

AD-A138 153

USER-FRIENDLY INTERFACE TO THE ROTH RELATIONAL DATABASE 1/2

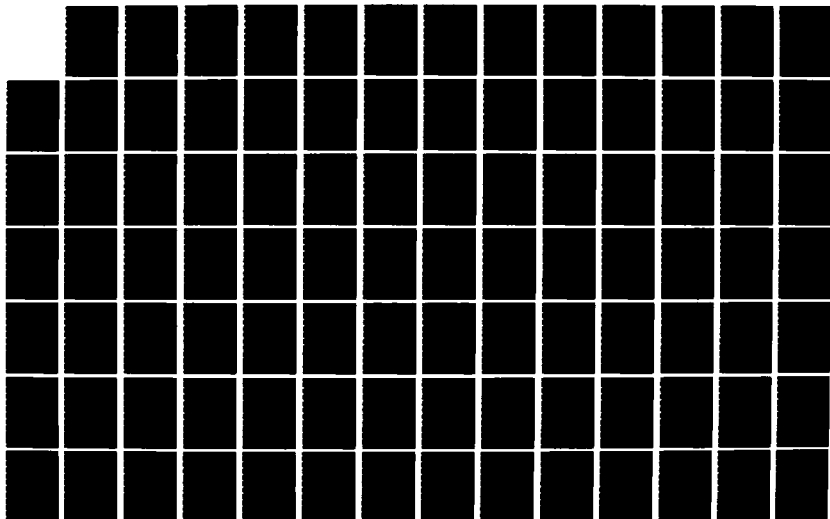
(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH

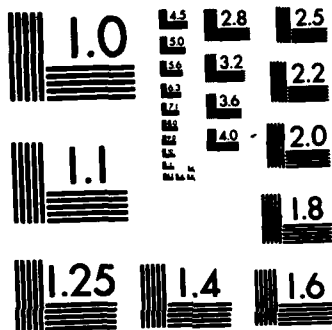
SCHOOL OF ENGINEERING D W VANKIRK 16 DEC 83

UNCLASSIFIED AFIT/GCS/EE/83D-21

F/G 9/2

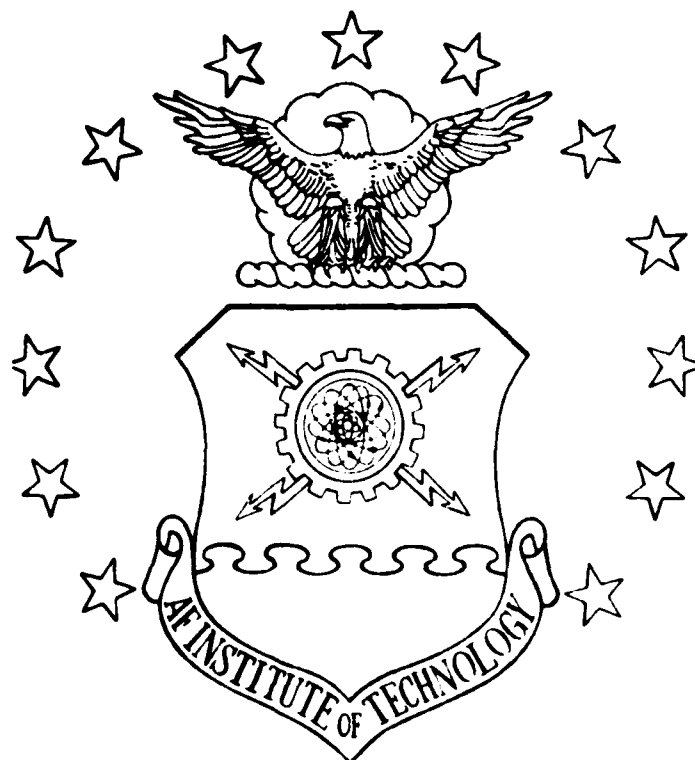
NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A138153



USER-FRIENDLY INTERFACE
TO
THE ROTH RELATIONAL DATABASE

THESIS

AFIT/GCS/EE/83D-21

Dale VanKirk
Capt USAF

DTIC FILE COPY

DTIC
ELECTE
FEB 22 1984
E

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

This document has been approved
for public release and sale; its
distribution is unlimited.

84 02 17 007

AFIT/GCS/EE/83D-21

Maj. Lillie
Dr. Hartrum
Dr. Potoczny

USER-FRIENDLY INTERFACE
TO
THE ROTH RELATIONAL DATABASE
THESIS

AFIT/GCS/EE/83D-21

Dale VanKirk
Capt USAF

DTIC
ELECTE
FEB 22 1984
S D E

Approved for public release; distribution unlimited

AFIT/GCS/EE/83D-21

USER-FRIENDLY INTERFACE
TO
THE ROTH RELATIONAL DATABASE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University (ATC)
In Partial Fulfillment of the
Requirements of the Degree of
Master of Science

by

Dale VanKirk
Capt USAF

Graduate Computer Systems

16 December 1983

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



Preface

Creating an interface to a database on a microcomputer is more "magic and mirrors", to quote a standard saying in the flying community, than anything else. This is particularly true in this case. Memory is at a premium when hard disks are not available. In trying to make a reasonable interface between someone without an extensive programming/database background and a computer, many short-cuts and assumptions must be made. One large assumption that was made herein is that a user who needs to use the interface will not be asking queries of a highly complex nature. This allowed use of the Universal Relation approach in attacking the database. Only queries that can be resolved with JOINS, SELECTs, and PROJECTs can be resolved by this interface. The specifics are inside.

To Maj. Lillie and Dr. Hartrum I say thank you for questioning the Universal Relation. Your questions prodded me to keep after the project. To Dr. Potoczny I say thanks - for providing a place to think aloud, and for encouraging me to keep after it. This committee gave me reason to study late and long.

To my wife, Susan, I say thanks for putting up with my late nights - and believing me when I went in to study after hours.

Table of Contents

Preface.....	iii
List of Figures.....	vi
List of Tables.....	vii
Abstract.....	viii
Chapter 1 Introduction.....	1
Background.....	3
Objective.....	4
Scope.....	4
Approach.....	5
Chapter 2 Concept of Design.....	6
Overview.....	6
Relational Query Requirements.....	6
Computer Concepts.....	10
Uniqueness Assumption.....	12
User Interface.....	18
Chapter 3 Constraints in Design.....	19
Overview.....	19
Software Capabilities.....	19
Hardware Limitations.....	22
Roth Database Requirements.....	23
Query Restrictions.....	25
Chapter 4 Reality of Design.....	29
Overview.....	29
System Structure.....	29
Testing.....	35
Chapter 5 Recommendations.....	40
NBS Conversion.....	40
String Functions.....	40
Memory Management.....	41
Join Optimization.....	42
NBS Pascal on LSI-11/2.....	43
Chapter 6 Conclusion.....	45
Review.....	45
Bibliography.....	47
Appendix A Intermediate Language Definition.....	50

Appendix B	Final Test.....	52
Appendix C	How MAKDIC Works.....	56
Appendix D	What NATQRY Really Does.....	58
Appendix E	What Error Messages Mean.....	64
Appendix F	NBS Compiler Notes.....	68
Appendix G	NATQRY Structure.....	74
Appendix H	NATQRY Listing.....	76
Appendix I	MAKDIC Listing.....	134
Appendix J	STD.DIC Listing.....	138
Vita.....		140

List of Figures

<u>Figure</u>		<u>Page</u>
1.	BANKING DATABASE HYPERGRAPH.....	16
2.	INTERFACE TOP LEVEL.....	30
3.	THE PROCESSQRY MODULE.....	32
4.	THE ALGEBRA MODULE.....	34

List of Tables

<u>Table</u>	<u>Page</u>
I. PART/SUPPLIER DATABASE.....	8
II. PART/SUPPLIER UNIVERSAL RELATION.....	13
III. DEPARTMENT STORE DATABASE.....	15
IV. RELATIONS IN THE BANKING DATABASE.....	16
V. TIME AND SPACE REQUIRED to run NATQRY.....	38

Abstract



A Friendly Interface to a Relational Algebra Database System was created on the Universal Relation concept. This concept allows the user to relate to the total database as a single relation. The user inputs a query using attributes of the single relation. The interface then creates the Universal Relation by way of relational algebra JOINS. Tuples of this relation are SELECTed according to constraints placed on attributes in the query (i.e. CITY = NEW YORK), and a PROJECTION of attributes desired is made from the result. Limitations of the interface are:

1. All relations in the database must be JOINable without data loss.
 2. The query must start with a verb.
- 

Chapter 1

Introduction

More information is available now than ever before, and the size and number of databases are growing every year. Virtually any topical area is referenced in a database somewhere. However, only those people who know how to access these specific databases can get at the information inside.

This is a report on a project to make one particular database accessible to more people. This was not accomplished by creating training courses for potential users, but by changing the interface to the database, making it more friendly. The resultant User-Friendly Interface allows a person with little or no knowledge of the inner workings of the database to access the information held inside.

Just what is a user-friendly database interface? The obvious answer is one friendly to the user. It seems that the friendliness of a database depends on the knowledge of the user. Of course, any interface at all assumes some knowledge on the user's part; but how much knowledge, and in what areas is it required? The more overlap the interface has with basic education requirements, the more friendly it will be to the greatest number of users. Naturally, what is friendly to one user can be unbearable to another.

Many approaches to achieving a level of friendliness acceptable to a perceived user of average intelligence are

being pursued. The Artificial Intelligence community is studying ways to have a computer verbally interact with the user. Sentence analysis of proper English (or whatever language the researcher is working with) is important to this study. A similar goal short of overall verbal communication is natural language interfacing [27]. A natural language is "a language whose syntax reflects and describes current usage, rather than prescribed usage" [35].

The natural language based interface is promising for large scale, future projects; but it is too complex and time consuming to implement for purposes at hand, because it requires recognition of parts of speech. Therefore, a much simpler approach to the interface was taken. A simple English-like statement starting with a verb makes up the input to the interface. This interface is certainly not friendly to non-English speaking users, but the vast majority of English speaking users with an elementary level education should be able to access the database with this interface.

Assuming the user has a minimal level of familiarity with computers, the interface:

1. Accepts English-like input.
2. Verifies input as a valid request.
3. Restates the input in a form consistent with the database management system.

Background

In 1979, Second Lieutenant Mark Roth designed a relational database based on relational algebra (see [5, 7, 27, 33] for discussions of relational databases and relational algebra). His purpose was to provide an instructional tool for use in database management. A major portion of the design was implemented. UCSD (University of California, San Diego) Pascal was used as the programming language. Although originally planned for an Altair 8800b system, the package now runs on an LSI-11 at the Air Force Institute of Technology (AFIT) Digital Engineering Lab [29].

Roth's database is the target database for this user-friendly interface. The English-like input to the interface is transformed to relational algebra and is stored in a file Roth refers to as a command file. This command file can then be executed against the database at the user's convenience.

Captain Michael Guidry designed a universal query resolution system in 1982. His purpose was to implement a software package that could accept natural language queries and restate them in a language easily translated into a form required by either a network, hierarchical, or relational database (see [7, 33] for discussions of these database types). Guidry did not fully implement his design. One part of his design that was nearly completed in implementation was called Analyzer. This package, written in UCSD Pascal, takes a natural language input and completes a syntax check on the query [12]. Analyzer originally was

to provide the interface created in this project, however, development of the universal aspect of his query resolution system kept Guidry from completing it.

Objective

The objective of this project, as stated earlier, was to implement a User-Friendly Interface to the Roth Relational Database. This was accomplished by supplying the user with a minimal number of rules, and showing him a list of the attributes available in the database. The user then inputs a query. The interface translates this query to an intermediate form which reflects the syntax required by the Roth system. This intermediate form is then transformed to relational algebra. During transformation, the query syntax is checked. Finally, the query is put on a disk file for future submission to the Roth database for optimization and resolution.

The user sees only the initial rules, the attributes of the database, and any error messages produced during transformation of the query. In this way, the user is only required to know what attributes are related to the information he desires to have.

Scope

The limit of this project was to implement the interface to the Roth Relational Database. Research in the areas of natural languages and relational databases was used to prepare for the task. Discovery of the Universal Relation concept of relational database design (also

referred to as the Uniqueness Assumption) had a major impact on implementation of this interface (see chapter 2).

Approach

Literature was researched in the areas of natural language and relational databases. Further, a capability comparison was conducted between UCSD Pascal and NBS (National Bureau of Standards) Pascal. The purpose of this comparison was to determine if a change of programming language would reasonably ease the size restrictions inherent to the UCSD Pascal -- LSI-11 match-up, without requiring massive rewrites of the existing software.

Next, a study of the Roth database and Analyzer package was conducted to give direction to development of the interface. This study dovetailed into the design of modules needed to complete the task.

Finally, coding, testing, and integration of the interface was completed.

Chapter 2

Concept of Design

Overview

This chapter presents the concept of a relational model User-Friendly Interface. The first consideration is what information must be provided to resolve a query in a relational database. Next, a discussion of how a data dictionary could be employed in query resolution is given. This is followed by a section introducing the Uniqueness Assumption of database design, and its part in simplifying the natural language to relational algebra transformation. The final section is a sketch of the interface created in this project.

Relational Query Requirements

Information is drawn out of a database by means of queries. Naturally, since computers do not reason, a specific format of information must be provided to the computer, directing it to the data required. If the query does not meet the format required, or does not supply enough information to locate the data, the query fails. That is, either the computer returns an error message indicating its inability to find the data from the information given, or incorrect data is returned.

For a relational database, the query must indicate:

1. What attributes are required as the answer.
2. What attributes are required to determine which data are in the answer.

3. What relations contain the attributes required for items one and two.
4. What operations must be performed to determine the answer.
5. Which relations/attributes are related with each required operation.

The computer's job is to recognize relation and attribute names, find them in the database, and apply the specified operations. This is no trivial task, however. Query preparation for this task is more than a database user with little or no experience can handle.

As an example of query recognition by the computer, consider the part/supplier database Date [7] uses. The relations are named S, P, and SP. Attributes are as follows:

Relation S - S#, SNAME, STATUS, CITY

Relation P - P#, PNAME, COLOR, WEIGHT

Relation SP - S#, P#, QTY

Relation S has information on part suppliers. S# is the supplier's number, SNAME is the supplier's name, STATUS is the supplier's status code, and city is the supplier's location. Similarly, Relation P has information on parts - their number, name, color, and weight. Finally, Relation SP has shipment information. It tells which suppliers provide the part in the shipment, which part the shipment is made of, and how many total parts are in the shipment. The database is depicted in Table I.

Table I
PART/SUPPLIER DATABASE

S	S#	NAME	STATUS	CITY	P	S#	P#	QTY
	S1	SMITH	20	LONDON		S1	P1	200
	S2	JONES	10	PARIS		S1	P1	700
	S3	BLAKE	30	PARIS		S2	P3	400
	S4	CLARK	20	LONDON		S2	P3	200
	S5	ADAMS	30	ATHENS		S2	P3	500
						S2	P3	600
						S2	P3	800
						S2	P5	100
P	P#	PNAME	COLOR	WEIGHT		S3	P3	200
	P1	NUT	RED	12		S3	P4	500
	P2	BOLT	GREEN	17		S4	P5	300
	P3	SCREW	BLUE	17		S5	P2	200
	P4	SCREW	RED	14		S5	P2	500
	P5	CAM	BLUE	12		S5	P5	500
						S5	P1	1000
						S5	P3	1200
						S5	P4	800
						S5	P5	400

Given this database, a user might want to know the supplier's number for all suppliers who supply part number P2. It is a simple task to answer this query with the database laid out as in Table 1. Notice, however, the associations that must be made to answer the query:

1. The answer is a list of supplier numbers.
2. Supplier numbers are listed as S#.
3. A particular S# is in the answer only if the supplier belonging to that S# supplies part P2.
4. Part P2 is a specific value, listed under the heading P#.
5. S# and P# are only related to each other in relation SP.

6. A comparison between every P# in relation SP must be made with the value P2.

7. The S# in SP is recorded as part of the answer every time P# associated with it results in equality based on the comparison.

These associations are very easy to make. It is almost difficult to believe all of them are needed because most are automatically made. However, the computer cannot make these associations, every one must be provided.

Another example shows the complications that can arise in making associations:

QUERY - Get supplier names for suppliers who supply all parts.

ASSOCIATIONS -

1. The answer is a list of supplier names.
2. Supplier's names are listed as SNAME.
3. SNAME is in the answer only if that supplier supplies every part.
4. Parts are referenced by P#.
5. SNAME can be associated to P# in relation SP by way of S# (a complex association).
6. Every possible P# is in relation P.
7. A comparison between every possible P# must be made with a list of every P# associated with each S# in relation SP (a complex association).
8. S# is saved if the previous comparison reveals that this S# is associated in relation SP with

every P# in relation P (a complex association).

9. Every SNAME in relation S that associates with S# in the saved list is recorded as part of the answer.

Associations five, seven, and eight are complex associations. Answering this query by looking at the database is much simpler than explaining how to answer it. Yet, after going through the mental exercise of explaining it, one must still formulate how to explain the solution in a language familiar to the computer. The next section deals with the task of explaining to the computer how to make the associations just covered.

Computer Concepts

As previously mentioned, a computer cannot make associations by itself. Some sort of aid must be provided. One aid in use is the data dictionary. As Peters [25] states,

"The basic goal is to create a catalog that identifies each data item, the data items of which it is composed (if any), any aliases by which it is known, and (when practical) the values it may take on".

A data dictionary for a relational database should include the relation names, attribute names, and legal operators. Each entity in the query is checked against the data dictionary to see if it is a recognized name or operator. If the entity is found in the dictionary, appropriate

actions are taken to clearly indicate what the entity is and how it is used. Identified relations show what attributes are contained in them, identified attributes show which relations they are found in, and operators are related to their proper symbolic form.

Consider the first example query again:

"What is the supplier's number for all suppliers
who supply part number P2?"

Checking with the data dictionary, the resultant query is built:

1. "What is" becomes LIST.
2. "supplier's number" becomes S#.
3. "all suppliers" becomes the condition ALL
for the operation LIST.
4. "who supply" becomes WHERE.
5. "part number" becomes P#.
6. "P2" becomes a condition on P#.

The resultant query, LIST ALL S# WHERE P# = P2, is then ready for resolution. The data dictionary also provided a list of relations associated with each attribute used in this query. This relation list is used during query resolution.

It is important to note here the default values of LIST and the equal sign for the condition on P#. These items are examples of capabilities based on semantics and syntax built into an interface. How this resultant query is produced in this interface is covered in chapter 4.

Uniqueness Assumption

The answer to a query of the form produced above is simple to produce under the Uniqueness Assumption of the Universal Relation type database [4, 21]. This type of database is formed so that every internal relation can be joined together into a single relation containing all of the data in the database. In this one relation, each attribute must have a unique name.

If the database were a single relation, most queries would be easily answered by a simple procedure:

1. Find the rows that meet the requirements to be in the answer.
2. Take the information from the column identified by the attribute requested.

Using this procedure, the previous query would be answered by finding every row in the universal relation where P# equals P2, and putting the S# from that row in the answer.

These two steps are effected by the relational algebra SELECT and PROJECT statements. Rows are SELECTed based on an attribute's relationship to either a specific value or another attribute. Once the rows are SELECTed, the appropriate columns (attributes) are PROJECTed out into the answer. The relational algebra for this query, based on a Universal Relation named TOTAL is:

```
SELECT ALL FROM TOTAL WHERE P# = P2 GIVING  
INTERMEDIATE.  
  
PROJECT S# FROM INTERMEDIATE GIVING ANSWER.
```

INTERMEDIATE is a relation created by the SELECT statement to hold the result of that statement. ANSWER is a relation created by the PROJECT statement to hold the final answer to the query. Table II depicts the Universal Relation for the part/supplier database, and can be used to verify the procedure.

Table II
PART/SUPPLIER UNIVERSAL RELATION

S#	SNAME	STATUS	CITY	P#	PNAME	COLOR	WEIGHT	QTY
S1	SMITH	20	LONDON	P1	NUT	RED	12	200
S1	SMITH	20	LONDON	P1	NUT	RED	12	700
S2	JONES	10	PARIS	P3	SCREW	BLUE	17	400
S2	JONES	10	PARIS	P3	SCREW	BLUE	17	200
S2	JONES	10	PARIS	P3	SCREW	BLUE	17	500
S2	JONES	10	PARIS	P3	SCREW	BLUE	17	600
S2	JONES	10	PARIS	P3	SCREW	BLUE	17	800
S2	JONES	10	PARIS	P5	CAM	BLUE	12	100
S3	BLAKE	30	PARIS	P3	SCREW	BLUE	12	200
S3	BLAKE	30	PARIS	P4	SCREW	RED	14	500
S4	CLARK	20	LONDON	P5	CAM	BLUE	12	300
S5	ADAMS	30	ATHENS	P2	BOLT	GREEN	17	200
S5	ADAMS	30	ATHENS	P2	BOLT	GREEN	17	500
S5	ADAMS	30	ATHENS	P5	CAM	BLUE	12	500
S5	ADAMS	30	ATHENS	P1	NUT	RED	12	1000
S5	ADAMS	30	ATHENS	P3	SCREW	BLUE	12	1200
S5	ADAMS	30	ATHENS	P4	SCREW	RED	14	800
S5	ADAMS	30	ATHENS	P5	CAM	BLUE	12	400

The simplicity of this procedure makes use of the Universal Relation very attractive. Given that the user-friendly interface has to run on a microcomputer, any method of saving memory is worth looking into. Yet, as useful as the method is to the purposes at hand, there is a difficulty with the Universal Relation.

The Universal Relation method can only be used on a

database whose relations can be joined into a single relation without loss of data. This lossless join property is not as bad a problem as it seems, however. Ullman [33] states that "It turns out that any relation scheme has a lossless join decomposition into Boyce-Codd Normal Form, and it has a decomposition into third normal form that has a lossless join and is also dependency-preserving". In other words, when a relational database scheme is designed, attention must be paid to normalizing the internal relations (see [7, 33] for discussion of normal forms). In doing this properly, the creator of the database can go to a third normal form or even Boyce-Codd normal form and still be compatible with the Universal Relation scheme. Fagin and company state further that the lossless join property is chief among all desired properties in a relational database scheme. By ensuring the scheme includes the lossless join, other desirable properties are automatically realized [10].

Beller [4] warns of the uniqueness aspect of the Uniqueness Assumption. He says that the Universal Relation does not reflect real world situations. His concern is based on a database constructed as in Table III.

Notice that the attribute FLOOR has a double dependency. In relation MAIN, it refers to the floor the DEPARTMENT office is on. In relation WORKER, FLOOR refers to the floor the employee works on. Therefore, any query involving FLOOR must include which reference to FLOOR is required. If the reference is not included, workers could

be reported as working on the same floor as their department head office, when this is not the case.

Table III
DEPARTMENT STORE DATABASE

MAIN

DEPARTMENT	FLOOR	DIRECTOR
Toys	2	Jones
Furniture	1	Harris
Linen	5	McKenzie

WORKER

EMPLOYEE	FLOOR	BOSS
Tom	3	Harris
Dick	7	McKenzie
Harry	2	Jones
Jones	2	Smith
Harris	1	Smith
McKenzie	5	Smith

Beller discusses an attribute name change method to clear up the ambiguity presented by a double use of a single attribute. By changing the attribute FLOOR in relation WORKER to the name PLACE, the ambiguity is gone. So is the capability for join. Therefore, in addition to renaming FLOOR in relation worker, the attribute FLOOR must be added to the relation with data consistent to its meaning in relation MAIN.

Maier and Ullman [21] take a different approach to the problem Beller presented. They work with the database of Table IV. In trying to join these relations, ambiguity occurs. This ambiguity cannot be traced to a specific attribute that has a double meaning, however the hypergraph

of Figure 1 shows the candidate attributes for causing the ambiguity. (A hypergraph is a graph in which edges are a set of nodes.) A cycle is formed by attributes BANK, ACCOUNT, CUSTOMER, and LOAN. One of these attributes is causing the ambiguity. Obviously, a BANK can be related to a CUSTOMER through either a LOAN or an ACCOUNT. Using the hypergraph, and starting with LOAN, two paths to CUSTOMER are available. The path by way of BANK and ACCOUNT

Table IV
RELATIONS IN THE BANKING DATABASE

RELATION	ATTRIBUTES
1	BANK, ACCOUNT
2	ACCOUNT, BALANCE
3	ACCOUNT, CUSTOMER
4	CUSTOMER, ADDRESS
5	LOAN, BANK
6	LOAN, CUSTOMER
7	LOAN, AMOUNT

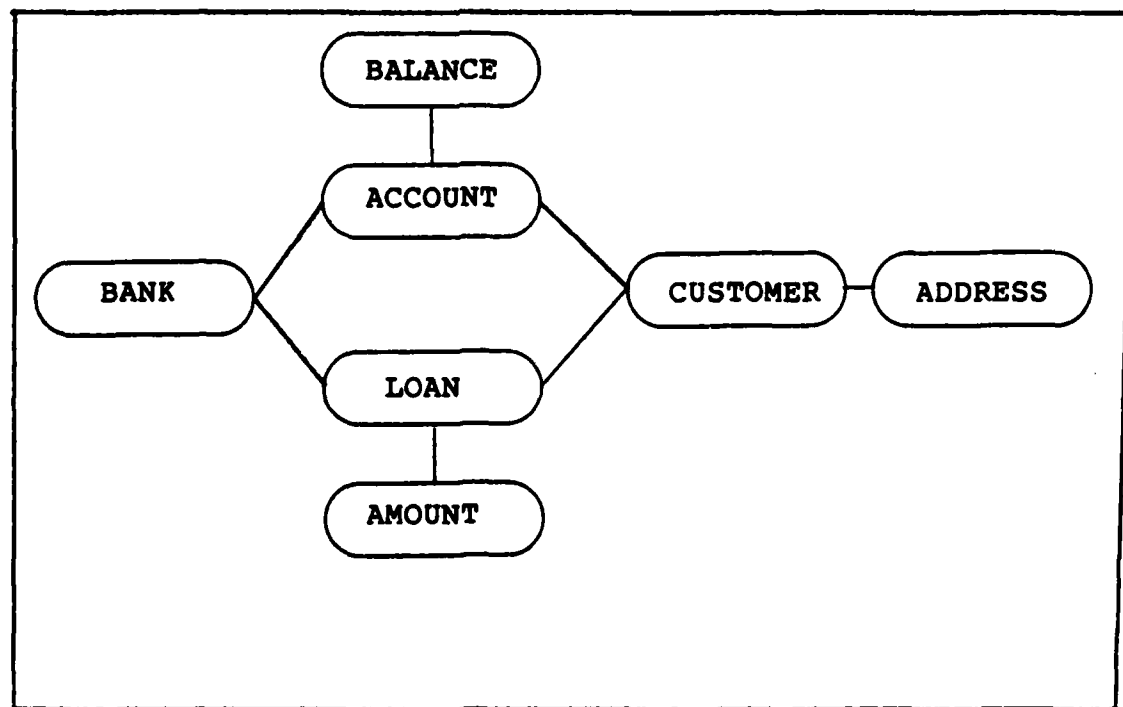


Figure 1. BANKING DATABASE HYPERGRAPH

indicates that the customer has an ACCOUNT at the same BANK he has a LOAN, where the direct path (LOAN to CUSTOMER) does not infer anything about an ACCOUNT in relation to this CUSTOMER. Similarly, the paths from LOAN to BANK are at odds.

Rather than redesigning the database, Maier and Ullman propose using query restrictions by way of maximal objects. A maximal object is defined as a subset of the Universal Relation within which a query is resolved. Two maximal objects were defined on the database of Table 4, the set of relations {1, 2, 3, 4} and {4, 5, 6, 7}. Queries involving LOANS are routed to the first maximal object, while queries involving ACCOUNTS are answered from within the second maximal object. Once inside a maximal object, data outside of that maximal object is disallowed. In this way, ambiguities previously present are done away, while the original form of the database is preserved.

Maier and Ullman conclude that while the Universal Relation "may not be perfect for everything, it does certain things well enough to be valued by its users". At least someone else agrees with Maier and Ullman, for King [18] states that QUIST (QQuery Improvement through Semantic Transformation) is based on the Universal Relation concept. Of course, QUIST can not handle every query that can be made, because of its dependence on the Universal Relation. However, it does handle a very important, large subset of all queries that are possible.

User Interface

The User-Friendly Interface created in this project, being based on the Universal Relation concept, requires that the database be well planned. As mentioned above, Boyce-Codd or Third Normal form with lossless join is required.

Operation of the interface is straight-forward. Upon receiving the query, the data dictionary is accessed to produce an intermediate query. All relations are JOINed to produce the Universal Relation, and the intermediate query is scanned to determine if a SELECT is required. If a SELECT is required, it is produced, and the intermediate query is again scanned to determine the attributes required in the PROJECT.

The SELECT can be skipped if the user fails to specify a constraint on an attribute ($P_1 = P_2$). In this event, the answer will be a PROJECTION from the Universal Relation.

After the relational algebra is produced, it is shown to the user. The user is given the opportunity to save the relational algebra in a command file, then is asked if he would like to try another query. The new query can be against the same database, or disks can be moved to run against a different database.

Chapter 3

Constraints in Design

Overview

This chapter deals with various limitations and situations encountered during the project, and how they were dealt with. The first area concerned is Software Capabilities. The discussion reveals why NBS Pascal (a Pascal compiler written for the National Bureau of Standards at the University of Montana) was chosen as the language of the interface. The second area is hardware. One of the requirements of this project was that it would run on an LSI-11 microcomputer. Why this requirement existed, and considerations involved are dealt with in Hardware Limitations. The third section, Roth Database Requirements, deals with how the interface blends in with the Roth package. Finally, Query Restrictions tells what restrictions the interface places on queries, and why they exist.

Software Capabilities

The Roth system was originally written in UCSD (University of California, San Diego) Pascal. Roth [29] cited five reasons for this choice:

1. Block structured design of the language.
2. High degree of variable typing.
3. Wealth of control structures.
4. Portability.
5. Capability to handle large programs through

program segmentation, separately compiled procedures, and segment swapping.

When the Roth system was moved from the Intel 8080 to take advantage of a hardware change in the AFIT Digital Engineering Lab, portability of UCSD Pascal was tested. Second Lieutenant Mau [24] found the task was not as simple as copy over and run. When Mau finished the move to the LSI-11, Second Lieutenant Rodgers [28] tackled the problem of finishing the implementation of the Roth system. Her first obstacle was the block structure. Although UCSD Pascal allowed segmentation of programs into independently run segments [32], seven segments were too few. Rodgers finally separated the system into two independent programs in an effort to have enough room to run them.

Unfortunately, the Roth system still had no ability to actually receive data. Since the actual database will take some memory space, regardless of how input and retrieval of data is performed, the size problem continued. At this juncture, the User-Friendly Interface was introduced.

Since the initial move to the LSI-11 was made, the Digital Engineering Lab started a project to tie seven LSI-11's together into a microcomputer network. This project presented a logical solution to the memory problem of the Roth system, that is, move each component to its own microcomputer, and pass data between machines. However, UCSD Pascal uses its own operating system, and the network project was pursued in the C language, based on the RT-11

operating system. Neither operating system (UCSD and RT-11) can read the disk directory of the other's making, keeping the benefits of the network from immediately being realized. Certainly, a program to convert one directory to the other's format would work, but the UCSD segmentation limitation would still be around to plague future plans for enhancements to the Roth system. Clearly, UCSD was a limiting factor to further development of the Roth system.

In 1981, Professor John Barr of the University of Montana headed a group that released version 1.6i of a Pascal compiler called NBS Pascal [2]. This compiler was created to run on an LSI-11/23, and was based in the RT-11 operating system. This Pascal had two major, practical points in its favor for replacing UCSD:

1. It was already available the Digital Engineering Lab.
2. It was compatible with the RT-11 operating system, making files needed by the interface available for transfer on the LSI network.

In addition, being Pascal, it should make conversion of the Roth system to NBS somewhat easier than a total rewrite into another language. By writing the User-Friendly Interface in NBS Pascal, the movement from UCSD would be underway.

While the benefits of NBS were obvious, they were not legion, and some drawbacks existed. NBS does not support strings, or string functions; and PACKED, GOTO, and LABEL are recognized as reserved words, but no special action is

taken. Fortunately, all character arrays in NBS are stored sequentially, allowing single statement comparisons and assignments between arrays of the same size (see chapter 5 for more on this area).

Hardware Limitations

The Digital Engineering Lab started acquiring LSI-11 microcomputers in 1980. Given the number of users for the Intel 8080 (where Roth implemented his database), the decision to move to one of the five new LSI-11's was easy to make [24]. Since that time, three more LSI-11's have been acquired. Of the eight machines, only one is configured as an LSI-11/23. This was crucial to the NBS Pascal decision. The NBS Pascal compiler is hardware dependent to a certain degree. One version exists for the LSI-11/2, and another exists for the LSI-11/23. This difference stems from the hardware setup for handling floating point numbers. The LSI-11/2 has an EIS/FIS (Extended Instruction Set/Floating Instruction Set) combination on a single chip [23]. Floating point calculations are handled by four instructions. On the other hand, the LSI-11/23 has the floating point instruction set on a chip by itself. It implements floating point calculations with a set of 26 instructions. The four EIS/FIS floating point instructions are not included in the LSI-11/23 instruction set. The result is that although no floating point calculations are done for the database project, NBS programs compiled on one machine will not run on the other.

Currently, none of the machines in the LSI network is an LSI-11/23. At least one of the machines in the network could be reconfigured, but that point was not critical to the task at hand. In the worst case, the Roth system could be relegated to a single LSI-11/23. The memory problems would still be evident, but separation of the system into separate programs would be at least as effective as under UCSD.

The next worst case would be to obtain a copy of the NBS Pascal compiler compatible with the LSI-11/2. The interface could then be recompiled, and the network would handle the memory restrictions. Therefore, the interface project was continued on the LSI-11/23 in NBS Pascal, realizing that the resulting situation would compel a decision to either put LSI-11/23's in the network or get the LSI-11/2 compatible NBS compiler (see chapter 5, NBS Pascal on the LSI-11/2).

Roth Database Requirements

The Roth system implements a relational database using relational algebra. It uses a tree-like structure to perform its function [24, 28, 29]. At the second level of the Roth system tree, six functions are available:

- | | |
|-------------|--------------|
| 1. SETUP | 4. ATTACH |
| 2. EDIT | 5. INVENTORY |
| 3. RETRIEVE | 6. QUIT |

Of these, only the RETRIEVE function is involved with database queries. Within the RETRIEVE module are five

functions concerned with user queries. They are explained as:

1. GET - Get a disk file into a specified memory buffer called the workfile.
2. SAVE - Saves a workfile by placing it on disk.
3. EDIT - Allows creation/modification of the workfile.
4. EXECUTE - Uses the workfile as a list of relational algebra statements to retrieve data from the database.
5. DISPLAY - Shows the contents of a relation on the CRT.

Obviously, the User-Friendly Interface belonged below the RETRIEVE function. Without being able to directly interface with the Roth system due to the UCSD-NBS incompatibility, the User-Friendly Interface had to be self-contained.

Also, the format of the interface output had to be considered. Two distinct possibilities were available. One was to create, optimize, and store the relational algebra in a tree-form. The other was to produce a file of relational algebra Roth calls a command file. Since Roth provides a form of query optimization, the second approach was used. This way, the interface makes use of the Roth optimizer, and need not plan program changes to accomodate tree structures stored in a disk file for a system that will require movement to another operating system, and probably change in the transfer.

Query Restrictions

Five rules are given to the user of the User-Friendly Interface. Explanation follows each rule.

1. A verb must be the first word in a query.

The purpose of this rule is to put the user into a specific line of thinking. By starting the query with a verb, the user will typically follow with a list of objects he wants to see in the answer. This list of objects is naturally followed by a list of constraints on which specific objects should or should not be in the answer. That is, a user wanting to know the supplier's name and city who supplies part P2 might say "Show supplier's name and city for suppliers of part P2". It is this direction of thinking that the User-Friendly Interface depends on. The verb itself is unimportant. Actually, if no verb is given, but the line of thinking is followed, the query will be successful. This is because the interface provides the verb.

2. The rest of the query must be in sentence format and use provided attribute names where possible.

The purpose of this rule is to further guide the user into the direction of thinking rule one established. The first part of this rule is designed to show the user that he can finish the query in English without an involved format to follow. The second part of this rule requests the use of attribute names. It is crucial that attributes be used.

The interface does not include an extensive dictionary of misspelled words or aliases. Only operators, attributes, and a few key words pass the dictionary check to be put in the intermediate query.

3. Specific attribute values must be in quotes.

Since the user's query is taken to an intermediate form without the aid of a formal format, and data is not yet available, a method of recognizing specific attribute values had to be created. Otherwise, the unrecognized value would be deleted from the query. By putting a beginning quote on the value, the specific value is signaled. The ending quote is added as a method of allowing the user to use a two word specific value, as in CITY = "New York". The restriction on specific value length is two words not to total more than 20 characters, including intervening space.

Certainly, any word following an operator (=, <, >, <>) could be considered a candidate for a specific value in the intermediate query. However, in the user's input of "Print SNAME for CITY equal to the value New York", there are three intervening words (to, the, value), and no way of knowing the specified value is in two parts. Quotation marks simplified the problem.

4. You must signal end of the query by ending with a period.

This rule lets the user hit a carriage return without signalling the end of the query. Normally, a carriage return signals end of input. By waiting for a period,

the probability is much higher that the user finished his input than if a carriage return alone were the signal. Carriage returns are almost a reflex action for many who are familiar with typewriters and keyboards. This rule also allows a query to extend beyond a single line.

5. The query may not be more than four lines long (320 characters).

Since NBS Pascal does not support strings, an array was used to hold the query. This was deemed more feasible than creating a linked list of single characters. It is both more memory efficient, and time efficient. The limit of four lines is given to communicate to those unaccustomed to counting characters. The size of the array (320 characters) should be sufficient for most queries.

Rules one and two presented the idea of a direction of thinking that would fit an internal language format. This internal language has its roots in a language used by Cousins [6]. The basic format of this language is:

· VERB OBJECT QUALIFIER

Cousins used the format to build query trees and determine data access paths. Since the Roth system uses a command file to build a query tree and determine an optimized data access path, the decision was made to use the same format Cousins used, only for the purpose of creating a command file. This decision saved internal memory during execution

of the interface.

The VERB-OBJECT-QUALIFIER format was implemented as follows:

VERB - Optional item. Default is the word LIST.

OBJECT - Optional item. An object is a list of one or more attributes.

QUALIFIER - Optional item. A list of one or more qualifiers joined by a logical operator (AND, OR). The qualifier itself is an attribute-operator-value grouping.

Chapter 5 adds detail to this introduction of the internal language, and a Backus-Naur Form representation of it is presented in Appendix A.

Chapter 4

Reality of Design

Overview

This chapter presents the structure of the User-Friendly Interface. Structure charts and algorithm flows are shown for the most important features. Also included is a discussion of testing completed on the interface, and a sample of time and disk space required to process a query (Table V).

System Structure

Figure 2 depicts the top level of the program. The module MAIN drives the modules needed for communicating with the user. All other modules are invisible to the user except when an error is discovered and reported. Algorithm flow for the MAIN program follows:

```
Clear the CRT.
UNTIL the user has disk space for a file:
    Request user identification.
    Ask the user if disk space is sufficient.
END.
WHILE the user wants to continue:
    Get drive locations for data dictionary files.
    List instructions for the user.
    Retrieve and show attributes from the database.
    IF a valid database is available:
        Recieve, translate, and edit the query.
        Transform the query to form relational algebra.
        IF no error occurred:
            Give user option to save relational algebra.
        END.
    END.
    Ask user if another query is desired.
END.
```

The disk space required on the default drive is fairly small. This question only comes into play if the user's disk has no available file larger than approximately 10

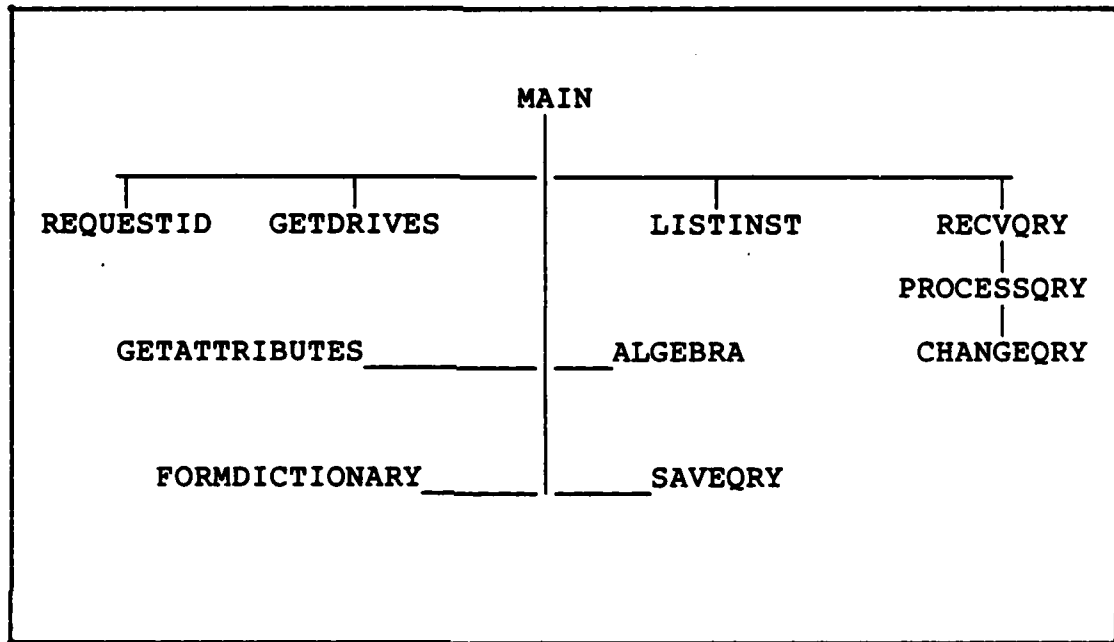


Figure 2. INTERFACE TOP LEVEL

blocks. The RT-11 operating system looks for contiguous space to store files. Therefore, even though a disk may have plenty of total space, if no contiguous space is of sufficient length, the file cannot be saved. During testing, the interface created files of up to 14 blocks, but most query command files are 5 - 8 blocks long. Because the command files created are relatively small, the user is advised to continue if he does not know how much space is available. Still, the safest method is to do an RT-11 SQUEEZE of the disk, followed by a DIRECTORY/FULL to be sure space is available.

Before running the interface, in addition to the space requirement, the user should know which disk has the following files on it:

SETUP.DAT - The file produced by Roth's DDL code as SETUP.DATA. This file name is consistent with the RT-11 operating system, which allows filenames of six characters and extensions of three characters. The file contains domain, relation, and attribute definitions for the database being queried.

STD.DIC - A file containing standard dictionary items. The file contains keywords and operators most often used in queries.

DATBAS.DIC - A file containing attribute aliases of the database being queried. Probable spelling errors can be put in this file.

STD.DIC and DATBAS.DIC are created by a supporting program developed with the interface called MAKDIC. MAKDIC is fully presented in the "User's Guide to the User-Friendly Interface to the Roth Relational Database" (see chapter 6, Final Note, on where to obtain the guide).

The user is asked, one file at a time, if that particular file is on the BOOT drive. These files contain the only words and operators that will appear in the intermediate query (quoted specific values excepted). If any one of the files cannot be found, the query translation process could eventually fail. Following the questions,

FORMDICTIONARY creates linked list dictionaries from these files.

The user's query is received by the RECVQRY module. When the user completes input (signalled by a period), PROCESSQRY is called to take the query to its intermediate form. This translated intermediate form of the query is shown to the user, pending approval. If the query is approved, transformation to relational algebra is accomplished. If the query is not approved, CHANGEQRY is activated to allow the user to designate where the query is incorrect. When the user finishes designating where the error is, RECVQRY is again called to receive the changes.

PROCESSQRY contains the most vital part of the interface, the translation to the intermediate language. The PROCESSQRY structure is shown in Figure 3.

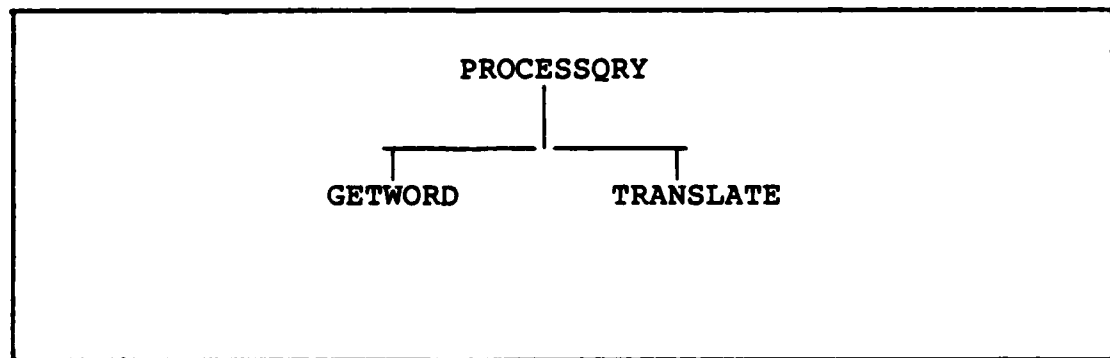


Figure 3. THE PROCESSQRY MODULE

The algorithm is a simple one:

```
WHILE not at the end of input query:
    Get the next word from input query.
    Translate the word to the intermediate query.
END.
```

Translate is a culling operation. Any word (unbroken string of characters) not found in one of the three files is ignored. If the user followed the rules discussed in chapter 3, the resulting intermediate query should be of the form:

LIST object [placeholder] attribute operator value
[logical attribute operator value]

where:

LIST is the verb.

object is a list of attributes.

placeholder, if present, is one of the words FOR, WHEN,
or WHERE.

operator is <, >, =, <>, >=, or <=.

value is either a specific quoted value or an
attribute.

The placeholder provides a logical divider between the list of attributes for the PROJECT, and the first attribute used as a row SELECTION rule. If the placeholder is not there, the answer will include a PROJECTION of the first attribute used in the SELECTION.

SAVEQRY gives the user an opportunity to see the relational algebra that was produced as a result of his query, and decide if it should be saved as a command file. If the relational algebra is to be saved, the user is prompted for a filename to store the relational algebra in.

Translation of the intermediate query to relational

algebra is handled by the ALGEBRA module, shown in Figure 4.

ALGEBRA's flow:

```
Create the Universal Relation.  
IF SELECTION criteria exist:  
    Produce relational algebra SELECT statement.  
END.  
Produce relational algebra PROJECT statement.
```

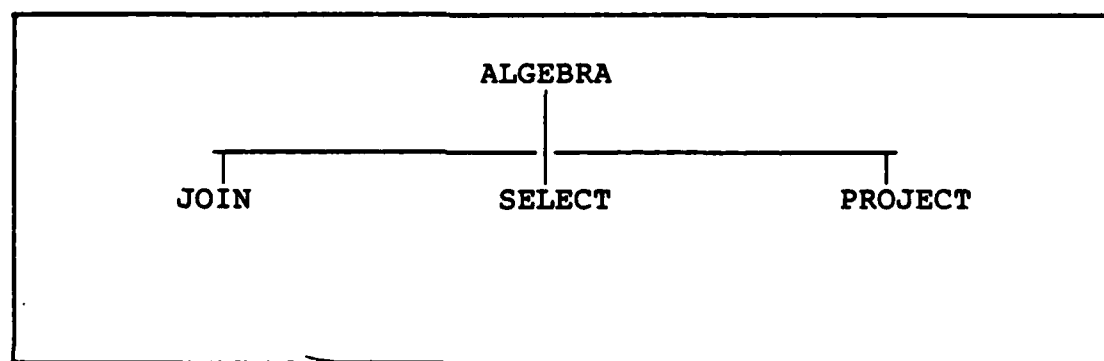


Figure 4. THE ALGEBRA MODULE

This algorithm totally depends upon the Uniqueness Assumption discussed in chapter 2. The Universal Relation is created by JOINing all the relations in the database. The query is not referenced for this step. Note, too, that the algorithm also depends on the Roth optimizer. JOIN is normally the most costly operation in terms of both time and space, therefore, optimization of the query to something less than the Universal Relation that will still answer the query is important (see chapter 5, Join Optimization).

The SELECTION criteria can be detected in many ways. The most obvious indicators of the attribute-operator-value

construct are:

1. An attribute follows the placeholder.
2. An attribute is followed by an operator.
3. A value follows an operator.
4. The whole construct is found.
5. An operator is found.

The last method was chosen as the indicator for the following reasons:

1. The operator can be detected in one character, by comparison with a set of only three characters (<, >, =).
2. The operator can be error checked in one step. Currently, since Roth only allows <, >, and =; the character following the operator must be blank.

The swift detection and error check of the operator takes care of one-third of the construct in minimal time, thus indicating presence of the construct and partial processing of it in two steps.

Testing

Testing showed that the interface responds as stated. The test was conducted in a top-down manner. Each major module was tested independently before insertion to the interface (minor modules were tested in conjunction with the major module they were created to support). The modules considered major were MAIN, GETATTRIBUTES, RECVQRY, PROCESSQRY, SAVEQRY, TRANSLATE, ALGEBRA, JOIN, SELECT, and

PROJECT. As each major module was inserted, the resultant program was subjected to an integration test. Finally, the total program was tested, using the test cases in appendix B. The database used for this test was the part/supplier database.

Test query 5 tries to compare SNAME and CITY - an obvious error - but no error occurs. This is because the interface checks attribute names for validity. SNAME could be defined as the name of the city the supplier's head office is in, while CITY could be the city the plant is in. In that case, the query would be valid.

Test query 6 tests the capability to pass a two word specific value to the Roth system. This two word specific value is not currently supported in Roth.

Test query 12 has no error, yet has no operator. Since no operator is given, the interface does not detect a SELECTION criterion, therefore, the attribute-operator-value construct is not violated. Also, no SELECT statement is created. As a result, CITY is PROJECTed out of the Universal Relation. Similarly, query 13 has no error because of the missing operator. However, the whole Universal Relation is PROJECTed because the placeholder WHERE separates the attribute CITY from the area reserved for the list of attributes to be PROJECTed.

Queries 14 and 15 are similar in that both seemingly have errors, but 14 does not produce an error. Query 15 produces an error because the word AND indicates that there

is another attribute-operator-value construct following. However, the operator (key to finding the construct) is missing. This situation is not discovered until after the word AND is reflected in the relational algebra being produced. Rather than close the temporary file, read up to the word AND, and process what is most assuredly a partial query, an error is generated.

Query 14, on the other hand, has no conjunctive word indicating another attribute-operator-value construct is following. Therefore, the relational algebra is produced with assurances that some useful information will be retrieved by the query. This is true due to the nature of the conjunctives available (AND, OR). If the user meant to insert AND, the query would be more restrictive with the second attribute-operator-value constraint. The query produced without the AND, and its next constraint, would retrieve all desired information with some unwanted information.

If the user meant to insert an OR conjunctive, the information retrieved by the produced query would be a subset of the information desired.

Finally, Table V presents a sample of disk space and time required to process a query using the part/supplier database.

Table V
TIME AND DISK SPACE REQUIRED

QUERY/TYPE	SECS	BLOCKS
SS	18	5
DS	20	7
TS	21	8
SP	16	3
DP	16	3
TP	17	3
SP/SS	19	6
DP/SS	19	6
TP/SS	19	6
SP/DS	21	8
DP/DS	21	8
TP/DS	22	8
SP/TS	22	9
DS/TS	22	9
TP/TS	22	9

The queries used for Table V follow:

SS (SELECTS on a Single attribute)

LIST FOR COLOR IS "RED".

DS (SELECTS on two attributes)

LIST FOR COLOR IS "RED" AND QTY IS "500".

TS (SELECTS on three attributes)

LIST FOR COLOR IS "RED" AND QTY IS "500" AND S# > WEIGHT.

SP (PROJECTS a Single attribute)

LIST SNAME.

DP (PROJECTS two attributes)

LIST SNAME, PNAME.

TP (PROJECTS three attributes)

LIST SNAME, PNAME, AND WEIGHT.

The other queries are combinations of these. For example,

TP/TS PROJECTS three attributes, and SELECTS on three attributes:

LIST SNAME, PNAME, AND WEIGHT FOR COLOR IS "RED" AND
QTY IS "500" AND S# > WEIGHT.

Chapter 5

Recommendations

NBS Conversion

The Roth system should be moved from UCSD Pascal. As discussed in chapter 3, UCSD Pascal is restricting further development of the system. Yet, UCSD Pascal does provide valuable string functions. NBS Pascal is a viable choice to move to, but strings are not accommodated. Conversion to another language could be worthwhile if string functions are available. However, none of the languages currently available to the Digital Engineering Lab except UCSD and C support strings. Unfortunately, the available C compiler is not a very good implementation. It is unreliable, and difficult to work with. Unless an appropriate language can be procured, the best option for further development is NBS Pascal.

String Functions

To support a move to NBS Pascal, consideration should be given to providing string functions in the NBS compiler. Source code to the compiler is contained in the NBS Pascal package. By creating functions in the NBS compiler providing string functions similar to the UCSD string functions, the amount of change required to make the Roth system run in the RT-11 operating system would be reduced.

This particular project would deal mainly with RT-11 assembly language programming. It would be ideal for someone interested in studying an operating system.

Memory Management

Each time the User-Friendly Interface is run, three linked lists are produced to act as the data dictionary. At the end of processing a query, the user has the option to cycle back to the beginning of the program to try another query. If this is done, three more linked lists are created since the interface cannot be sure the user did not change disks. Although the first three linked lists are not used, they still take up memory space. The computer has no way of knowing which areas of memory once used for the linked lists are no longer needed (NBS does not support the dispose function). With the database tested, memory was not filled when multiple queries were run. That is not to say that it might not happen when using the interface, though. Therefore, a management system for handling the memory used by the linked lists should be installed. Three methods can be used to implement the management system:

1. Implement the dispose function in the NBS compiler.
2. Add modules to the interface to track memory allocated and released.
3. Call an external program that chains back to the interface program.

Although the third method would work if the interface ran alone on a machine, if it were integrated into the Roth system as planned, this would not be advisable.

Join Optimization

Currently, the interface joins every relation in the database, requiring that the Roth optimizer to be used for efficient query resolution. Now that the interface runs and has proven its ability, it can be changed to optimize the relational algebra. The change entails allowing a JOIN of only those relations actually required to answer the query. The SELECT and PROJECT cannot be optimized, since they are already in minimal form. To optimize the JOIN, the following algorithm should be used:

1. Make a list of attributes required in the answer (used in PROJECT).
2. Add attributes to the list required in the SELECT statement.
3. Find the relation that has as many attributes from the list in it as possible. If more than one relation has the most attributes, use the relation with the least total number of attributes.
4. Put the found relation in a relation list.
5. If more than one relation is in the relation list, JOIN them, leaving only the resulting relation in the relation list.
6. Delete attributes from the attribute list that are in the relation in the relation list.
7. If any attributes are in the attribute list, go to step 3.

These steps ensure the fewest number of JOINS by taking only those relations actually required in the JOIN. Since the database scheme must have the lossless join property, the resulting subset of the Universal Relation is guaranteed to have the data required in it.

NBS Pascal on LSI-11/2

Professor Barr [3] related in a telephone conversation that there is a possibility to modify the NBS compiler in use on the LSI-11/23 for use in a non-floating point environment on the LSI-11/2. He stated that two instructions in the PASLIB.MAC code turn on the floating point instruction set, SETD and SETI. By deleting these two instructions, the floating point instruction set is not activated, making the NBS compiler compatible with either LSI machine. Naturally, the deletions would make the use of floating point numbers impossible. This change should be explored.

Professor Barr further stated that an NBS compiler for the LSI-11/2 is available, however it displays trouble with floating point expressions due to the way the LSI uses registers. Stack overflow is a frequent occurrence, unless the floating point expressions are broken down to the smallest possible pieces.

Efficiency Testing

No formal study of the disk file space or memory space required was done on the interface, nor was a speed of execution study run. Due to the size of the Roth system, a

study of how much memory is required to run the interface on different database sizes should be done. This would indicate whether it is feasible to incorporate the interface in the same file with the Roth USMAIN [28] code. Also, formal study of the size of command file the interface outputs will clarify how much disk space the user needs to have before running the interface. Finally, a speed of execution study of the interface will be useful in determining how to incorporate the interface into the Roth system.

Chapter 6

Conclusion

Review

The goal of this project was to provide a User-Friendly Interface to the Roth Relational Database. The result of the project is an operational interface, but only users of the interface can really judge its friendliness.

The Uniqueness Assumption of the Universal Relation database concept was introduced. Exploration revealed a capability to produce an interface of simple design. The Universal Relation concept was adopted, and the interface was designed and operating in under two months.

Hardware and software constraints were examined. It was discovered that NBS Pascal is a hardware dependent package, and has a limited capability. Still, the future of the Roth database system looks brighter on a machine soon to be in a network situation than on a single machine with memory limitations.

Constraints placed on the user in using the interface were presented. The discussion revealed that the interface depends as much on the direction the user is thinking as on the format of the query. Indeed, the format used is reflective of the direction the user should be thinking.

An introduction to the flow of the interface was given. Testing conducted and errors produced aided in understanding how the interface works, and revealed the thinking behind the purpose of the interface.

Finally, recommendations to improve the system were listed. Much more useful work can be accomplished in the Roth system.

Bibliography

1. Bachman, C. W. "Data Structure Diagrams," Database, The Quarterly Newsletter of the Special Interest Group on Business Data Processing of the ACM. Vol. 1, No. 2 1976.
2. Barr, John R., et al. "NBS Pascal User's Guide." Department of Computer Science, University of Montana, Missoula, Montana, 1981.
3. Barr, John R. Telephone conversation, 8 September 1983.
4. Beller, Aaron. "The Consequences of the Uniqueness Assumption for Relational Databases." Department of Decision Sciences, The Wharton School, University of Pennsylvania, 1980. (AD A093399)
5. Codd, E. F. "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, Vol. 13, No. 6, 1970.
6. Cousins, Thomas R. "A Methodology for the Inferential Derivation of Retrieval Semantics Utilizing a Relational View of a Meta-base." Doctoral Dissertation. University of Southwestern Louisiana, 1978.
7. Date, C. J. An Introduction to Database Systems (Third Edition). Reading, Mass.: Addison-Wesley Publishing Company, 1981.
8. Dell'Orco, P., V. N. Spadavecchia, and M. King. "Using Knowledge of a Data Base World in Interpreting Natural Language Queries," Information Processing 77. New York: North-Holland Publishing, 1977.
9. DeMarco, T. Structured Analysis and System Specification. New York: Yordan Press, 1976.
10. Fagin, R., A. O. Mendelzon, and J. D. Ullman. "A Simplified Universal Relation Assumption and its Properties," IBM Research Report (RJ2900). San Jose, California, 1982.
11. Goodman, Bradley Alan. "A Model for a Natural Language Data Base System." Unpublished MS thesis. University of Illinois, Urbana, Illinois, 1978. (AD A057641)
12. Guidry, Michael. "A minicomputer implementation of a Data Model Independent, User-Friendly Interface." Unpublished MS thesis. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio, 1982.

13. Hall, P. A. V. "Optimization of Single Expressions in a Relational Database System," IBM Journal of Research and Development, Vol. 20, No. 3 (1976).
14. Hallum, Maurice M. III. "A Relational-Based Data Management System for Engineering and Scientific Application," Systems Simulation and Development Directorate, US Army Missile Laboratory, 1980. (AD A86881)
15. Hammer, Michael and Stanley B. Zdonik, Jr. "Knowledge-Based Query Processing," Proceedings of the Sixth International Conference on Very Large Databases. Cambridge: 1980.
16. Harris, Larry R. "Natural Language Data Base Query," Dartmouth College, Hanover, New Hampshire, 1980. (AD A092500)
17. Jensen, Kathleen and Niklaus Wirth. Pascal User Manual and Report. New York: Springer-Verlag, 1974.
18. King, Jonathan J. "Query Optimization by Semantic Reasoning." Unpublished Doctoral Dissertation. Department of Computer Science, Stanford University, 1981. (AD A108735)
19. King, Jonathan J. "Quist: A System for Semantic Query Optimization in Relational Databases," Proceedings of the Seventh International Conference on Very Large Databases. Cannes: 1981.
20. Leong-Hong, Belkis W. and Bernard K. Plagman. Data Dictionary/Directory Systems Administration, Implementation, and Usage. New York: John Wiley and Sons, 1982.
21. Maier, David and Jeffery D. Ullman. "Maximal Objects and the Semantics of Universal Relation Databases." Department of Computer Science, Stanford University, 1981. (AD A113024).
22. Martin, J. Principles of Database Management. New Jersey: Prentice-Hall, 1976.
23. Microcomputers and Memories. Digital Equipment Corporation, Maynard, Maine, 1981.
24. Mau, James D. "Implementation of a Pedagogical Relational Database System on the LSI-11 Microcomputer." Unpublished MS thesis. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio, 1981.

25. Peters, Lawrence J. Software Design: Methods and Techniques. New York: Yourdan Press, 1981.
26. Petrick, S. R. "On Natural Language Based Computer Systems," IBM Journal of Research and Development, Vol. 20 No. 4, 1976.
27. Popa, J. H. "Relational Data Management." 1976. (AD A029892)
28. Rodgers, Linda M. "The Continued Design and Implementation of a Relational Database System." Unpublished MS thesis. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio, 1982.
29. Roth, Mark A. "The Design and Implementation of a Pedagogical Relational Database System." Unpublished MS thesis. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio, 1979.
30. Schank, Roger, Janet Kolodner, and Gerald DeJong. "Conceptual Information Retrieval." Department of Computer Science, Yale University, 1980. (AD A095372).
31. Smith, John M. and Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface," Communications of the ACM, Vol. 18, No. 10, (1975).
32. "UCSD Pascal System II.0 User's Manual." Institute for Information Systems, La Jolla, California, 1979.
33. Ullman, Jeffery D. Principles of Database Systems (Second Edition). Potomac, Maryland: Computer Science Press, 1982.
34. Weber, Herbert. "On the Unambiguous Use of Names in Database Design," Improving Database Usability and Responsiveness. New York: Academic Press, 1982.
35. Welk, Martin H. Standard Dictionary of Computers and Information Processing (Revised Second Edition). Rochelle Park, New Jersey: Hayden Book Company, 1977.

Appendix A

INTERMEDIATE LANGUAGE DEFINITION

query ::= ender | verb ender | verb object ender | verb
object qualifier ender | verb object placeholder
qualifier ender

object ::= {attribute {[,] attribute}}

qualifier ::= {attribute operator value {logical attribute
operator value}}

ender ::= .

placeholder ::= FOR | WHEN | WHERE

value ::= attribute | "database-entry"

database-entry ::= word | word word

operator ::= equal-to | greater-than | greater-than-or-
equal-to | less-than | not-equal | less-than-or-
equal-to

logical ::= AND | OR

equal- to ::= EQUALS | EQ | IS | EQUAL | =

greater-than ::= GREATER THAN | GT | >

greater-than-or-equal-to ::= GREATERTHANOREQUALTO | GTEQ | >=

less-than-or-equal-to ::= LESSTHANOREQUALTO | LTEQ | <=

less-than ::= LESSTHAN | LT | <

not-equal ::= NOTEQUAL | NE | <> | !=

Appendix B

Final Test

1. Query: .

Test for: Recognize ender, automatic verb insertion.

Expected Results: JOIN statements.

2. Query: LIST SNAME.

Test for: Proper simple PROJECT.

Expected Results: JOIN statements and PROJECT SNAME
statement.

3. Query: LIST SNAME, PNAME, AND CITY.

Test for: Proper complex PROJECT statement.

Expected Results: JOIN statements and PROJECT SNAME,
PNAME, CITY statement.

4. Query: LIST WHERE SNAME = "TOM".

Test for: Recognize placeholder, need for SELECT, quoted
value, and produce proper SELECT statement.

Expected Results: JOIN statements, SELECT WHERE SNAME =
TOM statement.

5. Query: LIST WHERE SNAME = CITY.

Test for: Recognize attribute value and produce proper
SELECT.

Expected Results: JOIN statements, SELECT WHERE SNAME =
CITY statement.

6. Query: LIST WHERE CITY = "NEW YORK".
Test for: Recognize two word quoted value.
Expected Results: JOIN statements, SELECT WHERE CITY =
NEW YORK statement.
7. Query: LIST CITY = "LA".
Test for: No placeholder.
Expected Results: JOIN statements, SELECT WHERE CITY = LA
statement, PROJECT CITY statement.
8. Query: LIST CITY IS "LA" OR STATUS > "40".
Test for: Recognize logical.
Expected Results: JOIN statements, SELECT WHERE ((CITY =
LA) OR (STATUS > 40)) statement,
PROJECT CITY statement.
9. Query: LIST SNAME, PNAME, AND CITY FOR STATUS > "60" AND
COLOR IS "RED".
Test for: Complex query.
Expected Results: JOIN statements, SELECT WHERE ((STATUS
> 60) AND (COLOR = RED)) statement,
PROJECT SNAME, PNAME, CITY statement.
10. Query: LIST CITY IS "LA".
Test for: Improper quoted value.
Expected Results: Failure to process query, statement
indicating = followed by LA, which has

no ending quote.

11. Query: LIST CITY IS DARK.

Test for: Illegal value DARK.

Expected Results: Failure to process query, statement
indicating operator is followed by
illegal value.

12. Query: LIST CITY "DARK".

Test for: Missing operator, without placeholder.

Expected Results: JOIN statements, PROJECT CITY
statement.

13. Query: LIST WHERE CITY "DARK".

Test for: No operator, with placeholder.

Expected Results: JOIN statements.

14. Query: LIST CITY IS "DARK" SNAME IS "TOM".

Test for: No logical.

Expected Results: JOIN statements, SELECT WHERE CITY =
DARK statement, PROJECT CITY
statement.

15. Query: LIST CITY IS "DARK" AND SNAME "TOM".

Test for: Compound SELECT with no second operator.

Expected Results: Failure to process query, statement
indicating the logical has a
constraint with no operator.

16. Query: LIST COW IS "FAT".

Test for: Illegal attribute COW, without placeholder.

Expected Results: Failure to process query, statement
indicating ALL is not an attribute.

17. Query: LIST WHERE COW IS "FAT".

Test for: Illegal attribute COW, with placeholder.

Expected Results: Failure to process query, statement
indicating WHERE is not an attribute.

Appendix C

How MAKDIC Works

MAKDIC creates the files STD.DIC and DATBAS.DIC for use by the User-Friendly Interface. STD.DIC contains operators and key words used in querying any database. DATBAS.DIC contains attribute aliases (abbreviations and possible incorrect spellings) for a specific database. Thus, STD.DIC need only be created once, while DATBAS.DIC needs to be created for each database created.

MAKDIC is very easy to run. It interacts in the following manner:

ADD OR CREATE A DICTIONARY? (A OR C)

Response A results in:

MAKE SURE PROPER FILE IS ON DEFAULT DRIVE

Since the user indicated a desire to add information to an existing dictionary, he must be sure the disk with the file he wants to add to is on the proper drive. Following this, the user is asked:

ADD TO STANDARD OR DATABASE DICTIONARY? (S OR D)

The response causes either STD.DIC (response S) or DATBAS.DIC (response D) to be searched for on the default drive.

If the response to the question (above):

ADD TO OR CREATE A DICTIONARY? (A OR C)

had been C, the program would have come back with:

FILE WILL BE PUT ON DEFAULT DRIVE

Following this, the user is asked:

CREATE STANDARD OR DATABASE DICTIONARY? (S OR D)

The file indicated will be created on the default drive.

Regardless of the responses above, the rest of the program works the same. The user is prompted:

INPUT WORD TO DEFINE

The program accepts any input, up to 20 characters, then prompts with:

INPUT ALIAS (REPLACEMENT) FOR INPUT WORD

Again, a 20 character maximum length input is acceptable.

This word is the "definition" of the previous word.

Finally, the user is prompted:

DONE? (Y OR N)

Only a response of N will cause the program to loop back for another input word. Any other response causes the file to be closed, and the program to end.

Appendix D

What NATQRY Really Does

The User-Friendly Interface is contained in the files NATQRY.<ext>. NATQRY is short for Natural Query. Running NATQRY.SAV (.SAV files are RT-11 executable code files) is accomplished by typing either:

RUN NATQRY
or
R NATQRY

The command RUN is for executing files on the default drive (usually DY1:), while R (short for RUN) is for executing files on the boot drive (DY0:).

Following is a step-by-step account of what NATQRY does:

Step 1 - Request user's last name.

The purpose of this step is to get an identifier to use in database security. Since the security features are not yet implemented, no response other than a carriage return is required. However, a name can be entered.

Step 2 - Inform user that disk space is required on the default drive.

It is assumed that the default drive is DY1:. The disk space required is relatively small, but if no disk is in drive DY1:, or the disk there is nearly full, the opportunity is given to change disks. Typing the letter 'N' tells the system to pause for a disk change in the default drive. Only typing the letter 'Y' will allow the program to continue, once the disk is changed.

Step 3 - Request location of files SETUP.DAT,
DATBAS.DIC, and STD.DIC.

Each file location is requested in the following manner:

IS <FILENAME> ON THE BOOT DRIVE? (Y OR N)

The question for each file must be answered either by the letter 'N' or the letter 'Y'. Any other response will cause the question to be reasked.

A 'Y' response indicates that a particular file is on drive DY0: (the boot drive). An 'N' response indicates the file is on the disk in drive DY1:.

SETUP.DAT is the file created by the Roth DDL program as SETUP.DATA. If this file does not exist, trying to run NATQRY is futile. The other two files, STD.DIC and DATBAS.DIC, are created by the program MAKDIC. At least STD.DIC must be available to properly run NATQRY. If DATBAS.DIC is unavailable, an error message indicating inability to open a file will be produced, but as long as all attributes are correctly spelled (correct means the spelling in the file SETUP.DAT), NATQRY can successfully run. The dictionaries are created when the file locations are all received.

Step 4 - List the five rules for making a query.

The rules are:

1. A verb must be the first word in the query.
2. The rest of the query must be in sentence format an use provided attribute names wherever possible.
3. Specific attribute names must be in quotes, such

as NAME IS "JONES".

4. End of query must be signalled by a period.
5. The query must not be more than 4 lines long (320 characters, including spaces and returns).

After reading the rules, hit the RETURN key to continue.

Step 5 - List the attributes for the user.

The file SETUP.DAT is accessed to show the attributes of the database. The attributes are listed in the order they are encountered in the file, and put in four columns across the screen.

Step 6 - Receive the user's query.

Input a query by following the five rules.

Step 7 - Show user the query.

When a period is encountered, the system shows the query as it is understood. The query is translated to its intermediate form before it is shown to the user. If the query does not correctly reflect the intent of the input query, type 'N'. Any other response is taken as an indication of a correct query.

An incorrect query indication sends the system to Step 8, while a correct query indication takes the system to Step 10.

Step 8 (Query was incorrect) - Show user instructions for changing the query.

The instructions are:

Input a string of characters uniquely identifying the part of the query you want to change. Maximum length of the string is one line (80 characters).

THE QUERY WILL BE OVERLAID STARTING AT THE FIRST CHARACTER YOU INPUT.

Input string (end with <CR>):

Determine which character (in the query the system showed) marks the beginning of the incorrect information. Input that character followed by as many characters as required from the showed query to identify that string as its first occurrence. As an example, suppose the system said it received:

SHOW BILLY A VOLLEYBALL

If the word VOLLEYBALL should have been FOOTBALL, enter a V. Since there is only one occurrence of that letter, the change would start with the space holding the V. Similarly, if BILLY should be BIRD, the letter L (or string LL) would indicate that the change starts at the first L in BILLY.

Hitting the RETURN key after the string moves the program to the next step.

Step 9 (User is changing the query) - Prompt user for input to change query.

The system displays this message:

Input changes (end with .):

The system saved the query up to the spot indicated in Step 8. The rest of the query is gone. Any characters input will be appended to what the system saved. In the example of changing VOLLEYBALL to FOOTBALL, the input here would have to be FOOTBALL, even though BALL was already in the

original query beyond the part changed. After typing the period, the system goes back to Step 7.

Step 10 (Step 7 query was approved) - Process the query.
The system displays the message:

PROCESSING YOUR QUERY

while it transforms the query to relational algebra. If errors are encountered, the screen is changed so that only the error message is displayed, followed by the message:

Hit RETURN to continue

See Section 2 for error message meanings. If no errors are encountered, Step 11 is performed; otherwise Step 13 is performed.

Step 11 (No errors in the query) - Show user the relational algebra produced.

Following the relational algebra statements is the question:

Would you like to save it as a command file? (Y or N)

If the relational algebra is unwanted, type 'N'. Any other response causes Step 12 to be performed.

Step 12 (Save the relational algebra) - Request filename.

The system shows this message:

Input a disk filename to save query in.
If a disk drive is not specified, the default drive will be used:

Type in a filename using the format <DRIVE NAME .EXTENT>.

Where:

DRIVE is optional. If used, it must be either DY0: or DY1:. DY0: is the boot drive, DY1: is the default drive. If not used, DY1: is assumed.

NAME is required. It can be from one to six characters long. The first character must be a letter, but subsequent characters may be letters or numbers (alphanumeric).

.EXTENT is optional. If used, the period must be the first character. The rest of EXTENT is from one to three letters.

The name of the file actually consists of NAME.EXTENT; DRIVE is used only to determine which disk to put the file on.

Step 13 - Translate another query.

The user is asked if he wants to translate another query:

Would you like to try another query,
possibly on another database? (Y or N)

Only a response of 'N' will end the program. Any other response will loop back to Step 3.

Disks (databases) may be changed at this time. Do not change disks after giving the location of the second file. Also, only change disks when the system is awaiting a response.

Appendix E

What Error Messages Mean

RI-11 OPERATING SYSTEM ERRORS

?ERROR OPENING A FILE

- Running NATQRY - STD.DIC, SETUP.DAT, or DATBAS.DIC was not on the drive indicated. The program will continue to run, but translation to relational algebra will not be successful.
- Running on an LSI-11/2 - Possibly, the drive name in NATQRY.PAS has not been changed from DY0: to DX0:, or from DY1: to DX1:. Check modules GETATTRIBUTES and FORMDICTIONARY. (If the drive names have not been changed in NATQRY.PAS, then check MAKDIC.PAS for the same changes.)
- Running any NBS Pascal program - A file referenced in the program by RESET or REWRITE was not on the disk drive specified. Make sure that the filename is either in quotes ("NATQRY.PAS") or that the array passed to RESET or REWRITE has a character zero (chr(0)) immediately following the actual filename.

Example:

```
Program sample;
var afile : text;
    filename : array[1..15] of char;
begin
    |           (statements)
    filename[1] := 'N';
    filename[2] := 'A';
    |           (spell NATQRY.PAS)
    filename[7] := '.';
```

```

filename[8] := 'P';
filename[9] := 'A';
filename[10] := 'S';
filename[11] := chr(0);
|           (more statements)
reset(afile, filename);
|           (finishing statements)
end.

```

?ERROR WHILE DOING A BREAK

-- Running Pass one of the NBS Pascal compiler - One of the files being created was not allocated enough disk space.

Example:

```

.R PASS1
*- NATQRY.PAS INT[30] DAT[10] LST[220]

```

?ERROR WHILE DOING A BREAK

The error says that either INT, DAT, or LST requires more space than allocated (30, 10, 220 BLOCKS - See NBS Compiler Notes) or the disk does not have enough contiguous space to accommodate one or more of the requested file sizes.

NAIRY MESSAGES

UNABLE TO PROCESS YOUR QUERY DUE TO ERROR

- Standard error message header.

<op> is an illegal operator.

The Roth system only allows <, >, or =

- <op> is one of the following: <>, <=, >=

The interface understands not equal, less than or equal to, and greater than or equal to; but the Roth system does not.

<op> is followed by <word>
which has no ending quote

- <op> is: <, >, or =

<word> is a word from the input query that has a starting quote. Since no ending quote was given, the interface cannot tell if this one word is the whole specified value, or if the next word should be part of the specified value. That is, if <word> was "NEW, the interface cannot tell if that is the first part of "NEW YORK" or just "NEW".

Operator <op> is followed by a
value that is neither an attribute, nor quoted

- <op> is: <, >, or =

An unclear rule on how to select which
information to retrieve was given.

Example:

LIST PNAME FOR CITY IS DENVER.

DENVER is not quoted, indicating it is
not a value found in the database under the

name CITY. However, DENVER is also not an attribute in the database. The interface does not know where to find what DENVER is; and without knowing, the correct PNAME cannot be found.

Operator <op> is not preceded by an attribute

- <op> is: <, >, or =

Similar to the error above, except that the word in front of <op> is not an attribute.

Logical <log> has invalid constraint following it.
The constraint has no operator.

- <log> is AND or OR

A query indicating more than one rule on which data to select was input. However, a rule following <log> had no operator (<, >, or =) in it.

Example:

LIST PNAME FOR CITY IS "DENVER" AND SNAME "S2".

Only PNAMEs associated with DENVER and S2 are desired, but it is unclear whether SNAME is more, less, or equal to S2. The constraint CITY IS "DENVER" is legal, the constraint SNAME "S2" is not.

Appendix F

NBS Compiler Notes

PASGUI.DOC is the disk file sent with the NBS Pascal package that explains what NBS does and doesn't do. The few notes here are covered in PASGUI.DOC, but are not as easy to find in that document - nor as easy to understand.

What NBS Pascal Does NOT Support

Chaining	DISPOSE	GOTO
LABEL	*PACKED ARRAY	STRINGs
String Functions	**Passing Procedure/Function Parameters	

The Roth Database system uses string functions (Position, Concatenate, etc.) and some GOTOs. Changes must be made to the Roth code to transfer the system to NBS Pascal.

* All character arrays are PACKED. Therefore, single statement compares and assignments of the same TYPE arrays are supported:

```
program compare;
var
  dog, cat : packed array[1..3] of char;

begin
  dog := 'DOG';
  cat := 'CAT';
  if (dog = cat)
  then writeln('Dog is weird')
  else writeln('Dog is normal')
end.
```

** Paragraph 3.9 of PASGUI.DOC references passing procedural and functional parameters. The statements are

not obvious in their meaning. However, the following program segment depicts a situation that does not run properly, because it violates this paragraph:

```
program linked_list;
type
  list = record
    name : array[1..20] of char;
    ptr : ^list
  end;

var
  base, temp : ^list;
  word : array[1..20] of char;

procedure walk(var POINTER : ^list);
begin
  if (POINTER^.name <> word) { Chain down the list }
  then walk(POINTER^.ptr)
end;

begin
  |
  |
  |
end.
```

The program will compile without error, but the procedure will not properly chain down the list. This is because the variable POINTER was declared in the procedure header, making it a procedural parameter. Subsequent passing of POINTER to any other procedure (even WALK) is illegal. A test of this recursive call setup to chain down a linked list resulted in the procedure calling itself in an infinite loop. By looking at POINTER addresses, it was determined that WALK properly chained two links down the list, then reset itself to the beginning of the list.

What NBS Added to Pascal

BREAK is a function added to Pascal for output to a file. Write statements to any file are buffered and held until the buffer is full, or a BREAK to the file is issued. This allows the program to run slightly faster by not interrupting flow for I/O until required. In a PROMPT-RESPONSE situation, if BREAK(output) is not issued, the user will be required to respond to a prompt he did not receive. An example follows:

```
program prompt;
var
  name : array[1..15] of char;
begin
  writeln('ENTER NAME');
  readln(name)
end.
```

When this program is run, nothing seems to happen. After the user enters some characters and a carriage return, the statement ENTER NAME appears; and the program ends. By inserting "break(output);" before the readln, ENTER NAME will appear before the user can enter his name.

Running the NBS Compiler

Finally, a few notes on running the compiler. The files that make up the compiler have been renamed in the Digital Engineering Lab (DEL):

PASGUI.DOC

PASS1F.SAV
PASS2F.SAV
NPASL.OBJ
CPASL.OBJ

AEI1/DEL

P1.SAV
P2.SAV
NBSLIB.OBJ
CPASL.OBJ

These files are compatible with the LSI-11/23 (Programs created with these files will only run on LSI-11/23 machines). Different names should be given the files for the LSI-11/2.

File size and disk space are major concerns when running the NBS compiler. Merely entering a command such as:

```
.R P1
*- PROG.PAS INT DAT LST
```

is only sufficient for small files. The compiler needs to know how large INT, DAT, and LST are going to be. For a fully commented program with module headers, PROG.PAS can overfill the default sizes given to INT, DAT, and LST when it is only 26 to 30 blocks long. (INT and DAT are intermediate files used by pass two to create the program's object file. PASGUI.DOC suggests using the names INT, DAT, and LST rather than PROG.INT, PROG.DAT, and PROG.LST. LST is an error listing file. All three files must be specified for pass one to run, and the dash must both be flush against the asterisk and have a space between it and INT. Pass one supplies the asterisk as a prompt.) During pass one, if one of the files should be filled, the system error ?ERROR WHILE DOING A BREAK is produced. This error is sent to the CRT for almost every line of code encountered after the file is filled, giving an indication of how near the end pass one got.

To remedy the full file error, square brackets are used

to give the system maximum file sizes:

```
.R P1
*- PROG.PAS INT[20] DAT[10] LST[100]
```

This command lets INT take up to 20 contiguous blocks before it fills. Naturally, DAT gets 10 blocks, and LST gets 100. Guessing what size to allow is a trick. For NATQRY.PAS (the User-Friendly Interface file), the following file sizes were used:

File	Size
NATQRY.PAS	142
INT	29
DAT	6
LST	210

As NATQRY grew during development, this rule of thumb was developed "Allow INT 20% of the size the source file has, let LST have 150% of source, and give DAT 20 blocks". DAT fluctuated, but never got as large as 15 blocks, however other compiled files have reached 18 blocks for DAT. All of these files, naturally, require contiguous disk space. Frequent use of the SQUEEZE command is highly recommended.

The second pass of the NBS compiler never required specifying the file size, however, the option is available:

```
.R P2
*- INT DAT PROG.OBJ[62]
```

Again, like pass one, the asterisk is a supplied prompt, and the dash must be placed exactly as shown (unless options are used, in which case option letters would immediately follow the dash).

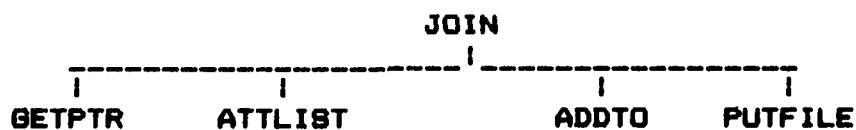
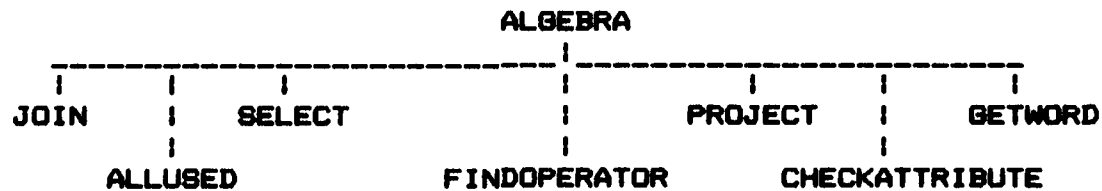
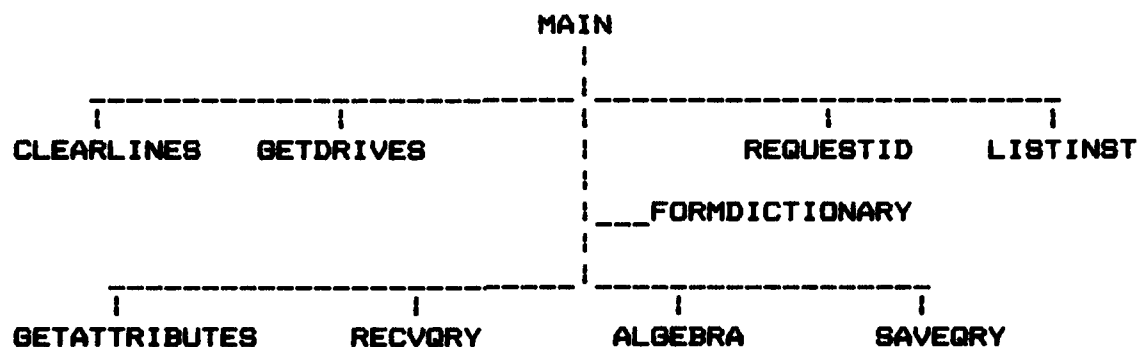
The LINK is the standard RT-11 LINK. However, if the pascal program being created sends very many messages to the CRT, the LINK command will need to use the BOTTOM option. This is because of the way RT-11 uses registers in Input/Output. The stack overflows quite rapidly. The default stack has its top at address 0400, and bottom at 1000. It can be lengthened by moving the bottom of the stack, as follows:

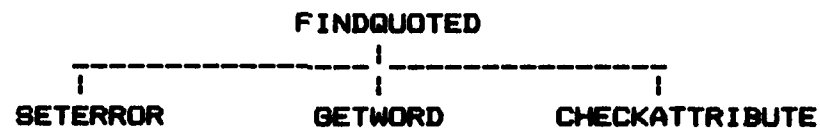
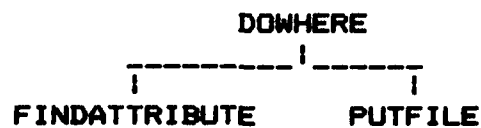
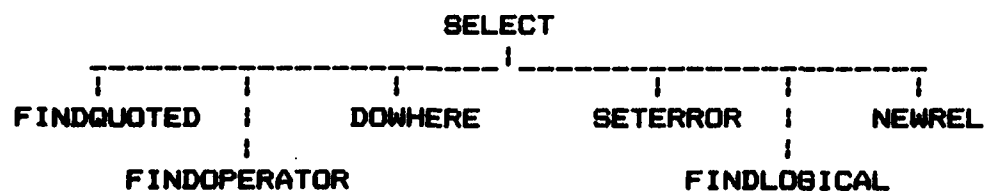
```
.LINK/BOTTOM:004000 PROG,NBSLIB
```

This command moves the bottom of the stack to address 4000, increasing the stack size by a factor of about 4. This happens to be the size stack NATGRY uses.

Appendix G

NATQRY STRUCTURE





Appendix H

```
(*****  
*   DATE: 5 Sep 83  
*   FILE: NATQRY.PAS  
*   FUNCTION: Interfaces with user to allow him to query the  
*             Roth Relational Database without knowing  
*             Relational Algebra. The actions taken are  
*             based on the Universal Relation idea. See  
*             Thesis written by Capt Dale VanKirk in 1983  
*             for further explanation.  
*   FILES READ: SETUP.DAT, STD.DIC, DATBAS.DIC  
*               SETUP.DAT - The file created by the Roth DDL  
*               package. It contains domain,  
*               relation, and attribute  
*               definitions.  
*  
*               STD.DIC - The STANDARD DICTIONARY file  
*               created by a separate program  
*               named MAKDIC. MAKDIC is written in  
*               Pascal (NBS), and is to be used by  
*               the DATABASE ADMINISTRATOR to  
*               create the dictionaries required  
*               to run this program.  
*  
*               DATBAS.DIC - Also made by MAKDIC -- This  
*               dictionary is specific to the  
*               database in use. It contains  
*               aliases for attributes.  
*  
*   FILES WRITTEN: QRYTMP.TMP, and a User specified file.  
*  
*   THE USER SPECIFIED FILE EQUATES TO A ROTH COMMAND FILE  
*  
*       QRYTMP.TMP - A temporary file written to  
*       hold all of the relational  
*       algebra commands made to  
*       fulfill the Natural language  
*       query. It is created on the  
*       default disk, and rewritten  
*       (erased, however name is  
*       left on disk directory) when  
*       no longer needed.  
*       User specified - File named by the user  
*       to hold the relational  
*       algebra in. This allows the  
*       user to build many queries  
*       at running of NATQRY. Disks  
*       can be changed allowing  
*       different databases to be  
*       queried.  
*   AUTHOR: VanKirk  
*****)
```

```

program naturalquery;

const
  trash = '@';
  space = ' ';
  ender = '.';

type
  short = packed array[1..20] of char;

  listrel = record
    relname : short;
    used : char;
    atrpointer : ^list;
    relpointer : ^listrel;
  end;

  listatr = record
    atrname : short;
    pointeratr : ^listatr;
    pointerrel : ^list;
  end;

  list = record
    identifier : short;
    pointer : ^list;
  end;

var
  atrbase, atrptr : ^listatr;    ( List keyed on attributes )
  relbase, relptr : ^listrel;    ( List keyed on relations )
  atrtemp, reltemp : ^list;      ( atrtemp points to
                                  attributes in list keyed on
                                  relations. reltemp points to
                                  relations in list keyed on
                                  attributes. )

  query : packed array[1..320] of char;
                                  ( The Query )
  j : integer;                    ( Counter for stepping thru query )
  length : integer;              ( Actual length of query )
  stopprocess : boolean;         ( Stop program if no database )
  setdrive : char;              ( Answer to question "Is SETUP.DAT
                                  on BOOT DRIVE?" )
  stddrive : char;              ( Is STD.DIC on BOOT DRIVE? )
  datdrive : char;              ( Is DATBAS.DIC on BOOT DRIVE? )
  lines : integer;              ( Number of clean lines to append
                                  on screen )
  continue : boolean;           ( Lets user loop through program
                                  to create another command file
                                  using same or different database )
  ch : char;                    ( To receive user response in MAIN )

  database, stdbase, dattemp, stdtemp : ^list;
                                  ( Standard dictionary (stdbase)

```

and database dictionary (database) }

```
word : short;           { Holds word while checking dictionaries }
endquery : boolean;      { Indicates end of original query reached }
calltranslate : integer; { Counter-type flag indicating need to check
                           for first verb }
newquery : packed array[1..320] of char;
                           { Intermediate query between input and
                             Relational Algebra }
nq : integer; { Counter to step thru newquery }

{*****}
procedure changeqry; forward;
{*****}
```

```

(*****
* DATE: 22 Aug 83 *
* MODULE: Clearlines *
* FUNCTION: Appends specified number of *
*           blank lines to screen. *
* INPUTS: lines *
* OUTPUTS: none. *
* LOCAL VARIABLES: i *
* GLOBALS USED: none. *
* MODULES CALLED: none. *
* CALLED BY: main, getdrives, requestid,*
*           listinst, formdictionary, *
*           recvqry *
* AUTHOR: VanKirk *
*****)

```

```

procedure clearlines;

```

```

var
  i : integer;

```

```

begin
  for i := 1 to lines do writeln;
  break(output)
end;

```

```

(*****
*   DATE: 22 Aug 83
*   MODULE: Getdrives
*   FUNCTION: Retrieves information on
*             which drive has files needed*
*   INPUTS: User supplied
*   OUTPUTS: setdrive, stddrive, datdrive *
*   LOCAL VARIABLES: getinfo
*   GLOBALS USED: setdrive, stddrive, dat-#
*             drive, lines
*   MODULES CALLED: clearlines
*   CALLED BY: main
*   AUTHOR: VanKirk
*****)

```

```

procedure getdrives;

```

```

var
  getinfo : boolean;

```

```

begin
  getinfo := true;      ( Find out drive relation/attribute )
  while getinfo do      ( definitions are on )
    begin
      writeln('Is SETUP.DAT on the BOOT DRIVE? (Y or N)');
      break(output);
      lines := 12;
      clearlines;
      readln(setdrive);
      if ((setdrive = 'Y') or (setdrive = 'N'))
        then getinfo := false
      end;
    end;

```

```

  clearlines;

```

```

  getinfo := true;      ( Find out drive standard dictionary )
  while getinfo do      ( definitions are on )
    begin
      writeln('Is STD.DIC on the BOOT DRIVE? (Y or N)');
      break(output);
      clearlines;
      readln(stddrive);
      if ((stddrive = 'Y') or (stddrive = 'N'))
        then getinfo := false
      end;
    end;

```

```

  clearlines;

```

```

  getinfo := true;      ( Find out drive data base dictionary )
  while getinfo do      ( definitions are on )
    begin
      writeln('Is DATBAS.DIC on the BOOT DRIVE? (Y or N)');
      break(output);
      clearlines;
      readln(datdrive);
    end;

```

```
        if ((datdrive = 'Y') or (datdrive = 'N'))  
            then getinfo := false  
        end  
    end;
```

```

*****
*   DATE: 11 Aug 83                                     *
*   MODULE: Getattributes                               *
*   FUNCTION: Accesses file SETUP.DAT and              *
*               creates structures to use               *
*               as database dictionary. One             *
*               structure ( relbase root )             *
*               keyed on relations, the                 *
*               other ( atrbase root ) is              *
*               keyed on attributes. Also              *
*               outputs a list of attributes*          *
*               to help user make query.              *
*   INPUTS: none.                                       *
*   OUTPUTS: 2 record structures with                  *
*               relbase and atrbase as roots           *
*   LOCAL VARIABLES: ch, i, k, count,                 *
*               constraint, atfile,                   *
*               relationend, savechar                  *
*   GLOBALS USED: stopprocess                          *
*   MODULES CALLED: none.                              *
*   FILE READ: SETUP.DAT                              *
*   CALLED BY: main                                     *
*   AUTHOR: VanKirk                                    *
*****}

```

```

procedure getattributes;

```

```

var

```

```

    atfile : text;
    count, i, k : integer;
    relationend, constraint : boolean;
    ch, savechar : char;
    holder : short;

```

```

begin

```

```

    new(atrbase);
    new(relbase);
    atrptr := atrbase;
    relptr := relbase;
    atrptr^.pointeratr := nil;
    atrptr^.pointerrel := nil;
    relptr^.atrpointer := nil;
    relptr^.relpointer := nil;

```

```

    if (setdrive = 'Y')
    then reset(atfile, "DY0:SETUP.DAT")
    else reset(atfile, "DY1:SETUP.DAT");

```

```

    i := 1;
    read(atfile, ch);
    while (ord(ch) <> 13) do                ( Read first line )
    begin
        holder[i] := ch;
        if not eof(atfile) then read(atfile, ch);
        i := i + 1;
    end

```

```

end;

while (i < 21) do
begin
    holder[i] := ' ';
    i := i + 1
end;

savechar := ' ';

if (holder <> 'NODOMAINSDEFINED')
    ( There is a database, skip domain definitions )
then
    while (not eof(atfile) and (ch <> '*')) do
    begin
        readln(atfile, ch);
        if (not eof(atfile) and (ch = '*'))
            then read(atfile, savechar);
        if (savechar = '*')
            then
            begin
                stopprocess := true;
                ( Domains defined, but no relations!! )
                writeln('SORRY, NO RELATIONS DEFINED');
                while not eof(atfile) do read(atfile, ch)
            end
        end
    else
    begin
        writeln('SORRY, NO DATABASE DEFINED');
        stopprocess := true;
        while not eof(atfile) do read(atfile, ch)
    end;

if ((not eof(atfile)) and (ch = '*') and (savechar <> '*'))
then ch := savechar;
    ( Savechar holding first letter of relation name )

if not eof(atfile)
then
begin
    ( Save relation name )
    k := 1;
    while (ord(ch) <> 13) do ( Check for end of line )
    begin
        relptr^.relname[k] := ch;
        read(atfile, ch);
        k := k + 1
    end;

    while (k < 21) do
    begin
        relptr^.relname[k] := ' ';
        k := k + 1
    end;

```

```

        read(atfile, ch)                ( Get rid of line feed )
    end;

    if not eof(atfile) then read(atfile, ch);
                                ( Read first letter of attribute )

    new(relptr^.atrpointer);
    atrtemp := relptr^.atrpointer;

    while not eof(atfile) do
    begin ( Save attribute name in list keyed on relations )
        k := 1;
        while (ord(ch) <> 13) do      ( Check for end of line )
        begin
            atrtemp^.identifier[k] := ch;
            read(atfile, ch);
            k := k + 1;
        end;

        while (k < 21) do
        begin
            atrtemp^.identifier[k] := ' ';
            k := k + 1;
        end;

        atrtemp^.pointer := nil;
        atrptr := atrbase;
        ( See if attribute already in list keyed on attributes )
        while ((atrptr^.atrname <> atrtemp^.identifier)
            and (atrptr^.pointeratr <> nil)) do
            atrptr := atrptr^.pointeratr;

        if (atrptr^.atrname = atrtemp^.identifier)
            then reltemp := atrptr^.pointerrel
                                ( Already in list )
            else
                begin
                    atrptr^.atrname := atrtemp^.identifier;
                                ( Now its in list )
                    write(atrptr^.atrname);    ( Output to user )
                    break(output);
                end;

        if atrptr^.pointerrel = nil
            then ( Put relation name in list keyed on attributes )
                begin
                    new(atrptr^.pointerrel);
                    reltemp := atrptr^.pointerrel;
                    reltemp^.pointer := nil;
                    reltemp^.identifier := relptr^.relname;
                end
            else
                begin
                    while (reltemp^.pointer <> nil) do

```

```

        reltemp := reltemp^.pointer;
        new(reltemp^.pointer);
        reltemp := reltemp^.pointer;
        reltemp^.pointer := nil;
        reltemp^.identifier := relptr^.relname
    end;

    for i := 1 to 3 do readln(atfile, ch);
                                { Skip sort info }
    if (ch = 'N')
        then constraint := false    { Constraint defined ? }
        else constraint := true;

    if constraint                { Skip constraint info }
        then for i := 1 to 2 do readln(atfile, ch);

    read(atfile, ch);
        { Either read * for end of relation definition,
          or first letter of another attribute }
    if (ch = '*')
        then relationend := true
        else relationend := false;

    if relationend
    then
        begin                    { Skip password section }
            read(atfile, ch);
            while (ch <> '*') do readln(atfile, ch);
            for i := 1 to 7 do readln(atfile, ch);
            read(atfile, ch);
            { Either read * for end of database definition,
              or first letter of new relation name }
            if ((ch = '*') and not eof(atfile))
            then
                begin
                    while not eof(atfile) do read(atfile, ch);
                    atrtemp^.pointer := nil
                end
            else
                if (ch <> '*')
                then
                    begin
                        k := 1;
                        new(relptr^.relpointer);
                        ( Get ready for another relation name in relbase list )
                        relptr := relptr^.relpointer;
                        relptr^.relpointer := nil;
                        new(relptr^.atrpointer);
                        atrtemp := relptr^.atrpointer;
                        while (ord(ch) <> 13) do
                            { Put relation name in list }
                            begin
                                relptr^.relname[k] := ch;
                                read(atfile, ch);
                                k := k + 1
                            end
                        end
                    end
                end
            end
        end
    end

```

```

        end;

        while (k < 21) do
            begin
                relptr^.relname[k] := ' ';
                k := k + 1
            end;

            read(atfile, ch);
                { Get rid of line feed }
            read(atfile, ch)
                { Get first letter of attribute }
        end
    end;

    if not relationend
    then
        begin
            new(atrtemp^.pointer);
            atrtemp := atrtemp^.pointer;
            while (atrptr^.pointeratr <> nil)
                do atrptr := atrptr^.pointeratr;
                    { Get to end of list }

            new(atrptr^.pointeratr);
            atrptr := atrptr^.pointeratr;
            atrptr^.pointeratr := nil;
            atrptr^.pointerrel := nil
        end
    end;

    ( THE FOLLOWING SECTIONS ARE FOR DEBUGGING USE )

    ( THIS SECTION PRINTS RELATION NAME AND ATTRIBUTES
      FROM RELATION BASED LIST )
    ( relptr := relbase;
      while (relptr <> nil) do
          begin
              atrtemp := relptr^.atrpointer;
              writeln('RELATION');
              writeln(relptr^.relname);
              break(output);
              while (atrtemp <> nil) do
                  begin
                      writeln(atrtemp^.identifier);
                      break(output);
                      atrtemp := atrtemp^.pointer
                  end;
              relptr := relptr^.relpointer
          end;
      )

    ( THIS SECTION PRINTS OUT ATTRIBUTES
      FROM ATTRIBUTE BASED LIST )
    ( atrptr := atrbase;
      while (atrptr <> nil) do
          begin

```

```
        writeln(atrptr^.atrname, 'Z');  
        break(output);  
        atrptr := atrptr^.pointeratr  
    end;  
    break(output) }  
end;
```

AD-A138 153

USER-FRIENDLY INTERFACE TO THE ROTH RELATIONAL DATABASE 2/2

(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH

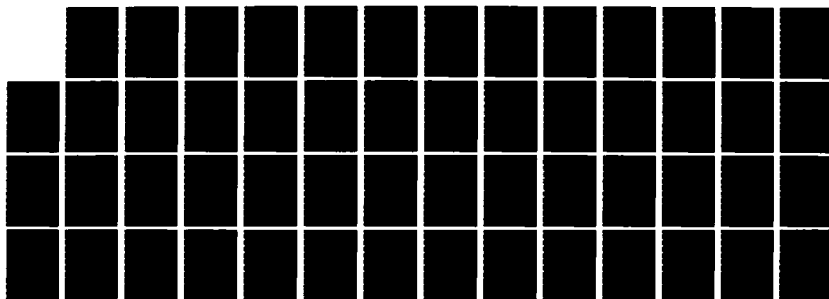
SCHOOL OF ENGINEERING D W VANKIRK 16 DEC 83

UNCLASSIFIED

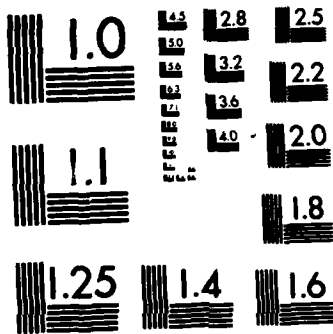
AFIT/GCS/EE/83D-21

F/G 9/2

NL



END
1
FILMED
3
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```

*****
* DATE: 3 Aug 83 *
* MODULE: Requestid *
* FUNCTION: Gets user name. *
* INPUTS: none. *
* OUTPUTS: none. *
* LOCAL VARIABLES: i, ch, name *
* GLOBALS USED: lines *
* MODULES CALLED: clearlines *
* CALLED BY: main *
* AUTHOR: VanKirk *
*****

```

```

procedure requestid;

```

```

var

```

```

    i : integer;
    ch : char;
    name : short;      ( For future use in database security )

```

```

begin

```

```

    writeln('Welcome to the Natural Query Interface');
    writeln;
    break(output);
    writeln('PLEASE END ALL INPUTS WITH A RETURN');
    writeln;
    break(output);
    writeln('Enter your last name');
    lines := 11;
    clearlines;

```

```

    for i := 1 to 15 do name[i] := ' ';      ( Clear name )

```

```

    i := 1;
    while (i < 16) do      ( Get name, stop on full name )
        begin
            read(ch);
            if (ord(ch) = 10) then i := 17;      ( Stop on <CR> )
            if (i < 16) then name[i] := ch;
            i := i + 1;
        end;
    lines := 24;
    clearlines;

```

```

    writeln('THIS IS IMPORTANT : IF THE DEFAULT DRIVE DOES NOT');
    writeln('HAVE ENOUGH SPACE FOR A SMALL FILE
                                         (10 TO 15 BLOCKS)');

```

```

    writeln('THEN TYPE LETTER N.');
```

```

    writeln;
    writeln('IF YOU ARE NOT SURE, HIT ANY OTHER KEY --');
```

```

    writeln('THERE IS PROBABLY ENOUGH SPACE.');
```

```

    break(output);
    lines := 9;
    clearlines;
    readln(ch);

```

```

if (ch = 'N')
then
begin
lines := 24;
clearlines;
writeln('Put a disk with some space on it
        in the default drive');
writeln('Then hit letter Y');
break(output);
readln(ch);
if (ch = 'Y')
then continue := true
else continue := false;
end
else continue := true;
clearlines
end;

```

```

(*****
*  DATE: 3 Aug 83
*  MODULE: Listinst
*  FUNCTION: Lists instructions on how
*             to query the database.
*  INPUTS: none.
*  OUTPUTS: none.
*  LOCAL VARIABLES: i, ch
*  GLOBALS USED: lines
*  MODULES CALLED: clearlines
*  CALLED BY: main
*  AUTHOR: VanKirk
*****)

```

```

procedure listinst;

```

```

var

```

```

    i : integer;
    ch : char;

```

```

begin

```

```

    writeln;
    writeln('To query a database,
             a legal format must be followed');
    break(output);
    writeln('The rules of the format follow:');
    writeln;
    writeln('
    1. A verb must be the first word in the query.');
```

- ```

 writeln('
 (Such as LIST, SHOW, GIVE, TYPE, PRINT) ');
 break(output);
 writeln;
 writeln('
 2. The rest of the query must be in sentence format');
 writeln('
 and use provided attribute names where possible.');
```
- ```

        writeln;
        break(output);
        writeln('
        3. Specific attribute values must be in quotes.');
```

 - ```

 writeln('
 Such as NAME = "Jones"');
```
- ```

        writeln;
        break(output);
        writeln('
        4. You must signal end of query by ending with a period');
```
- ```

 writeln;
 break(output);
 writeln('
 5. The query may not be more than 4 lines long (320');
 writeln('
 characters, including spaces and returns).');
```

```

 writeln;
 break(output);

```

```
writeln('Hit return to continue.');
```

lines := 4;

```
clearlines;
break(output);
read(ch); (Pause to let user read screen)

lines := 24;
clearlines;
writeln('A list of available attributes follows');
writeln;
break(output);

end;
```

```

(*****
* DATE: 29 Aug 83 *
* MODULE: Formdictionary *
* FUNCTION: Forms 2 linked lists from *
* files DATBAS.DIC and STD.DIC*
* to act as dictionaries. *
* INPUTS: none. *
* OUTPUTS: Linked lists with database and*
* stdbase as roots. *
* LOCAL VARIABLES: i, datfile, stdfile *
* GLOBALS USED: stdbase, database, dat- *
* temp, stdtemp *
* MODULES CALLED: clearlines *
* CALLED BY: Main *
* AUTHOR: VanKirk *
*****)

```

```

procedure formdictionary;

```

```

var

```

```

 i : integer;
 datfile, stdfile : text;

```

```

begin

```

```

 clearlines;
 writeln('CREATING DICTIONARIES');
 break(output);
 clearlines;
 new(database);
 dattemp := database;
 dattemp^.pointer := nil;

```

```

 (Get root)

```

```

 if (datdrive = 'Y')
 then reset(datfile, "DY0:DATBAS.DIC", 2)
 else reset(datfile, "DY1:DATBAS.DIC", 2);

```

```

 while not eof(datfile) do
 begin
 for i := 1 to 19 do
 read(datfile, word[i]);

```

```

 (Read from file)

```

```

 (For some unknown reason, read(datfile, dattemp^.identifier[i])
 does not work here -- when this module was inside PROCESSQRY,
 it worked; but at this level, for some reason, the file is
 found and opened, the characters can be read to a character
 holder (I read into ch), but assignment to
 dattemp^.identifier[i] resulted in no character transfer.
 Indirection through WORD works -- lucky to find it.)

```

```

 readln(datfile, word[20]);
 dattemp^.identifier := word;
 if eof(datfile)
 then dattemp^.pointer := nil
 else
 begin

```

```

 new(dattemp^.pointer);
 { Not done, get another box }
 dattemp := dattemp^.pointer
 end
end;

new(stdbase); { Same comments (and code) as above }
stdtemp := stdbase;
stdtemp^.pointer := nil;

if (stddrive = 'Y')
 then reset(stdfile, "DY0:STD.DIC", 2)
 else reset(stdfile, "DY1:STD.DIC", 2);

while not eof(stdfile) do
 begin
 for i := 1 to 19 do
 read(stdfile, word[i]);
 readln(stdfile, word[20]);
 stdtemp^.identifier := word;
 if eof(stdfile)
 then stdtemp^.pointer := nil
 else
 begin
 new(stdtemp^.pointer);
 stdtemp := stdtemp^.pointer
 end
 end
 end
 end;
end;

```

```

(*****
* DATE: 11 Aug 83 *
* MODULE: Getword *
* FUNCTION: Gets next word for trans- *
* lation out of array query. *
* INPUTS: none. *
* OUTPUTS: word *
* LOCAL VARIABLES: i, k *
* GLOBALS USED: j, length, query, word, *
* endquery *
* MODULES CALLED: none. *
* CALLED BY: Processqry, translate, *
* algebra, findquoted, find- *
* logical, findattribute *
* AUTHOR: VanKirk *
*****)

```

```

procedure getword;

```

```

var
 i, k : integer;

begin
 i := 1;
 while ((query[j] = ' ') and (j <> length)) do j := j + 1;
 while ((query[j] <> ' ') and (j < (length + 1))) do
 begin
 word[i] := query[j];
 i := i + 1;
 j := j + 1;
 end;
 if (j >= length) then endquery := true;
 for k := i to 20 do word[k] := ' ';
end;

```

```

(*****
* DATE: 16 Aug 83
* MODULE: Checkattribute
* FUNCTION: Looks for word in attribute
* dictionary.
* INPUTS: gotword.
* OUTPUTS: word, gotword
* LOCAL VARIABLES:none.
* GLOBALS USED: word, gotword, atrbase,
* atrptr
* MODULES CALLED:none.
* CALLED BY: Translate, algebra, find-
* attribute
* AUTHOR: VanKirk
*****)

```

```

procedure checkattribute(var gotword : boolean);

```

```

begin
 atrptr := atrbase;
 while ((atrptr <> nil) and (not gotword)) do
 (Go to attribute keyed list)
 if (atrptr^.atrname = word) (Go to correct attribute)
 then
 begin
 gotword := true;
 word := atrptr^.atrname
 (Found it!!)
 end
 else atrptr := atrptr^.pointeratr
 end;
end;

```

```

* DATE: 11 Aug 83 *
* MODULE: Processqry *
* FUNCTION: Oversees translation of *
* query to intermediate form. *
* INPUTS: none. *
* OUTPUTS: query. *
* LOCAL VARIABLES: none. *
* GLOBALS USED: query, j, nq, endquery, *
* calltranslate, newquery, *
* length *
* MODULES CALLED: getword, translate. *
* CALLED BY: Recvqry *
* AUTHOR: VanKirk *

```

```

procedure processqry!

```

```

(*****
* DATE: 11 Aug 83 *
* MODULE: Translate *
* FUNCTION: Checks query word-by-word *
* against dictionaries. *
* INPUTS: none. *
* OUTPUTS: word *
* LOCAL VARIABLES: i, gotword, thisfile, *
* gotquotes, saveit *
* GLOBALS USED: word, calltranslate, j *
* MODULES CALLED: checkword, check- *
* attribute, getword, *
* putword *
* CALLED BY: processqry *
* AUTHOR: VanKirk *
*****)

```

```

procedure translate;

```

```

var
 gotword, gotquotes : boolean;
 thisfile : text;
 i : integer;
 saveit : short;

```

```

(*****
* DATE: 11 Aug 83
* MODULE: Checkword
* FUNCTION: Looks for word in database
* and standard dictionaries.
* INPUTS: gotword.
* OUTPUTS: word, gotword.
* LOCAL VARIABLES: found, useptr
* GLOBALS USED: word
* MODULES CALLED: none.
* FILE READ: DATBAS.DIC, STD.DIC
* CALLED BY: translate
* AUTHOR: VanKirk
*****}

```

```

procedure checkword(var useptr : ^list; var found : boolean);
var

```

```

begin
 while (useptr <> nil) do
 begin
 if (useptr^.identifier = word)
 then
 begin
 useptr := useptr^.pointer;
 word := useptr^.identifier;
 found := true;
 useptr := nil;
 end
 else
 useptr := useptr^.pointer^.pointer
 { Skip next word }
 end
 end;
 end;
end;

```

```

* DATE: 29 Aug 83 *
* MODULE: Putword *
* FUNCTION: Puts word found in dic- *
* tionary in the new query. *
* INPUTS: none. *
* OUTPUTS: newquery. *
* LOCAL VARIABLES: i, spacecount *
* GLOBALS USED: newquery, nq, word *
* MODULES CALLED: none. *
* CALLED BY: Translate *
* AUTHOR: VanKirk *

```

```

procedure putword;

```

```

var

```

```

 i, spacecount : integer;

```

```

begin

```

```

 spacecount := 0;

```

```

 i := 1;

```

```

 while ((i < 21) and (spacecount <> 2)) do

```

```

 { Allow two words separated by one space for word }

```

```

 begin

```

```

 if ((word[i] = ' ') and (word[i + 1] = ' '))

```

```

 then spacecount := 2;

```

```

 newquery[nq] := word[i];

```

```

 { Put word in query }

```

```

 i := i + 1;

```

```

 if (nq < 320) then nq := nq + 1

```

```

 end

```

```

 end;

```

```

begin (***** TRANSLATE *****)
 calltranslate := calltranslate + 1;
 gotquotes := false; (Flag specific values)

 if (calltranslate = 1)
 then
 begin
 gotword := false;
 stdtemp := stdbase;
 checkword(stdtemp, gotword);
 if not gotword (Not a known word)
 then
 begin
 saveit := word; (Replace with LIST)
 word := 'LIST';
 putword;
 word := saveit;
 checkattribute(gotword)
 end
 end
 else
 begin
 if (word[1] = '"')
 (Words (up to 2) in quotes are specific values)
 then
 begin
 gotword := true;
 gotquotes := true
 end
 else gotword := false;
 if not gotword then checkattribute(gotword);
 (Is it attribute?)

 if not gotword
 then
 begin
 dattemp := database;
 checkword(dattemp, gotword)
 (Is it attribute alias?)
 end;
 if not gotword
 then
 begin
 stdtemp := stdbase;
 (Look in standard dictionary)
 checkword(stdtemp, gotword)
 end
 end;
 end
 if gotword
 then
 begin
 putword;
 if gotquotes
 then
 begin

```

```

i := j + 1;
while (query[i] <> ' ') do
 (See if only one word in quotes)
begin
 if query[i] = '"'
 then gotquotes := true
 else gotquotes := false;
 if gotquotes
 then
 begin
 getword;
 putword
 end;
 i := i + 1
 end
end
end
end;

```

```

begin (***** PROCESSORY *****)
 for nq := 1 to 320 do newquery[nq] := trash;
 (Fill with unlikely char)
 endquery := false;
 j := 1; (Counter for query array)
 nq := 1; (Counter for newquery array)
 calltranslate := 0;
 (Counter-type flag set to look for verb)
 while not endquery do
 begin
 getword;
 translate
 end;
 nq := nq - 1; (nq points past end of query)
 if (newquery[nq] = ' ') then nq := nq - 1;
 (If last character of newquery blank,
 no sense in keeping it)
 query := newquery;
 (Save translated query in global area)
 length := nq
 end;

```

```

* DATE: 3 Aug 83 *
* MODULE: Recvqry *
* FUNCTION: Lets user insert query. *
* INPUTS: i - start marker to array. *
* OUTPUTS: array query *
* LOCAL VARIABLES: ch, k *
* GLOBALS USED: query, length, lines *
* MODULES CALLED: changeqry, clearlines,*
* processqry *
* CALLED BY: main, changeqry *
* AUTHOR: VanKirk *

```

```

procedure recvqry(var i : integer);
 { Counter for where to start in in array query }
var
 k : integer; { Counter for query length }
 ascii : integer; { Decimal equivalent of ascii }
 ch : char;
begin
 if (i = 1)
 then
 begin
 writeln;
 writeln('Input query (end with .):');
 break(output)
 end;
 k := i;
 while (i < 321) do { Receive the query }
 begin
 read(ch);
 ascii := ord(ch);

 if ((ascii < 33) or (ascii > 90)) then ascii := 36;
 { Unwanted characters }
 if (((ascii > 47) and (ascii < 58)) or ((ascii > 64) and
 (ascii < 91))) then ascii := 65;
 { Numbers and capital letters }

 case ascii of
 33, 34, 35, 38, 42, 43, 45, 46, 47, 60, 61, 62, 65: ;
 { Save numbers, letters, operators, and special symbols }
 36, 37, 39, 40, 41, 44, 58, 59, 63, 64: ch := ' ';
 { Toss out unwanted characters }
 end;

 if (ch = ender)
 { Stop at . since <CR> can happen many times }
 then
 begin
 i := 350;
 readln(ch)
 end;
 end;
end;

```

```

 if (i < 321) then query[i] := ch;
 { Save only 320 characters }
 k := k + 1;
 { Count characters }
 i := i + 1
 end;

k := k - 2;
length := k;
 { k points one spot past ender }
 { length points at last letter }

processqry;
 { Take query to intermediate form }
writeln('This is the query as understood:');
writeln;
break(output);
for i := 1 to length do write(query[i]);
 { Ignore screen end while outputting query }
break(output);
writeln;
writeln;
break(output);
write('Is it correct? Answer Y or N:');
break(output);
readln(ch);
if (ch = 'N')
 then changeqry
 else
 begin
 lines := 24;
 clearlines;
 writeln('PROCESSING YOUR QUERY');
 break(output);
 lines := 12;
 clearlines
 end
end;
end;

```

```

* DATE: 3 Aug 83 *
* MODULE: Changeqry *
* FUNCTION: Allows user to edit the *
* query. *
* INPUTS: none. *
* OUTPUTS: query *
* LOCAL VARIABLES: find, i, j, ch, done, *
* count *
* GLOBALS USED: query, j *
* MODULES CALLED: recvqry *
* CALLED BY: recvqry *
* AUTHOR: VanKirk *

```

```

procedure changeqry;

```

```

var

```

```

 find : packed array[1..80] of char;
 (String indicating where change starts)
 i, j, count : integer;
 (i steps thru find, j thru query count
 has number characters in find)

 ch : char;
 done : boolean;

```

```

begin

```

```

 writeln;
 writeln('Input a string of characters uniquely identifying');
 break(output);
 writeln('the part of the query you want to change. Maximum');
 writeln('length of the string is one line (80 characters).');
 writeln;
 break(output);
 writeln('THE QUERY WILL BE OVERLAID STARTING AT THE FIRST');
 writeln('CHARACTER YOU INPUT.');
```

```

 writeln;
 break(output);
 writeln('Input string (end with <CR>):');
 writeln;
 break(output);

```

```

 for i := 1 to 80 do find[i] := trash;
 (Fill with unlikely character)
 read(ch);
 i := 1;

```

```

 while (ord(ch) <> 10) do (Stop reading on <CR>)
 begin
 if (i < 81) then find[i] := ch;
 i := i + 1;
 read(ch)
 end;

```

```

 count := 0;

```

```

i := 1;

while ((find[i] <> trash) and (i < 81)) do
begin
 count := count + 1;
 (Count number of characters in find)
 i := i + 1
end;

j := 1;
i := 1;
done := false;

while not done do
begin
 if (find[i] = query[j])
 (Locate string in query held by find)
 then i := i + 1
 else i := 1;

 j := j + 1;

 if ((i - 1) = count) then done := true
 (Found string)
end;

j := j - count;
 (j points at first character in find string)
writeln;
writeln('Input changes (end with .):');
writeln;
break(output);
recvqry(j)
end;

```

```

* DATE: 31 Aug 83 *
* MODULE: Algebra *
* FUNCTION: Transforms intermediate form*
* of query to relational *
* algebra. *
* INPUTS: none. *
* OUTPUTS: See FILES WRITTEN. *
* LOCAL VARIABLES: attribute, done, *
* doubleparen, quoted, *
* lastfile, relation, *
* logical, operator, *
* opholder *
* GLOBALS USED: word, j, relbase, relptr*
* MODULES CALLED: Findquoted, select, *
* getword, join, check- *
* attribute, project, *
* allused *
* FILES WRITTEN: QRYTMP.TMP *
* CALLED BY: Main *
* AUTHOR: VanKirk *

```

procedure algebra;

```

var
 attribute, done, doubleparen,
 logical, operator, quoted : boolean;
 (Flags to track format of algebra)
 lastfile : text;
 (Temporary file created until user makes final
 decision on whether to keep query)
 relation : packed array[1..3] of char;
 (Made up relation names for join, select, and
 project operations)
 opholder : char;
 (Holds one character relational algebra
 operator)

```

```

* DATE: 2 Sep 83 *
* MODULE: Newrel *
* FUNCTION: Changes variable RELATION *
* to make new relation name. *
* INPUTS: none. *
* OUTPUTS: relation[2] changes. *
* LOCAL VARIABLES: none. *
* GLOBALS USED: relation *
* MODULES CALLED: none. *
* CALLED BY: Join, select *
* AUTHOR: VanKirk *
*****}

```

```

procedure newrel;
begin
 relation[2] := succ(relation[2])
end;

```

```

(*****
* DATE: 5 Sep 83
* MODULE: Putfile
* FUNCTION: Writes single word to file
* GRYTMP.TMP by way of last-
* file.
* INPUTS: Nameit.
* OUTPUTS: See FILE WRITTEN.
* LOCAL VARIABLES: i, nameit.
* GLOBALS USED: lastfile
* MODULES CALLED: none.
* FILE WRITTEN: GRYTMP.TMP
* CALLED BY: Dowhere, project, join
* AUTHOR: VanKirk
*****)

procedure putfile(nameit : short);

var
 i : integer;

begin
 i := 1;
 while ((i < 21) and (nameit[i] <> ' ')) do
 begin
 write(lastfile, nameit[i]);
 i := i + 1
 end
 end;
end;

```

```

(*****
* DATE: 5 Sep 83
* MODULE: Allused
* FUNCTION: See if all relations have
* been used in JOIN.
* INPUTS: none.
* OUTPUTS: boolean.
* LOCAL VARIABLES: notused
* GLOBALS USED: relbase
* MODULES CALLED: none
* CALLED BY: Join
* AUTHOR: VanKirk
*****)

```

```
function allused : boolean;
```

```

var
 notused : ^listrel; (Pointer to relation list)

begin
 notused := relbase; (Start at top of list)
 while ((notused <> nil) and (notused^.used = 'Y'))
 do notused := notused^.relpointer;
 if (notused = nil)
 then allused := true (All relations used)
 else allused := false
end;

```

```

(*****
* DATE: 5 Sep 83
* MODULE: Join
* FUNCTION: Creates relational algebra
* JOIN.
* INPUTS: latest - Last relation created
* OUTPUTS: See FILE WRITTEN
* LOCAL VARIABLES: current, gorel, match
* GLOBALS USED: lastfile, word, atrtemp,
* reltemp
* MODULES CALLED: Getptr, attlist, new-
* rel, addto, putfile
* FILE WRITTEN: QRYTMP.TMP
* CALLED BY: Algebra
* AUTHOR: VanKirk
*****}

```

```

procedure join(var latest : short);

```

```

var

```

```

 gorel,

```

```

 current : ^listrel;
 match : boolean;

```

```

 { Pointer to unused relation found
 with same attribute in it }
 { Pointer to result of last join }
 { Tells if found a relation that is
 not yet used in join AND has an
 attribute in common with CURRENT }

```

```

* DATE: 5 Sep 83 *
* MODULE: Getptr *
* FUNCTION: Gets a pointer in relation *
* list that points to relation*
* named. *
* INPUTS: Findrel, return *
* OUTPUTS: Return *
* LOCAL VARIABLES: none. *
* GLOBALS USED: relbase *
* MODULES CALLED: none. *
* CALLED BY: Join *
* AUTHOR: VanKirk *
*****}

```

```

procedure getptr(findrel : short; var return : ^listrel);

```

```

begin
 return := relbase;
 while (return^.relname <> findrel)
 do return := return^.relpointer
end;

```

```

* DATE: 5 Sep 83 *
* MODULE: Attlist *
* FUNCTION: Looks thru attribute list *
* for attribute SENT, then *
* returns pointer to a *
* relation that has SENT *
* attribute in it. If that *
* attribute can't be found, *
* moves pointer SENT to next *
* attribute, and looks for it.* *
* INPUTS: none. *
* OUTPUTS: thatptr *
* LOCAL VARIABLES: thisptr *
* GLOBALS USED: atrbase, atrtemp, rel- *
* temp *
* MODULES CALLED: none. *
* CALLED BY: Join *
* AUTHOR: VanKirk *
*****}

```

```

procedure attlist;

```

```

var
 thisptr : ^listatr; (Pointer to attribute list)

begin
 if (reltemp = nil)
 (Nil indicates haven't found common attribute yet)
 then
 begin
 thisptr := atrbase; (Start at top)
 while ((atrtemp^.identifier <> thisptr^.atrname)
 and (thisptr <> nil))
 do thisptr := thisptr^.pointeratr;
 (Try next attribute in attribute list)
 if (thisptr = nil)
 then (Look for next attribute from list)
 begin
 atrtemp := atrtemp^.pointer;
 attlist
 end
 else reltemp := thisptr^.pointerrel
 (Found a relation with this attribute)
 end
 else reltemp := reltemp^.pointer;
 (Last relation name sent back was already used)
 if (reltemp = nil)
 then (No more relations have this attribute)
 begin
 atrtemp := atrtemp^.pointer;
 attlist
 end
end;

```

```

(*****
* DATE: 5 Sep 83
* MODULE: Addto
* FUNCTION: Adds relation named to list
* of relations, and makes it
* "used" for JOIN purposes.
* INPUTS: Rel1, rel2
* OUTPUTS: none.
* LOCAL VARIABLES: add, addrel, puthere,
* addit
* GLOBALS USED: relbase, relation
* MODULES CALLED: getptr
* CALLED BY: Join
* AUTHOR: VanKirk
*****)

```

```

procedure addto(rel1, rel2 : short);
 (Rel1 and rel2 are relations joined to
 produce current relation name created)
var
 add, (Pointer to rel1/rel2 name in relation list)
 addrel : (Pointer to end of relation list, where
 relation is added)
 ^listrel;

 puthere, (Pointer to attribute list of addrel
 where attributes are being added)
 addit : (Pointer to rel1/rel2 attribute)
 ^list;

begin
 getptr(rel1, add); (Get pointer to first relation)

 addrel := relbase;
 while (addrel^.relpointer <> nil)
 do addrel := addrel^.relpointer; (Move to end of list)
 new(addrel^.relpointer); (Create a new holder)
 addrel := addrel^.relpointer; (Ready to add info)

 addrel^.relname := space; (Put in relation name)
 addrel^.relname[1] := relation[1];
 addrel^.relname[2] := relation[2];
 addrel^.relname[3] := relation[3];

 addrel^.relpointer := nil; (Put new end on list)

 new(addrel^.atrpointer);
 (Open up new relation's attribute list)

 puthere := addrel^.atrpointer; (Set PUTHERE)

 addit := add^.atrpointer; (Pointer to rel1 attributes)
 puthere^.identifier := addit^.identifier;
 (Put first attribute on list)
 addit := addit^.pointer; (Get next attribute)

```

```

while (addit <> nil) do
 { Put all of rel1 attributes on list }
begin
 new(puthere^.pointer);
 { Get ready for next attribute }
 puthere := puthere^.pointer;
 puthere^.identifier := addit^.identifier;
 addit := addit^.pointer { Get next attribute }
end;

puthere^.pointer := nil;
{ Temporarily end attribute list }
getptr(rel2, add); { Rel1 done, get pointer to rel2 }
addit := add^.atrpointer;

while (addit <> nil) do
begin
 puthere := addrel^.atrpointer;
 { Check attributes already added for duplication }
 while ((puthere^.identifier <> addit^.identifier)
 and (puthere^.pointer <> nil)) do
 puthere := puthere^.pointer;

 if (puthere^.identifier <> addit^.identifier)
 then
 { Didn't have it, add it }
 begin
 new(puthere^.pointer);
 puthere := puthere^.pointer;
 puthere^.pointer := nil; { End the list }
 puthere^.identifier := addit^.identifier
 end;

 addit := addit^.pointer
 end;

 addrel^.used := 'Y'
end;
{ Make relation "used" }

```

```

begin (***** JOIN *****)
 getptr(latest, current);
 (Find CURRENT relation in relation list)
 atrtemp := current^.atrpointer;
 (Set now to attributes)
 current^.used := 'Y'; (Current is "used" in join)
 match := false;
 reltemp := nil; (Set RELTEMP for ATTLIST call)

 while not match do
 begin
 attlist;
 (Find pointer to relation with same attribute)
 getptr(reltemp^.identifier, gorel);
 (See if this relation is used)
 if (gorel^.used = 'N')
 then match := true (Wasn't used)
 end;

 gorel^.used := 'Y'; (Used now!)

 write(lastfile, 'JOIN ');
 (Write relational algebra to QRYTMP.TMP)
 putfile(current^.relname); (First relation in join)
 write(lastfile, ', ');
 putfile(gorel^.relname); (Second relation in join)
 write(lastfile, ' WHERE ');
 putfile(atrtemp^.identifier); (The attribute in common)
 write(lastfile, ' = ');
 putfile(atrtemp^.identifier);
 newrel; (Create a name for the resultant relation)
 writeln(lastfile, ' GIVING ', relation);
 latest := space;
 (Reset LATEST to created relation name)
 latest[1] := relation[1];
 latest[2] := relation[2];
 latest[3] := relation[3];

 addto(current^.relname, gorel^.relname);
 (Add latest to relation list)

 break(lastfile)
 end;
end;

```

```

* DATE: 31 Aug 83 *
* MODULE: Seterror *
* FUNCTION: Set flags for quick exit of *
* program, and give initial *
* error message. *
* INPUTS: none. *
* OUTPUTS: See GLOBALS USED *
* LOCAL VARIABLES: none. *
* GLOBALS USED: attribute, stopprocess, *
* quoted, logical, double- *
* paren, done, lines *
* MODULES CALLED: clearlines *
* CALLED BY: findlogical, findoperator, *
* findattribute *
* AUTHOR: VanKirk *
*****}

```

```

procedure seterror;

```

```

begin
 lines := 24;
 clearlines;
 attribute := false;
 quoted := false;
 logical := false;
 doubleparen := false;
 done := true;
 stopprocess := true;
 writeln;
 writeln('UNABLE TO PROCESS YOUR QUERY DUE TO ERROR');
 break(output)
end;

```

```

{*****}
* DATE: 10 Sep 83 *
* MODULE: Findoperator *
* FUNCTION: Finds operator indicating *
* need to create SELECT. *
* INPUTS: none. *
* OUTPUTS: findoperator *
* LOCAL VARIABLES: i *
* GLOBALS USED: query, j, length, *
* opholder *
* MODULES CALLED: seterror *
* CALLED BY: Algebra, select *
* AUTHOR: VanKirk *
{*****}

```

```
function findoperator : boolean;
```

```
var
```

```
 i : integer;
```

```
begin
```

```
 i := j; { Where to start looking }
 while ((i <> length) and (not((query[i] = '=')
 or (query[i] = '>') or (query[i] = '<')))) do
 i := i + 1; { Find the next operator }
```

```
 if ((query[i] = '=') or (query[i] = '<')
 or (query[i] = '>'))
```

```
 then
```

```
 begin
```

```
 j := i; { Put j on the operator }
```

```
 findoperator := true;
```

```
 opholder := query[j]; { Save the operator }
```

```
 if ((query[i+1] = '=') or (query[i+1] = '<')
 or (query[i+1] = '>'))
```

```
 then
```

```
 begin
```

```
 seterror;
```

```
 writeln(query[i], query[i+1],
 ' is an illegal operator.');
```

```
 writeln('The Roth system only allows
 <, >, or =');
```

```
 writeln;
```

```
 writeln('Hit RETURN to continue');
```

```
 break(output);
```

```
 readln(ch)
```

```
 end
```

```
 end
```

```
 else findoperator := false
```

```
 end;
```

```

(*****
* DATE: 2 Sep 83 *
* MODULE: Select *
* FUNCTION: Creates relational algebra *
* SELECT. *
* INPUTS: none. *
* OUTPUTS: See FILE WRITTEN *
* LOCAL VARIABLES: logholder, value, *
* twowords *
* GLOBALS USED: doubleparen, quoted, *
* lastfile, logical, *
* operator, ch *
* MODULES CALLED: Findlogical, find- *
* quoted, dowhere, *
* findoperator, seterror*
* FILE WRITTEN: QRYTMP.TMP by way of *
* lastfile *
* CALLED BY: Algebra *
* AUTHOR: VanKirk *
*****)

```

```

procedure select;

```

```

var

```

```

 logholder : packed array[1..3] of char;
 (Holds logical operator between this quoted
 value and next attribute)
 value : short;
 (Holds specified value or attribute following
 an operator)
 twowords : boolean;
 (True if quoted value consists of two words)

```

```

* DATE: 10 Sep 83 *
* MODULE: Findquoted *
* FUNCTION: Finds specific quoted value *
* or attribute following an *
* operator. *
* INPUTS: none. *
* OUTPUTS: value *
* LOCAL VARIABLES: i, l, goodatt *
* GLOBALS USED: j, ch, query, value, *
* twowords *
* MODULES CALLED: getword, check- *
* attribute, seterror *
* CALLED BY: Select *
* AUTHOR: VanKirk *

```

```
function findquoted : boolean;
```

```
var
```

```

 i, l : integer;
 goodatt : boolean; (Indicates if word is an attribute)

```

```
begin
```

```

 twowords := false; (No words in quotes yet)
 value := space; (Clear value out)

```

```
 findquoted := false;
```

```
 i := j + 2;
```

```
 (Set i to character past blank following operator)
```

```
 if (query[i] = '"')
```

```
 then
```

```
 begin
```

```
 (Quoted value)
```

```
 i := i + 1;
```

```
 (Get past quote)
```

```
 l := i;
```

```
 (Pointer for value)
```

```
 while (not ((query[i] = ' ') or (query[i] = '"'))) do
```

```
 begin
```

```
 if ((l < 21) and (i < (length + 1)))
```

```
 then (Save value)
```

```
 begin
```

```
 value[l] := query[i];
```

```
 l := l + 1
```

```
 end;
```

```
 i := i + 1
```

```
 end;
```

```
 if (query[i] = ' ')
```

```
 then
```

```
 (Two words in quote)
```

```
 begin
```

```
 twowords := true;
```

```
 value[l] := query[i];
```

```
 l := l + 1;
```

```
 i := i + 1;
```

```
 while (not ((query[i] = ' '))
```

```

 or (query[i] = '"')) do
begin
 if ((i < 21) and (query[i] <> '"')
 and (i < (length + 1)))
 then
 begin
 value[i] := query[i];
 l := l + 1
 end;
 i := i + 1
 end
end;

if ((i > length) or ((query[i] <> '"')
 and (i < (length + 1))))
then
begin
 seterror;
 write('Operator ', query[j]);
 j := j + 1;
 getword;
 writeln(' is followed by ', word);
 writeln('Which has no ending quote');
 writeln;
 writeln('Hit RETURN to continue');
 break(output);
 readln(ch)
end
else findquoted := true
end

else
begin
 i := j; { Save j }
 j := j + 1; { Step into blank after operator }
 getword;
 j := i; { Reset j }
 value := word; { Save value }
 goodatt := false;
 checkattribute(goodatt); { See if it is an attribute }
 if not goodatt
 then
 begin
 seterror;
 writeln('Operator ', opholder,
 ' is followed by a');
 writeln('value that is neither an
 attribute, nor quoted');
 writeln;
 writeln('Hit RETURN to continue');
 break(output);
 readln(ch)
 end
 else findquoted := true
end

```

end;  
end

```

(*****
* DATE: 10 Sep 83
* MODULE: Findlogical
* FUNCTION: Looks for a logical operator*
* following quoted material.
* INPUTS: none.
* OUTPUTS: boolean
* LOCAL VARIABLES: i
* GLOBALS USED: j, word, logholder,
* twowords
* MODULES CALLED: getword
* CALLED BY: Select
* AUTHOR: VanKirk
*****)

```

```
function findlogical : boolean;
```

```
var
 i : integer;
```

```

begin
 i := j; (j points to operator)
 j := j + 1; (j saved in i, move to next word)
 getword; (The attribute or specified value)
 getword;
 if twowords (If last value was two words long)
 then getword; (the logical is one word farther)
 if (((word[1] = 'A') and (word[2] = 'N') and
 (word[3] = 'D') and (word[4] = ' '))
 or
 ((word[1] = 'O') and (word[2] = 'R')
 and (word[3] = ' ')))
 then (Found a logical)
 begin
 findlogical := true;
 logholder[1] := word[1];
 logholder[2] := word[2];
 logholder[3] := word[3];
 end;
 j := i (Reset j)
end;

```

```

(*****
* DATE: 2 Sep 83
* MODULE: Dowhere
* FUNCTION: Creates WHERE part of
* relational algebra SELECT.
* INPUTS: none.
* OUTPUTS: See FILE WRITTEN and GLOBALS
* USED.
* LOCAL VARIABLES: operator, opholder,
* attholder
* GLOBALS USED: query, j, attholder,
* opholder, operator,
* attribute, lastfile,
* twowords, quoted
* MODULES CALLED: findattribute
* FILE WRITTEN: GRYTMP.TMP, by way of
* lastfile
* CALLED BY: Select
* AUTHOR: VanKirk
*****}

```

```

procedure dowhere;

```

```

var
 i : integer;
 attholder : short;
 (Holds attribute associated with current
 opholder and quoted value)

```

```

(*****
* DATE: 1 Sep 83
* MODULE: Findattribute
* FUNCTION: Looks forward of operator
* in search of attribute to
* do SELECT on.
* INPUTS: none.
* OUTPUTS: boolean, attholder
* LOCAL VARIABLES: find, k
* GLOBALS USED: query, j, attholder,
* word, length, ch,
* value
* MODULES CALLED: checkattribute,
* getword, seterror
* CALLED BY: Dowhere
* AUTHOR: VanKirk
*****}

```

```
function findattribute : boolean;
```

```
var
```

```
 find : boolean;
```

```
 { Local holder to use in calling checkattribute }
```

```
 k : integer;
```

```
begin
```

```
 find := false;
```

```
 findattribute := false;
```

```
 k := j - 2; { Skip forward of operator, past blank }
```

```
 while ((query[k] <> ' ') and (k > 1))
```

```
 do k := k - 1; { Find beginning of word }
```

```
 i := j; { Save j in i since getword changes j }
```

```
 j := k; { Position j to word wanted }
```

```
 getword; { Global WORD now has word }
```

```
 checkattribute(find);
```

```
 { Check global WORD against attributes }
```

```
 j := i; { Restore j to operator }
```

```
 if find
```

```
 then { WORD has attribute in it }
```

```
 begin
```

```
 attholder := word;
```

```
 findattribute := true
```

```
 end
```

```
 else
```

```
 begin
```

```
 seterror;
```

```
 writeln('Operator ', opholder,
```

```
 ' is not preceded by an attribute');
```

```
 writeln;
```

```
 writeln('Hit RETURN to continue');
```

```
 break(output);
```

```
 readln(ch)
```

```
 end
```

```
end;
```

```

begin (***** DOWHERE *****)
 if quoted
 then attribute := findattribute;
 if attribute
 then
 begin
 putfile(attholder); { Write attribute }
 write(lastfile, ' ');
 write(lastfile, opholder, ' '); { Write operator }
 i := 1;
 while ((i < 21) and (value[i] <> ' ')) do
 begin
 write(lastfile, value[i]); { Write value }
 i := i + 1;
 end;
 if twowords
 then
 begin
 write(lastfile, value[i]); { Write space }
 i := i + 1;
 while ((i < 21) and (value[i] <> ' ')) do
 begin
 write(lastfile, value[i]);
 i := i + 1;
 end
 end;
 break(lastfile)
 end
 end;
 end;
end;

```

```

begin (***** SELECT *****)
 write(lastfile, 'SELECT ALL FROM ', relation, ' WHERE ');
 break(lastfile);
 quoted := findquoted;
 if quoted
 then logical := findlogical;
 if logical
 then (Need double parenthesis)
 begin
 write(lastfile, '(');
 break(lastfile);
 doubleparen := true
 end
 else doubleparen := false;
 if doubleparen
 then (As long as there are logicals,
 put in selection criteria)
 begin
 write(lastfile, '(');
 dowhere;
 write(lastfile, ') ');
 write(lastfile, logholder);
 if (logholder[3] <> ' ')
 then write(lastfile, ' ');
 while logical do
 begin
 j := j + 1; (Put j past last operator)
 operator := findoperator;
 if not operator
 then (Logical has no following constraint)
 begin
 seterror;
 j := 500;
 (Make j too large for further processing)
 writeln('Logical ', logholder,
 ' has invalid constraint');
 write('following it. The constraint has
 no operator ');
 writeln('<, >, =' in it. ');
 writeln;
 writeln('Hit RETURN to continue');
 break(output);
 readln(ch)
 end;
 if (j < 500)
 then quoted := findquoted
 else quoted := false;
 write(lastfile, '(');
 dowhere;
 write(lastfile, ') ');
 if quoted
 then logical := findlogical;
 if logical
 then
 begin

```

```

 write(lastfile, ' ');
 write(lastfile, logholder);
 if (logholder[3] <> ' ')
 then write(lastfile, ' ')
 end
 end;
 write(lastfile, ') ');
 { End list of constraints on attributes }
end

else dowhere; { No logicals, only one constraint }

newrel; { Get new relation name }
writeln(lastfile, ' GIVING ', relation);
break(lastfile)
end;

```

```

(*****
* DATE: 2 Sept 83
* MODULE: Project
* FUNCTION: Produces relational algebra
* PROJECT.
* INPUTS: none.
* OUTPUTS: See FILE WRITTEN
* LOCAL VARIABLES: i, more
* GLOBALS USED: lastfile, word
* MODULES CALLED: getword, check-
* attribute, putfile
* FILE WRITTEN: QRYTMP.TMP by way of
* lastfile
* CALLED BY: Algebra
* AUTHOR: VanKirk
*****}

```

```

procedure project;

```

```

var

```

```

 i : integer;
 more : boolean;

```

```

begin

```

```

 more := false;
 write(lastfile, 'PROJECT ', relation, ' OVER ');
 putfile(word); { Put attribute in relational algebra }
 if (word <> 'ALL
 then
 begin
 getword; { Get next word in query }
 checkattribute(more) { See if it is an attribute }
 end;
 while more do
 begin
 { Add on attributes until no more }
 write(lastfile, ', ');
 putfile(word);
 more := false;
 getword;
 if (word = 'AND
 then getword; { Skip conjunction in list }
 checkattribute(more)
 end;
 writeln(lastfile, ' GIVING NATANS');
 break(lastfile)
end;

```

```

begin { ***** ALGEBRA ***** }
 relation := 'Z0Z';
 { RELATION [2] gets character preceeding capital A }
 rewrite(lastfile, "QRYTMP.TMP");
 { File to store relational algebra in }
 relptr := relbase;

 while (relptr <> nil) do
 begin
 { Initialize relation linked list to
 show not yet used in join }
 relptr^.used := 'N';
 relptr := relptr^.relpointer
 end;

 relptr := relbase;
 word := relptr^.relname; { Got first relation -- hope the
 rest is this easy }
 done := false;

 while not done do
 begin
 join(word);
 done := allused { See if all relations used in join }
 end;

 j := 1; { Start at beginning of query }
 done := false;

 operator := findoperator;
 { See if user gave selection criteria }
 if operator
 then select; { Make relational algebra selections }

 if not done { No errors found }
 then
 begin
 j := 1;
 getword; { Skip "LIST" }
 getword;
 attribute := false;
 checkattribute(attribute);
 if attribute { If user did not specify which attribute
 then no PROJECT is needed }
 then project
 end
 end;
end;

```

```

* DATE: 24 Aug 83 *
* MODULE: Saveqry *
* FUNCTION: Shows Relational Algebra *
* form of query and gives *
* user option of saving as a *
* command file for the Roth *
* system. If saved, a file- *
* name is specified. *
* INPUTS: User responses. *
* OUTPUTS: Disk file. *
* LOCAL VARIABLES: i, commandfile, file- *
* name, ch *
* GLOBALS USED: query, length, lines *
* MODULES CALLED: clearlines *
* CALLED BY: main *
* AUTHOR: VanKirk *

```

```

procedure saveqry;

```

```

var
 i : integer;
 commandfile, queryfile : text;
 filename : packed array[1..15] of char;
 { User specified disk filename }
 ch : char;

begin
 reset(queryfile, "QRYTMP.TMP");
 writeln('The Relational Algebra form of the query is:');
 writeln;
 while not eof(queryfile) do
 begin
 read(queryfile, ch);
 write(ch)
 end;
 break(output);
 lines := 7;
 clearlines;
 writeln('Would you like to save it as a command file?
 (Y or N)');
 break(output);
 readln(ch);
 if (ch <> 'N') { Spring-loaded to save query }
 then
 begin
 writeln;
 writeln('Input disk filename to save query in. ');
 writeln('If a disk drive is not specified,
 the default');
 write('drive will be used: ');
 break(output);
 i := 1;
 read(ch);
 end;

```

```

while ((i < 15) and (ord(ch) <> 10)) do
begin
 filename[i] := ch;
 i := i + 1;
 read(ch)
end;
filename[i] := chr(0);
rewrite(commandfile, filename);
reset(queryfile, "QRYTMP.TMP");
while not eof(queryfile) do
begin
 read(queryfile, ch);
 write(commandfile, ch)
end;
break(commandfile)
end;
rewrite(queryfile, "QRYTMP.TMP")
end;

```

```

(***** MAIN PROGRAM *****)
begin
 lines := 24; { Clear screen }
 clearlines;
 stopprocess := false;
 continue := false;
 while not continue
 do requestid;
 while continue do
 { Once in program, can switch disks to keep
 changing databases to build queries against }
 begin
 getdrives; { Find out where dictionaries are }
 fordictionary; { Make the dictionaries from files }
 lines := 24;
 clearlines; { Clear screen }
 listinst; { Give instructions on how to query }
 getattributes; { Show user attributes from database }
 if not stopprocess
 then
 begin
 j := 1; { Set up new query string }
 recvqry(j); { Recieve query and change to
 intermediate form }
 algebra; { Translate into Relational Algebra }
 if not stopprocess
 then saveqry;
 end;
 ch := ' ';
 while ((ch <> 'Y') and (ch <> 'N')) do
 begin
 lines := 24;
 clearlines;
 writeln('Would you like to try another query,');
 write('possibly on another database? (Y or N): ');
 break(output);
 readln(ch);
 lines := 24;
 stopprocess := false;
 if (ch = 'N')
 then continue := false;
 clearlines;
 end
 end
 end.

```

# Appendix I

## MAKDIC LISTING

```

{*****}
* DATE: 18 Aug 83 *
* MODULE: Makedictionary *
* FUNCTION: Creates/Adds to files *
* STD.DIC or DATBAS.DIC. *
* These files are used by *
* NATQRY in query translation.*
* STD.DIC holds operators and *
* "universal" keywords, while *
* DATBAS.DIC is unique to a *
* particular database. *
* INPUTS: Requested interactively. *
* OUTPUTS: Files STD.DIC, DATBAS.DIC *
* LOCAL VARIABLES: circle, ch, func, *
* name, word, i *
* MODULES CALLED: none. *
* FILE READ: STD.DIC (if required). *
* CALLED BY: n/a (incorporate into the *
* ROTH DATABASE SYSTEM as a *
* module at a later date.) *
* AUTHOR: VanKirk *
{*****}

```

```

program makedictionary;

```

```

type

```

```

 list = record
 ident : array[1..20] of char;
 pointer : ^list
 end;

```

```

var

```

```

 circle : boolean; { Loop checker to force correct inputs }
 i : integer; { Counter to step through array word }
 ch, { Utility character holder }
 func, { Determines whether to add to existing
 dictionary or create new dictionary }
 name : char; { Determines whether to work on STD.DIC
 or DATBAS.DIC }
 dictionary : text; { Name in this program for file written }

 word : array[1..20] of char; { The word input to file }

 base, ptr : ^list; { Pointers for manipulating dictionary
 during add information operation }

```

```

begin

```

```

 circle := true;

```

```

 while circle do { Get function requested }

```

```

begin
 writeln('Add to or Create a dictionary? (A or C)');
 break(output); { Force out writeln data }
 readln(func);
 if ((func = 'A') or (func = 'C')) then circle := false
end;

if (func = 'A')
 then writeln('Make sure proper file is on default drive.')
 else writeln('File will be put on default drive.');
```

writeln;

break(output);

circle := true;

while circle do { Get file requested }

```

begin
 if (func = 'A')
 then writeln('Add to Standard or Database dictionary?
 (S or D)')
 else writeln('Create Standard or Database dictionary?
 (S or D)');
 break(output);

 readln(name);

 if ((name = 'S') or (name = 'D')) then circle := false
end;
```

if (func = 'C') { Create a dictionary }

```

then
begin
 if (name = 'S')
 then rewrite(dictionary, "STD.DIC", 2);
 if (name = 'D')
 then rewrite(dictionary, "DATBAS.DIC", 2)
end;
```

if (func = 'A') { Add to existing dictionary }

```

then
begin
 if (name = 'S')
 then reset(dictionary, "STD.DIC", 2);
 if (name = 'D')
 then reset(dictionary, "DATBAS.DIC", 2);

 new(base); { Read dictionary to save it }
 ptr := base;
 ptr^.pointer := nil;
 while not eof(dictionary) do
 begin
 for i := 1 to 19 do
 begin
 if not eof(dictionary)
 then read(dictionary, ch);
 end;
 end;
```

```

 ptr^.ident[i] := ch
 end;

 if not eof(dictionary) { Read end of line }
 then readln(dictionary, ch);
 ptr^.ident[20] := ch;

 if not eof(dictionary)
 then
 begin
 { Add to list }
 new(ptr^.pointer);
 ptr := ptr^.pointer;
 ptr^.pointer := nil
 end
 end;
end; { Done reading dictionary }

if (name = 'S')
then rewrite(dictionary, "STD.DIC", 2)
else rewrite(dictionary, "DATBAS.DIC", 2);

ptr := base; { Put info back in dictionary }
while (ptr <> nil) do
begin
 writeln(dictionary, ptr^.ident);
 break(dictionary);
 ptr := ptr^.pointer
end
end;

circle := true;

while circle do
begin
 writeln('Input word to define');
 break(output);

 i := 1;

 read(ch); { Get word }

 while (ord(ch) <> 10) do { Stop on <CR> }
 begin
 word[i] := ch;
 read(ch);
 i := i + 1
 end;

 while (i < 21) do { Pack with blanks }
 begin
 word[i] := ' ';
 i := i + 1
 end;

 writeln(dictionary, word); { Put word in dictionary }

```

```

writeln('Input alias (replacement) for input word');
break(output);

i := 1;

read(ch);

while (ord(ch) <> 10) do { Stop on <CR> }
begin
 word[i] := ch;
 read(ch);
 i := i + 1
end;

while (i < 21) do { Pack with blanks }
begin
 word[i] := ' ';
 i := i + 1
end;

writeln(dictionary, word); { Put in dictionary }

writeln('Done? (Y or N)');
break(output);

readln(ch);

if (ch = 'N') { Spring-loaded to quit }
then circle := true
else circle := false

end
end.

```

Appendix J

STD.DIC LISTING

<  
<  
>  
>  
=  
=  
EQUAL  
=  
EQUALS  
=  
LESSTHAN  
<  
LT  
<  
GREATERTHAN  
>  
GT  
>  
!=  
<>  
<>  
<>  
NOTEQUAL  
<>  
NE  
<>  
WHERE  
WHERE  
IS  
=  
OR  
OR  
AND  
AND  
FOR  
FOR  
WHEN  
WHEN  
EQ  
=  
PRINT  
PRINT  
GREATERTHANOREQUALTO  
>=  
>=  
>=  
GTEQ  
>=  
LESSTHANOREQUALTO  
<=  
<=

<=  
LTEQ  
<=

9

### Vita

Dale VanKirk was born on 29 April, 1951, in Corpus Christie, Texas to Albert and Ellinor VanKirk. In 1969, he graduated from Kent-Meridian High School, Kent, Washington. The next year, he entered Western Washington State College, where he graduated in 1973 among the first Computer Science degree holders on the West Coast. Graduation took him to the Air Force, where he pursued a career as a Weapon Systems Officer in the F-4 Phantom. A remote tour to OSAN AB, Korea in 1976 ended with orders to Pilot Training at Reese AFB, Texas.

May, 1979 found F-4 Pilot VanKirk grounded indefinitely for eye problems. Orders were forth-coming to Luke AFB, Arizona, where Capt. VanKirk was given charge of a software programming division in the SAGE Programming Agency. Within seven months, he was chosen as the 26 Air Division/NORAD Region's Junior Officer of the year. The next year, he was promoted to the position of Assistant Director of the agency, a Lt. Colonel position. He remained in that capacity, ensuring the programs providing NORAD an Air Defense capability were updated properly and efficiently, until entry to the School of Engineering at the Air Force Institute of Technology in June 1982. While there, he was selected as an Outstanding Young Man of America.

## REPORT DOCUMENTATION PAGE

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                            |                                                                                                          |                                       |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|---------------------------------------|
| 1. REPORT SECURITY CLASSIFICATION<br><b>unclassified</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                            | 1b. RESTRICTIVE MARKINGS                                                                                 |                                       |
| 2a. SECURITY CLASSIFICATION AUTHORITY                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                            | 3. DISTRIBUTION/AVAILABILITY OF REPORT<br><b>Approved for public release;<br/>distribution unlimited</b> |                                       |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                            |                                                                                                          |                                       |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br><b>AFIT/GCS/EE/034-21</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                            | 5. MONITORING ORGANIZATION REPORT NUMBER(S)                                                              |                                       |
| 6a. NAME OF PERFORMING ORGANIZATION<br><b>Air Force Institute of Tech.</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | 6b. OFFICE SYMBOL<br>(If applicable)<br><b>AFIT/EE</b>     | 7a. NAME OF MONITORING ORGANIZATION                                                                      |                                       |
| 6c. ADDRESS (City, State and ZIP Code)<br><b>AFIT/EE<br/>Wright-Patterson AFB, Ohio 45433</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                            | 7b. ADDRESS (City, State and ZIP Code)                                                                   |                                       |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | 8b. OFFICE SYMBOL<br>(If applicable)                       | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER                                                          |                                       |
| 8c. ADDRESS (City, State and ZIP Code)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                            | 10. SOURCE OF FUNDING NOS.                                                                               |                                       |
| 11. TITLE (Include Security Classification)<br><b>USER-FRIENDLY INTERFACE TO THE ROTH RELATIONAL DATABASE</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                            | PROGRAM ELEMENT NO.                                                                                      | PROJECT NO.                           |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                            | TASK NO.                                                                                                 | WORK UNIT NO.                         |
| 12. PERSONAL AUTHOR(S)<br><b>VanKirk, Dale Wayne</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                            |                                                                                                          |                                       |
| 13a. TYPE OF REPORT<br><b>FINAL</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | 13b. TIME COVERED<br>FROM <b>83/6/6</b> TO <b>83/12/16</b> | 14. DATE OF REPORT (Yr., Mo., Day)<br><b>83/12/16</b>                                                    | 15. PAGE COUNT<br><b>148</b>          |
| 16. SUPPLEMENTARY NOTATION<br><b>Master's Thesis</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                            |                                                                                                          |                                       |
| 17. COSATI CODES                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                            | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)                        |                                       |
| FIELD                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | GROUP                                                      | SUB. GR.                                                                                                 |                                       |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                            |                                                                                                          |                                       |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                            |                                                                                                          |                                       |
| 19. ABSTRACT (Continue on reverse if necessary and identify by block number)<br><b>A friendly interface to a Relational Algebra Database System was created on the Universal Relation concept. This concept allows the user to relate to the total database as a single relation. The user inputs a query using attributes of the single relation. The interface creates the Universal relation by way of relational algebra JOINS. Tuples of this relation are SELECTed according to constraints placed on attributes in the query, and a PROJECTION of attributes desired is made from the result. Limitations of the interface are:</b><br><br><b>1. All relations in the database must be JOINable without data loss.</b><br><b>2. The query must start with a verb.</b> |                                                            |                                                                                                          |                                       |
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT<br><b>UNCLASSIFIED/UNLIMITED</b> <input type="checkbox"/> SAME AS RPT. <input checked="" type="checkbox"/> DTIC USERS <input type="checkbox"/>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                            | 21. ABSTRACT SECURITY CLASSIFICATION<br><b>unclassified</b>                                              |                                       |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL<br><b>Dale VanKirk</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                            | 22b. TELEPHONE NUMBER<br>(Include Area Code)<br><b>513-255-3533</b>                                      | 22c. OFFICE SYMBOL<br><b>AFIT/ENA</b> |

FILM  
3-8