(Attribute) to SELECT;
       /* aggregation functions handled here */
       /* select sum of employee.salary ? */

       if (Entity is not processed)
       {
          append TableName(Entity) Alias(Entity) to FROM;
          mark Entity processed
       }
       if (Attribute is DATAVALUE)
       {
          if (WHERE buffer is not empty)
              append ''AND'' to WHERE;
           append Alias(Entity).FieldName(Attribute);
```

```
                FieldOp(Attribute) FieldValue(Attrubute) to WHERE;
        }
        break;
    case ERDErLink:
    case ERDEeLink:
        Entity = E-Node;
        if (Entity is not processed)
        {
            append TableName(Entity) Alias(Entity) to FROM;
            mark Entity processed;
        }
        Entity = parent E-Node;
        if (Entity is not processed)
        {
            append TableName(Entity) Alias(Entity) to FROM;
            mark Entity processed;
        }
        /* now the join condition */
        if (WHERE buffer is not empty)
            append ''AND'' to WHERE;
        append Alias(E-Node) ''.'';
        if (Node = Tail and Node is a relationship)
            append RoleName(Link) to WHERE;
        append Key(Head) ''='' Alias(parent of E-Node) ''.'';
        if (Node = Head) and (parent of E-Node's Node is a relationship)
            append RoleName(Link) to WHERE;
        append Key(Head) WHERE;
}
```

SELECT statement is formed by listing each attribute. We also handle some practical issues not mentioned in the algorithm. For example, we can set options, ALWAYS_LIST_KEY, ALWAYS_LIST_NAME which always list the primary key attributes and name attribute if they exist, respectively.

FROM statement is formed by appending each table instance. We mark the entity to ensure the instance will be listed only once. Each table instance is in the form of

<table_name instance_name>

It is to resolve the multiple instances of entity node. SQL statements of query in 4.12 are:

```
Select e1.name, e2.name
From   Employee e1 Employee e2
```

WHERE condition is formed when there is an attribute with DATAVALUE, or an ERDErLink (i.e relationship node involved), or an ERDEeLink (super-class and sub-class). All EQUJOINs are concatenated with AND as the connector. It is obvious the query has to satisfy all of the conditions we put in the WHERE buffer. In the DATAVALUE case, we just append the $< attribute = datavalue >$ to the WHERE buffer. In the ERDEeLink case, since the super entity and the sub-entity always have the same primary key, the EQUJOIN is simply formed as <super_entity.key = sub_entity.key>. In the ERDErLink case, based on the mapping algorithm in Section 3.5.5, we know the corresponding relationship node at least has two foreign keys. The table is created by SELECTing the two primary keys of each participating entities. The foreign keys keep the same name as the primary keys if there is no rolename. We combine role name and primary key respectively as the new field name if there are role names. The role name here will resolve the conflict in the case of that one of entities participates in a relationship twice. The EQUJOIN is

<entity1.key1 = relation.rolename_key1 AND entity2.key2 = relation.rolename_key2.>

The complete SQL statements of query in Fig. 4.12 are:

```
Select e1.name, e2.name
From   Employee e1 Employee e2 Supervision s
Where  e1.name= ''John'' AND e1.ssn = s.supervisee_ssn AND
       e2.ssn = s.supervisor_ssn
```

Chapter 5

Conclusions and Future Research

5.1 Conclusions

We have described a user interface to databases which uses *obvious abductive reasoning* to understand queries in natural language. The task of understanding is viewed as finding the structural relationships between unstructured inputs. The domain knowledge is represented as an E-R diagram for database. Queries are parsed syntactically and database related words in queries are mapped to nodes in the network, called observations. Observations are annotated with a set of attributes extracted from generic lexicon and domain-specific lexicon. An explanation of the observations is a generalized subtree of the network that connects all the observations. This connection is a coherent set of relationships between the observations; consequently, it explains how they are related and forms a query graph. The query graphs are translated into formal structural query language(SQL) and submitted to the DBMS.

The parsing of queries and finding of shortest connections are based on a message

passing algorithm. It is an instantiation of an efficient generic message passing algorithm for abductive inference. The object-oriented nature of the algorithm not only lends itself to distributed parallel processing, but it also facilitates extension and adaptation of the algorithm in different application domains.

5.2 Future Research

Future research consists of continued development of the experimental system and further investigation of the natural language user interface to database. We will highlight some areas of NLI research that have been less explored, namely database update, meta-knowledge question, temporal questions, and multi-modal interfaces.

5.2.1 Graphical Domain Knowledge Editor

The domain knowledge is the E-R diagram with some constraints. Since the size of E-R diagram is limited, it is acceptable to build the network by hand. One alternative is to use *graphical user interface* to construct the knowledge network. For example, the network can be constructed, and modified by direct manipulation on visual graphical objects. There will still be an internal representation of network but invisible to the users. One difference of the E-R diagram graphical editor from other graphical editors is that the constraints must be enforced during the interactions. For example, the editor will deny the attempt to link two attribute nodes. The graphical user interface must guarantee the E-R diagram to be consistent with the internal representations. We call the editor a *semantic-constrained E-R diagram editor.*

5.2.2 Domain Dependent Lexicon

For each different database, there is a different domain dependent lexicon. The lexicon depends on the symbol names in E-R diagram and the question type against the database. We need to find a better way to construct the lexicon automatically or semi-automatically though manual construction is acceptable since E-R diagrams are generally not very big. This work will involve knowledge acquisition, system training, etc.

5.2.3 Integration with Expert Systems

In some cases it may not be possible to answer a natural language question, although all the necessary raw data are present in database. Questions involving common sense or domain expertise are typical examples. For example, assume that a database holds the medical records of all patients in a hospital. It is hard to answer the query:

"Which patients are likely to develop lung disease?"

In these cases, the answer to the question is not explicitly stored in the database; to produce the answer, the NLI must be able to carry out reasoning based on the data stored in the database.

The reasoning module (i.e. the expert system) should not be considered part of the NLI, but rather an independent layer on top of the database system, since it may be needed by other components of an intelligent DB system.

5.2.4 Intelligent Response Generation

User requests often do not express literally what the user wants to know. The NLI should also report additional information that the user would probably be interested to know. The NLI must be able to understand the user's goal(This is known as "plan recognition" in the literature). This additional information could be achieved by having another reasoning module to find the user's goal. The NLI will explain how the user's requests relate to his/her goals, and what to do to satisfy the user's goal. The user's request would be mapped to a logical query, and the logical query would be passed to the reasoning module. The reasoning module would then try to satisfy the user by reasoning about his/her goals, and by retrieving information from the database.

5.2.5 Database Update

Few systems support natural language requests to update the underlying database because database updates by natural language introduce several new problems. Often, natural language update requests cannot be satisfied, or they are ambiguous, or they lead to unanticipated side-effects, due to database constraints of which the user is not aware. The following scenario illustrates an important weakness:

List the employees and their managers.

Employee	Manager
Brown	Jones
Smith	Jones

Change Brown's manager from Jones to Baker.

List the employees and their managers.

```
Employee        Manager
Brown           Baker
Smith           Baker
```

When Brown's manager is changed, the Smith's manager also is changed from Jones to Baker, although this was not what was originally requested. To a user unaware of the database's structure, such a behavior would appear erratic. The user does not know that in the database employees are linked to managers thought their department.

5.2.6 Meta-knowledge Questions

Meta-knowledge question are questions referring to knowledge about knowledge. For example, some meta-knowledge questions could be:

what information is in the database?

what are the possible employee job titles?

5.2.7 Temporal Questions

Most NLI were designed to interface to snapshot database. Snapshot databases only store information about one state of the world, usually taken to be the "present" state. Consequently, most NLI only support questions referring to the present state of the world. For example, a NLI to a company's database would typically be able to answer questions like:

who is the manager of the sales department?

but would usually not support temporal questions referring to the past:

who was the previous manager of the sales department?

In the last decade, computer scientists have become increasingly interested in temporal database systems, i.e database system designed to store information about previous or future states of the world and to generally support the notion of time.

5.2.8 Multi-modal interfaces

As discussed in Chapter 2, form-based interfaces and graphical interfaces appear to have advantages over natural language interfaces. Some system attempt to merge natural language with graphics, menus, and forms to combine the strengths of all modalities.

Appendix A

The Description of E-R Diagram

The following the the complete descriptioin script of "company" EE-R diagram:

```
(def-graph customer-supplier ERDiagram
  (graph-size 100)
  (mylexicon-size 500)
  (myweight-threshold 2)
  (max-explanations 100)
)
(def-node company ERDErNode
  (node-alias)
)
(def-node cno ERDAttrNode)
(def-node address ERDAttrNode)
(def-node phone ERDAttrNode)
(def-node customer ERDErNode
  (isa company)
  (node-alias)
)
(def-node supplier ERDErNode
  (isa company)
  (node-alias)
)
(def-node reliability ERDAttrNode)
(def-node credit ERDAttrNode)
(def-node product ERDErNode
  (node-alias)
)
(def-node order ERDErNode
  (node-alias)
```

```
)
(def-node ono ERDAttrNode)
(def-node pno ERDAttrNode)
(def-node part ERDErNode
  (node-alias)
)
(def-node color ERDAttrNode)
(def-node description ERDAttrNode)
(def-node part_no ERDAttrNode)
(def-node use ERDErNode
  (node-alias)
)
(def-node supply ERDErNode
  (node-alias)
)
(def-node quantity ERDAttrNode)
(def-node cost ERDAttrNode)
(def-node name ERDAttrNode key)
(def-link company
 (address ERDEraLink (link-id 0) (optional-feature))
 (phone ERDEraLink (link-id 1) (optional-feature))
 (name ERDEraLink (link-id 2) (optional-feature))
 (cno ERDEraLink (link-id 3) (optional-feature) (pk))
)
(def-link customer
 (credit ERDEraLink (link-id 4) (optional-feature))
)
(def-link supplier
 (reliability ERDEraLink (link-id 4) (optional-feature))
)
(def-link order
 (customer ERDErLink (link-id 5) (optional-feature) (reverse-link))
 (product ERDErLink (link-id 1) (optional-feature) (reverse-link))
 (ono ERDEraLink (link-id 2) (optional-feature) )
 (quantity ERDEraLink (link-id 3) (optional-feature))
)
(def-link product
 (pno ERDEraLink (link-id 0) (optional-feature) (pk))
 (cost ERDEraLink (link-id 1) (optional-feature))
 (name ERDEraLink (link-id 2) (optional-feature))
)
(def-link part
 (color ERDEraLink (link-id 0) (optional-feature))
```

```
 (description ERDEraLink (link-id 1) (optional-feature))
 (part_no ERDEraLink (link-id 2) (optional-feature) (pk))
 (name ERDEraLink (link-id 3) (optional-feature))
)
(def-link use
 (product ERDErLink (link-id 0) (optional-feature) (reverse-link))
 (part ERDErLink (link-id 1) (optional-feature) (reverse-link))
 (quantity ERDEraLink (link-id 2) (optional-feature))
)
(def-link supply
 (cost ERDEraLink (link-id 0) (optional-feature))
 (part ERDErLink (link-id 2) (optional-feature) (reverse-link))
 (supplier ERDErLink (link-id 6) (optional-feature) (reverse-link))
)
```

Appendix B

The Parse Tree

The following is the completd parse tree generated by the principle based parser of sentance *"display the reliability of the suppliers of blue parts"*:

```
(VP
  (Vbar
    (V
      (V_NP
        (V_NP display)
        (NP
          (Det the)
          (Nbar
            (N reliability)
            (PP
              (Pbar
                (P
                  (P of)
                  (NP
                    (Nbar
                      (N suppliers)
                      (PP
                        (Pbar
                          (P
                            (P of)
                            (NP
                              (Nbar
                                (AP
                                  (Abar
                                    (A blue)))
                                (N parts)))))))))))))))))
```

Bibliography

[1] P. A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Transaction of Database Systems*, 1:277–289, 1976.

[2] P. A. Bernstein and C. W. Chiu. Using semijoins to solve relational queries. *Journal of ACM*, 28(1):25–40, Jan. 1981.

[3] R. C. Berwick and S. P. Abney. *Principle-Based Parsing: Computation and Psycholinguistics.* Kluwer Academic Publishers, 1992.

[4] M. K. Brosey and B. Shneiderman. Two experimental comparisons of relational and hierarchical database models. *International Journal of Man-Machine Studies*, 10:625–637, 1978.

[5] J. S. Brown, R. R. Burton, and A. G. Bell. SOPHIE: A sophisticated instructional environment for teaching electronic troubleshooting. Report 2790, Bolt Beranek and Newman, Inc, Cambridge, Massachusetts, 1974.

[6] P.P.S. Chen. The entity-relationship model: Towards a unified vies of data. *ACM Trans. Database Systems.*, 1(1), March. 1976.

[7] E. F. Codd. A relational model of data for large shared data banks. *Communication of ACM*, 13(6):377–387, June 1970.

[8] E. F. Codd. How about recently? (English dialog with relational data bases using Rendezvous version 1). In Ben. Shneiderman, editor, *Databases: Improving Usability and Responsiveness.* Academic Press, 1978.

[9] Andyne Computing. *GQL User's Guide.* Andyne Computing Limited, Kingston, On, 1995.

[10] Ramez Elmasri. *Fundamental of database systems.* The Benjamin/Cummings Publishing Company, Inc, California, 1989.

[11] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness.* W. H. Freeman and Company, San Francisco, 1979.

[12] Gerald Gazdar, Ewan Klein, Geoffery Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar.* Basil Blackwell Publisher Ltd, Oxford, UK, 1985.

[13] D. Greenblatt and J. Waxman. A study of three database query languages. In B. Shneiderman, editor, *Database: Improving Usability and responsiveness*, pages 77–98. Academic Press, New York, 1987.

[14] L. Haegeman. *Introduction to Government and Binding Theory.* Basil Blackwell Ltd., 1991.

[15] M. Hammer and D. McLeod. Database description with sdm: A semantic database model. *ACM Transaction on Database Systems*, 6(3):351–386, Sept. 1981.

[16] L. Harris. Robot, a high performance natural language data base query system. *IJCAI-77*, pages 903–904, 1977.

[17] G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, and J. Slocum. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3(2), June 1978.

[18] Gary G. Hendrix and Brett A. Walter. The intelligent assistant. *Byte*, 12(14):251–258, December 1987.

[19] Jerry R. Hobbs, Mark Stickel, Paul Martin, and Douglas Edwards. Interpretation as abduction. In *Proceedings of 26th Annual Meeting of the Association for Computational Linguistics*, pages 95–103, Buffalo, 1988. ACL.

[20] et al I. Androutsopoulos. Masque/sql- an efficient and portable natural lanuage query interface for relational database. In *Proceedings of the Sizth International Conference on Industril Engineering Application of AI and Expert System*, Edinburgh, 1993.

[21] Henry F. Korth, Gabriel M. Kuper, Joan Feigenbaum, Allen van Gelder, and Jeffrey D. Ullman. System/U: A database system based on the universal relation assumption. *ACM Trans. on Database Systems*, 9 (3):331–347, Sept. 1984.

[22] D Lin. *Obvious Abduction.* PhD thesis, Dept. of Computing Science, University of Alberta, Edmonton, 1992.

[23] D. Lin and R. Goebel. Context-free grammar parsing by message passing. In *Proceedings of PACLING–93*, Vancouver, BC, 1993.

[24] Dekang Lin. A dependency-based method for evaluating broad-coverage parsers. In *Proceedings of IJCAI-95*, Montreal, Canada, August 1995.

[25] Dekang Lin and Bonnie Dorr. Government and binding theory and principle-based parsing. (submitted to Computational Linguistics), 1994.

[26] D. Maier and J. D. Ullman. Maximal objects and the semantics of universal relation databases. *ACM Transaction of Database Systems*, 8(1):1–14, 1983.

[27] D. Maier, J. D. Ullman, and M. Y. Vardi. On the foundations of the universal relation model. *ACM Trans. on Database Systems*, 9 (2):283–308, June 1984.

[28] A. Motro. Constructing queries from tokens. In *Proceedings of SIGMOD 86 Conference*, pages 120–131. ACM, 1986.

[29] Charles S. Peirce. *Collected Papers of Charles Sanders Peirce.* Cambridge, Mass., 1958.

[30] P. Reisner. Use of psychological experimentation as an aid to development of a query language. *IEEE Transactions of Software Engineering*, Se-3(3):218–229, 1977.

[31] S. Y. W. Su. SAM*: A semantic association model for corporate and scientific statistical databases. *Information Science*, 29:151–199, 1983.

[32] J. C. Thomas and J. D. Goold. A psychological study of query by example. In *Proceedings of the National Computer Conference*, Montvale, N. J, 1975. AFIP Press,.

[33] Jeffrey D. Ullman. *Principles of Database systems.* Computer Software Engineering Series. Computer Science Press, Rockville, Maryland, 2nd edition, 1982.

[34] J. Wald. *Problems in query inference.* PhD thesis, Dept. of Computational Science, University of Saskatchewan, Saskatoon, Canada, 1985.

[35] D. C. Waltz. An english language question answering system for a large relational database. *Communication of ACM*, 21(7):526–539, July 1978.

[36] D. Warren and F. Perira. An efficient easily adaptable system for interpreting natural lanuage queries. *Computational Linguistics*, 1995.

[37] W. Woods. Transition network programs for natural language analysis. *CACM*, 13:591–606, 1970.