

A GRAPHICAL QUERY LANGUAGE SUPPORTING  
FLEXIBLE DATABASE ACCESS

by

NANCY LYNN PETERS

A.B., Computer Science and Fine Arts  
Brandeis University, Waltham, Massachusetts  
(1982)

Submitted to the Department of Architecture  
in Partial Fulfillment of  
the Requirements of the Degree of  
Master of Science

at the

Massachusetts Institute of Technology

September 1988

© Nancy L. Peters 1988. All rights reserved.

The author hereby grants to MIT permission to reproduce and  
to distribute copies of this thesis document in whole or in part.

Signature of Author \_\_\_\_\_  
Nancy L. Peters  
Department of Architecture  
August 12, 1988

Certified by \_\_\_\_\_  
Timothy Johnson  
Principal Research Associate, MIT  
Thesis Supervisor

Accepted by \_\_\_\_\_  
William L. Porter  
Chairman, Departmental Committee  
for Graduate Students

MASSACHUSETTS INSTITUTE  
OF TECHNOLOGY

SEP 23 1988  
Rotch  
LIBRARIES

A GRAPHICAL QUERY LANGUAGE SUPPORTING  
FLEXIBLE DATABASE ACCESS

by

NANCY LYNN PETERS

Submitted to the Department of Architecture  
on August 12, 1988 in partial fulfillment of the  
requirements for the Degree of Master of Science

ABSTRACT

GRAF-ASQ (GRaphical And Fully-Accessible Structure-based Queries) is a graphical query language designed to provide flexible, wide access to data via the use of n-tuples and Prolog concepts. It is also designed to provide the ability to view the database schema graphically and to store queries which can be retrieved and from which more complex queries can be built.

The system is built to interface with MacDRAW so that it can store and retrieve information connected to graphical objects. The system is independent of MacDRAW, however. It accepts data in a general format that other programs can give it.

Implemented so far is the ability to view the schema with different central foci and to make atomic attributes invisible. Also implemented is the ability to get data about graphical objects from MacDRAW, including type and simple attribute information, and to query and search for data via menus. If the data found relates to MacDRAW objects, those objects can be highlighted within MacDRAW. The graphical query language itself has not been implemented.

Thesis Supervisor: Timothy Johnson

Title: Principal Research Associate

## **Sponsorship**

**This thesis was funded in part by Apple Computer, Inc.,  
Educational Marketing Division.**

## Table of Contents

<i>Contents</i>	<i>Page</i>
Abstract	2
Sponsorship	2a
Table of Contents	3
Table of Tables	6
Table of Figures	7
Acknowledgements	8
1. Introduction	9
2. Database Background	11
2-1. Databases	11
2-1-1. Introduction	11
2-1-2. Hierarchic	11
2-1-3. Network	12
2-1-4. Relational	14
2-1-4-1. Regular	14
2-1-4-2. Irreducible	16
2-1-5. Entity-Relationship	17
2-1-6. Summary - Classification of GRAF-ASQ	18
2-2. Computer-Aided Design (CAD) Database Facilities	20
2-2-1. Introduction	20
2-2-2. Simple Attributing (AutoCAD)	20
2-2-3. Summary - GRAF-ASQ for CAD	21
3. Recent Developments in Graphical Databases	22
3-1. Introduction	22

<i>Contents</i>	<i>Page</i>
3-2. Guide	22
3-3. LID	23
3-4. SKI	24
3-5. GORDAS	25
3-6. ISIS	27
3-7. QBE	27
3-8. Summary - GRAF-ASQ vs. the Mentioned Systems	28
4. The Reason for this System	29
5. The Use of N-Tuples	32
6. Relational Completeness of Language	38
7. The System	44
7-1. Data Model	44
7-2. Choice of Prolog	47
7-3. Detailed Examples	50
7-3-1. Assigning Attributes	50
7-3-1-1. Simple Attributing	50
7-3-1-2. Complex Attributing	50
7-3-1-3. Putting Functional Attribute Results in Database	51
7-3-2. Schema (or Meta Data)	51
7-3-3. Queries	52
7-3-3-1. Creating	53
7-3-3-2. The Eight Permutations of a 3-tuple	53
7-3-3-3. The N-Tuple	54
7-3-3-4. Complex Query	54

<b>Contents</b>	<b>Page</b>
7-3-3-5. Creating Domains	55
7-3-4. Functions	55
7-3-4-1. Creating	55
7-3-4-2. Executing	56
7-3-4-3. Linking	56
7-3-5. Functional Attributes	56
7-3-5-1. Creating	57
7-3-5-2. Recursive	57
7-3-5-3. Relational Product	58
7-3-6. Comparison Query	59
7-3-7. Relational Database Only Query	59
7-3-8. Modification	59
7-3-8-1. Of Schema	60
7-3-8-2. Of Data	60
8. Conclusion	76
Bibliography	78
Appendix I: Symbol List	83
Appendix II: Legal Connections	84
Appendix III: Syntax	85
Appendix IV: Semantics of Symbols	90
Appendix V: Menues	94
Appendix VI: Windows	97
Appendix VII: Schema Display	98
Appendix VIII: Attribute Desk Accessory	99

## Table of Tables

<i>Table Number</i>	<i>Title</i>	<i>Page</i>
Table 1	Person Relation 1	32
Table 2	Person Relation 1 with 3-tuple highlighted	33
Table 3	Person Relation 2 with atomic and non-atomic values	33
Table 4	Person Relation 3 showing n-tuples	34
Table 5	The Sister, Brother, and Sibling Relations	38
Table 6	The Manufacturer, Part, and Manufacturer/Part Relations	40
Table 7	The Sister Relation where Name = "Joan"	41
Table 8	The Name Relation	42

## Table of Figures

<i>Figure Number</i>	<i>Title</i>	<i>Page</i>
Figure 1	Union in GRAF-ASQ	39
Figure 2	Difference in GRAF-ASQ	40
Figure 3	Product in GRAF-ASQ	41
Figure 4	Selection in GRAF-ASQ	42
Figure 5	Projection in GRAF-ASQ	43
Figure 6	Simple attributing	61
Figure 7	Complex Attributing	62
Figure 8	Schema Views	63
Figure 9	Steps to Create Query	64
Figure 10	The first four of the eight permutations of straight 3-tuples	65
Figure 11	The second four of the eight permutations of straight 3-tuples	66
Figure 12	5-tuple Query	67
Figure 13	Complex Query	68
Figure 14	Creating a Domain	69
Figure 15	Creating a Function	70
Figure 16	Linking Functions	71
Figure 17	Creating a Functional Attribute	72
Figure 18	A Recursive Functional Attribute	73
Figure 19	Comparison Query	74
Figure 20	Relational Database Only Query	75



## Acknowledgments

I want to thank Patrick Purcell for hanging in there for me, Ron MacNeil for introducing me to this project, and Muriel Cooper for her help at various times during my stay at M.I.T. I want very much to thank Stanley Zdonik, whose comments along the way helped me greatly in thinking about the system.

I appreciate the help of Linda Okun, Leon Groisser, Dean Frank Perkins and my soon-to-be little one for helping assure that the thesis got completed this summer.

I want to thank Doug Lanam for all the phone help he gave with his product, Advance A. I. Systems Prolog and for the Prolog itself, which was great to work with.

I especially want to thank my advisor, Timothy Johnson, for the wonderful discussions we had which helped me to develop the system to its present form and for his thoroughness in testing out my ideas along the way.

Lastly, I want to thank all my relatives who gave me encouragement. In particular, I want to thank my Aunt Erica, my sister-in-law, Jan, my parents and, most of all, my sweet love, my husband Bob, who gave me support the entire way.

## 1. Introduction

The system described in this paper, GRAF-ASQ (GRaphical And Fully-Accessible Structure-based Queries) is designed as a graphical query system that will provide a number of advantages:

- flexible, wide, graphical access to data via the use of n-tuples and Prolog concepts, including the ability to graphically store and retrieve queries in the form of "functions".
- the ability to interface with MacDRAW for use with CAD or other graphical data (as well as the built-in possibility of connecting with other programs).
- a clear and simple graphical language.
- a graphical view of the database schema, with the ability to vary the central focus.
- desirable features from both relational and entity-relationship databases.

The paper describes these advantages by giving some background material in relationship to GRAF-ASQ and by describing the system itself.

Section 2 gives a brief history of databases, describing how various types of databases work and placing GRAF-ASQ in the context of those database types. Then, the simple attributing ability present in AutoCAD is described and compared to the more complex, object-to-object attributing available in GRAF-ASQ.

Section 3 discusses recent developments in graphical database systems, including a variety of entity-relationship type databases and a relational type database, QBE. It then compares these systems to GRAF-ASQ.

Section 4 discusses the reason for the system, including detail on the

advantages listed above.

Section 5 goes into detail on the definition of an n-tuple in this system. (The definition is different from that of an n-tuple in a relational database). It first defines a 3-tuple and then expands the definition to an n-tuple. It also explains how general retrieval is done on the n-tuple.

As mentioned before, GRAF-ASQ has desirable features from both relational and entity-relationship databases. Section 6 goes into detail on a desirable relational database feature found in GRAF-ASQ, relational completeness. This feature gives GRAF-ASQ a retrieval flexibility found only in relational databases.

Section 7 provides detail about the system itself, including a data model, why Prolog was chosen as the language for this system, and many examples of how it works. The examples include scenarios of how a user would perform certain actions and a variety of sample queries.

Section 8 is the conclusion.

The appendices provide information on the syntax and semantics of the language, the contents of the menus, and the windows. There is also an appendix which shows constructs for schema display. The last appendix shows the attribute desk accessory, used for assigning attributes.

It should be noted that the ability to view the schema with different central foci and with atomic attributes made invisible is implemented. Also implemented is the ability to get simple attributed data from MacDRAW and to query and search for data via menus. If the data found relates to MacDRAW objects, those objects can be highlighted within MacDRAW. The graphical query language itself has just been designed, not implemented. The ability to query and search via menus is a temporary substitute for the graphical query language.

## 2. Database Background

The following sections will provide background on the general types of databases that exist and on database facilities in computer-aided design environments.

### 2-1. Databases

#### 2-1-1. Introduction

Before computers, filing was the about the only storage and retrieval system available. However, with computers came the invention of databases and, with the invention of databases, came numerous database styles. The following sections will discuss the basic styles of databases: hierarchical, network, relational, and entity-relationship.

#### 2-1-2. Hierarchic

A hierarchic database consists of *trees of records*. Each record has a type. A *record type* is made up of one or more *field types*. Field types identify the pieces of information stored in the record. For example, a person record type might contain a first name field type, a last name field type, a social security field type, etc. A record is a common database structure.

Each tree also has a type. A *tree type* is made up of a *root record type* and an ordered set of zero or more dependent *sub-tree types*. Each sub-tree type, in turn, is made up of a root record type and an ordered set of zero or more dependent sub-tree types. Since all the sub-tree types ultimately relate to the original root

record type, the structure of the entire database is one big tree type.

Types specify the structure of the data. The data itself is referred to as occurrences: occurrences of trees, occurrences of records, occurrences of fields. For example, if there is a person record type, each record that contains information about a specific person is known as an occurrence of a person record, or as just a person record.

The operators in a hierarchic database include:

- an operator to locate a specific tree in the database,
- an operator to move from one tree to the next,
- operators to move from record to record within tree by moving up and down various hierarchic paths,
- operators to move from record to record among all the sub-records under a particular root record type,
- an operator to insert a new record,
- an operator to delete a specified record.

### 2-1-3. Network

A network database is a more generalized form of a hierarchic database.

It differs in the following ways:

- Its "trees" are not really trees. Record types that belong to one root record type can also belong to another.
- There does not have to be one big "tree" for the entire database. There can be any number.

The main components of a network database are known as *sets*

(which correspond to hierarchic trees), *owners* (which correspond to hierarchic root records) and *members* (which correspond to hierarchic sub-records). A set consists of one owner and an ordered set of members. A set is not restricted as much as a tree. Any of the following may be true of a set:

- A member of one set can be the owner of another.
- An owner of one set can be the owner of any number of sets.
- A member of one set can be a member of any number of sets.
- If there is a set with owner of record type A and members of record type B, there can be other sets with this same combination. What would differ is the ordering of members or that a particular owner would have at least one different member in each of the sets.
- Record type A might be the owner record type to member record type B in one set, yet that same record type B may be the owner record type to member record type A in another set.

The operators in a network database include:

- an operator to locate a specific record, given some value for a field in the record,
- an operator to move from an owner to its first member within a set,
- an operator to move from member to member within a set,
- an operator to move from member to owner within a set,
- an operator to create a new record,
- an operator to delete an existing record,
- an operator to modify an existing record -- that is, to modify fields within it,
- an operator to take an existing record that has a record type which is the record type of members of a particular set and connect that record into

the set as a new member,

- an operator to disconnect a current member of the set from the set,
- an operator to disconnect a current member of one set and reconnect it with another set of the same set type.

#### 2-1-4. Relational

The sections below describe the relational database model and a variation, an irreducible relational database model, which is a variation corresponding more directly to GRAF-ASQ.

##### 2-1-4-1. Regular

The structure of a relational database looks very different from that of the hierarchic or network structures. Data is stored in the form of *relations*, which are essentially tables. A relation is made up of *attributes*, *values*, *tuples\** and *primary keys*. A tuple is a row in a relation, an attribute is a column header and each member of the column is a value that corresponds to that attribute, and a primary key is a unique identifier such that each tuple can be distinguished via the primary key alone. Another relational term is *domain*. A domain is a set of values from which an attribute can obtain its specific values from. For example, if there was an attribute City Name, that attribute's domain would be the set of all

\* Note that the term "tuple" here is different than the term "n-tuple" used by GRAF-ASQ. See section 5 for further explanation.

names of cities.

A relation has the following properties:

- It contains no duplicate tuples.
- Its tuples are unordered.
- Its attributes are unordered.
- All the values of its attributes are atomic. This means two things: (1) A row-column position can contain no more than one value and (2) A value cannot itself be a relation.

The operators in a relational database include:

- select, which extracts specified tuples from a specified relation,
- project, which extracts the values of specified attributes from a specified relation,
- product, which builds a relation from two specified relations consisting of all possible concatenated pairs of tuples, one from each of two specified relations,
- union, which builds a relation from two specified relations consisting of all possible concatenated pairs of tuples, one from each of the two specified relations,
- intersect, which builds a relation consisting of all tuples appearing in both of two specified relations,
- difference, which builds a relation consisting all tuples appearing in the first and not the second of two relations,
- join, which builds a relation from two specified relations consisting of all possible concatenated pairs of tuples, one from each of the two specified relations, such that, in each pair, the two tuples satisfy some specified condition,



- divide, which takes two relations, one binary and one unary, and builds a relation consisting of all values of one attribute of the binary relation that match (in the other attribute) all values in the unary relation.

#### 2-1-4-2. Irreducible

Before discussing the irreducible model, let's discuss the related binary model. A binary relational database consists of a bunch of atomic facts. Each fact consists of a primary key, an attribute, and the value for that attribute in relation to the primary key. For example, (person-id-number, first-name, John), would be a single atomic fact (in section 5, this is referred to as a 3-tuple). However, not every piece of data can be reduced in this way. For example, if one wanted to store the batting averages of baseball players, taking into account the year played and the teams they played on, the following problem would arise. If one stored (person-id-number, batting-average, .265), (person-id-number, year, 1980), and (person-id-number, team, Slammers), there would be no way to know that these three pieces of information are actually associated with one another.

One way to get around this would be to allowing nesting, such as (((person-id-number, year, 1980), team, Slammers), batting-average, .265). This would need to be converted into a representation consisting of pointers to the nested atomic facts. For example, it would be stored as (fact-a, batting-average, .265) , where fact-a would be (fact-b, team, Slammers), and fact-b would be (person-id-number, year, 1980). This is representation has the disadvantage of being assymetric and of needing to work with the pointers.

Another way to get around it would be to have special key that would relate the three tuples: (batting-average-id, player, person-id-number),

(batting-average-id, team, Slammers), (batting-average-id, year, 1980), and (batting-average-id, batting-average, .265). This would preserve the atomic fact, or 3-tuple structure, symmetrically, yet it introduces an artificial construct and the need for more data.

Another alternative is a variation on the binary model, known as the irreducible data model. In it, its definition of an atomic fact extends to interrelated data as in the batting average example. To do this it allows information to be composite. Thus, batting-average information stated above could be represented by having the composite information (person-id-number, Slammers, 1980) to produce ((person-id-number, Slammers, 1980), batting-average, .265), or have the composite information (Slammers, 1980, .265) to produce (person-id-number, batting-average, (Slammers,1980,.265)).

#### 2-1-5. Entity-Relationship

An entity-relationship database consists of *entities*, *relationships*, *sub-types*, and *super-types*. An entity is an object of some particular type. A person might be an entity, for example and have the type person type. A relationship is either some property of the object or some way in which it relates to another object. Some entity-relationship databases distinguish between these two types of relationships. In fact, some entity-relationship databases consider a relationship as a special type of entity and so do not even make the entity-relationship distinction.

Entities can have a sub-type, which means that, in addition to having a type A, they a sub-type B and every entity that has sub-type B also have type A. A super-type is actually a type that has been added to make two or more types into

sub-types. For example, if there existed the types tree type and flower type, a desirable super-type could be plant type.

Operators in this type of model vary from database to database, but usually consist of some way to extract information in a way that corresponds to the relational operators selection, projection, union, intersection, and difference.

#### 2-1-6. Summary - Classification of GRAF-ASQ

GRAF-ASQ relates to the irreducible model and the entity-relationship model. It is like the irreducible model in that it stores irreducible chunks of information in a manner similar to that of a irreducible model and that it is relationally complete (see section 6). It does not store irreducible chunks in exactly the same way, however, as it does not use the idea of the composite. Instead, it flattens out the information so that, in the batting average example expressed in section 2-1-4-2, the GRAF-ASQ representation would be (person-1, batting-average, Slammers, 1980, .265). This is extending the idea of the atomic fact expressed in the binary model (see section 2-1-4-2), known as 3-tuple in GRAF-ASQ, into an n-tuple (see section 5).

GRAF-ASQ relates to the entity-relationship model because each irreducible chunk of information either relates an entity (referred to as an object in GRAF-ASQ) to another entity or to some atomic value via a relationship (known as as attribute in GRAF-ASQ). The way the user conceives of data in the GRAF-ASQ environment is similar to other entity-relationship databases. That is, it is conceived as a network of entity and atomic types connected by relationships. Thus, a GRAF-ASQ schema (an overall view of the types and how they interrelate) looks like a typical entity-relationship schema.

The reasons for this combined relational (specifically, irreducible)/entity-relationship design is discussed in further sections (in particular, see section 4).

## 2-2. Computer-Aided Design (CAD) Database Facilities

### 2-2-1. Introduction

Computer-aided design (CAD) systems help a user produce and modify designs more quickly and easily, much as a word processor helps a user produce and modify documents more quickly and easily. In other words, a CAD system can help speed up typical design work. However, there is more that it can do. It can also add features that would be difficult or impossible to have when only working with pen and paper.

One possible feature would be a database facility that would allow a user to store and retrieve information about objects in the design. Thus far, simple attributing, such as found in AutoCAD, is about the limit this feature has been introduced into CAD. The next section will discuss simple attributing in AutoCAD and the summary section will compare simple attributing to what is available with GRAF-ASQ.

### 2-2-2. Simple Attributing (AutoCAD)

AutoCAD allows the user to assign attributes to objects within a CAD drawing. Later the user can retrieve attribute information, restricting the retrieval to only certain attributes. For example, an object can be tagged as a "desk". If the attributes size, manufacturer, and material are associated with this tag, the user will be prompted for the appropriate values to go with the particular desk. When the user wants to retrieve information, he can either individually select the objects he wants data on or can specify certain tags. All the objects

that have those tags will be retrieved. The user can restrict what attributes are retrieved, instead of retrieving all the values for all the attributes associated with the tag. This is basically the extent of the facility. A user can modify attribute information as well, but there is no further ability to search and retrieve.

### 2-2-3. Summary - GRAF-ASQ for CAD

AutoCAD provides a limited attributing capability. GRAF-ASQ extends that ability considerably. First of all, GRAF-ASQ allows two objects to be related to each other. For example, there can be an attribute like "next to" and a desk object can be given the attribute "next to" with, say, a wall object as the value for that attribute.

Another thing added by GRAF-ASQ is more extensive search capabilities. GRAF-ASQ can search for objects that have attributes with certain values. For example, if "manufactured by" was an attribute, it could search for all doors manufactured by Stanley. It can even use operators such as "and", "not" and "or" to construct more complex queries such as "search for all doors manufactured by Stanley that are not taller than 10 feet". The search capabilities extend far beyond this and can be read about it in sections 5, 6, and, especially, 7.

Thus, CAD is an area where database capabilities would be useful, yet are thus far found in only a very primitive form. GRAF-ASQ provides an alternative.

### 3. Recent Developments in Graphical Databases

#### 3-1. Introduction

Most query languages that exist are based on typing textual commands, which is a very error-prone approach. Graphical interfaces, such as employed by programs on the MacIntosh computer, aid in ease of use, learning and in the reduction of errors.

There are not many graphical query languages that exist in experimental stages or in actuality. Here are some examples of the few recently developed graphical query languages and how they compare to GRAF-ASQ. The following are either entity-relationship or relational database systems.

#### 3-2. GUIDE

The GUIDE system [WO82, WO83] provides the means to graphically view a statistically-based database. The user can view the database schema as a graph and manipulate that graph for better viewing. For example, it can be moved and zoomed; nodes can be hidden, and a particular node can be made the central focus. Also, the schema can be viewed on different levels, based on a pre-ranking. The user can also view an individual attribute, seeing all the possible values for the attribute.

By picking items off the graph-like schema, and a using menu, the user can construct queries. A user can create partial queries, view the results, and then put those queries together to create more complex queries.

One of the qualities that most distinguishes this system from other

graphical query systems is that it provides the user with the ability to construct partial queries, test them, and combine them to produce more complex queries. GRAF-ASQ provides this facility as well by the use of functions. Using GRAF-ASQ, a user can create a query, test it out, and save it as a function, reduced to just a name. Then, a user can link up functions or use a function as a part of a more complex query. GRAF-ASQ provides the advantage of allowing the functions to be saved and by allowing the user to use them without having to see the details of their contents.

Another useful facility that GUIDE has is to allow the schema to be viewed based on different central foci. GRAF-ASQ allows this too. In GRAF-ASQ, the user specifies the object type that is to serve as the central focus when displaying the schema. GUIDE allows different levels of the schema to be viewed, based on a pre-ranking. GRAF-ASQ allows different levels to be viewed, based on distance of level from the current central focus. GUIDE is more flexible in this regard.

However, a limiting factor to GUIDE is that its use is directed toward statistically-based databases. GRAF-ASQ does not have this limitation. It can be used for statistical, or other alphanumeric data, and it can also be hooked onto graphical data, such as architectural blueprints.

### 3-3. LID

LID [FO84] is a graphical browser which allows a user to pick one type of entity from a menu and see information about it in a form similar to a deck of 3X5 cards. The user can then browse or search through these 3X5 card look-alikes to select the desired specific entity. From there, the user can see a schema diagram directly connected to the entity and traverse the database to find related



information.

One limitation of traversing in this one-by-one manner is that, in some cases, it could lead to having to go up and down branches to get all the desired information. This was noted in the paper about LID. Another limitation that was noted was its inability to retrieve lists of items. Although an extension, "list-oriented LID", was proposed, this extension did not provide for the use of boolean operators in constructing queries, so even the extension would be limited for selective retrieval.

Although LID is interesting in its graphical, hierarchical card file approach, it is limited in that it does not provide retrieval by query.

### 3-3. SKI

SKI [KI84] is another graphical schema-display/query language. When the article about it was written, the underlying language SEMBASE was in place, but the graphical interface SKI was in the design stage.

As part of its graphical interface, a window/menu is used to display schema and query information, with the window divided into the categories Parent Types, Attributes, Attribute Ranges, Predicates, and Subtypes. Set notation is used to construct sub-types in the "Predicates" section. This can be good if the user is familiar with set notation, otherwise there is that much more for the user to learn. The sub-types help restrict domains and therefore provide the user with an easier means to browse. (Browsing is done via first, last, next, and previous operations). This restricted information can also be reported. However, it does not appear that attributes can be selectively displayed; the user has to display all the attributes of a particular object type.

An interesting feature to this system is that it allows a user to check how any two object types are related. A visual display of the path between the two is shown in the display menu. This facility is not directly provided by GRAF-ASQ. However, although the information would not be in graphical form, the same information is accessible by creating a recursive function that would identify the path of attributes that connect two objects. (See the section on recursive functions).

Another facility provided by SKI is the ability to determine how the deletion of a particular object type would affect other object types. This is actually similar to the "show how two objects related" facility.

Based on these two emphasized features, the main purpose of this system is to allow the user to be able to work with the part of the schema he most wants to work with and not worry about the rest, even if the "part of the schema" really isn't a part at all, but two disconnected parts. In GRAF-ASQ, a part of the schema can be viewed, and functions can be created to retrieve how two objects are related without searching around the schema. However, there is no direct "how related" facility. "How related" is certainly a useful function, yet it should not be provided at the expense of a global overview, which is also important. SKI does not provide a way to view all the interconnections between types at once, or even a section as such. The user must gradually unveil global information.

### 3-5. GORDAS

GORDAS [EL85] is another schema-view/query language which allows the user differing views of the database. The schema will actually change to

correspond with the differing views. For example, when there is a different central focus for a view, a many-to-one relationship may become one-to-one. When this happens, the attributes from the two objects in the one-to-one relationship can be combined and become attributes of one object instead.

For querying, the user chooses attributes from the graphical schema, operators from a menu, and types in any literal values desired. At various steps along the way, if it is necessary to find out if the user wants further conditions, natural language questions are created by the system to help the user create the precise query intended.

The ability to change focus (present also in GRAF-ASQ) and the natural language questioning are good features. However, there is a rather clumsy aspect to the system. The user has to jump from mode to mode to select entities and relationships desired in the query, choose the root entity, select the conditions, select restrictions, and to select attributes to be displayed. This could become very confusing. This confusion is compounded by the distinction that is made between conditions (conditions for the root entity) and restrictions (conditions for other entities). In GRAF-ASQ, the user switches windows, one for constructing queries and one for viewing the schema. However, this is a clear delineation. There is no confusion as to what mode one is in. Also, there is not the choppy distinction made between all those different steps.

Another minor, but potentially bothersome, aspect of the language is that it is based on sets in a way that can confuse the user. In GORDAS, the symbol "=" stands for set equality. When a user wants to indicate what would more typically be "Name = Joan", he has to put "Name Includes Joan", meaning Joan is included in the set of names. To help with this problem, GORDAS prompts a user to make sure he means set equality when he uses "=". However, the need for this kind of

prompting is an artificial appliance to cover up a syntax design problem.

### 3-6. ISIS

ISIS [GO85,ZD86] provides a means for navigating through and modifying schema and data in a consistent, graphical fashion. It provides a query facility. As a result of query searches, sub-classes and attributes are derived. Another feature important to the system is version control. However, GRAF-ASQ is not dealing with this issue.

The system is nice in its consistency. A user needs to change environment only to construct a query. It also shows all data, yet boldfaces the data that is chosen. This is useful when there is not too much data. It would probably be difficult to find boldfaced items in a large list obtained if dealing with a large database, however.

ISIS, GORDAS, LID, and SKI all emphasize browsing as the main access vehicle. Query is used to restrict the browsing domain. GUIDE is somewhat more query-oriented, yet is designed for a restricted type of data (statistical, numerical data). GRAF-ASQ has more of an emphasis on queries, allowing the user flexibility and wider capability in this area. Besides working on alphanumeric data, it connects to graphical data.

### 3-7. QBE

QBE [DA86] is a graphical version of a relational database and also has an emphasis on search. It allows a user to create queries by placing variables, operators, and literals into a skeleton relational table, instead of having to make

query statements.

It is useful that the form that data is displayed and the form in which queries are constructed are the same. However, the commands themselves are cryptic and it is not always intuitively obvious how to work with relational constructs, such as joins.

### 3-8. Summary - GRAF-ASQ vs. the Mentioned Systems

GRAF-ASQ is designed to provide a graphical query language that captures the best of both the relational and entity-relationship worlds. It is designed to provide flexibility and wider capability in query construction, as is provided in a relational language, by providing n-tuples and relational completeness (see sections on "The Use of N-Tuples" and "Relational Completeness of Language"). It takes advantage of Prolog languages concepts for retrieval and to provide the user with the flexibility provided by functions and functional attributes. Even functional attributes that use recursion can be created.

It also takes the more intuitive form of an entity-relationship type structure. A user can think in more real-life terms of entities, or objects, rather than tables. Although GRAF-ASQ does not provide all the features present in all the languages mentioned, it provides the bulk of them, including one of the most useful: a schema view with a central focus that can be varied. This is very important because it aids the user in remembering the structure of the data so that accessing it can be accomplished more easily. With the use of relational tables, the user has to depend upon his memory because there is no overall view as such.

#### 4. Reason for this System

Some recently-developed graphical database systems have been mentioned and compared to GRAF-ASQ. They all provide a graphical interface with either a relational or entity-relationship structure. None of them, however, actually work with graphical data and none have a graphical query language whereby the components of the language are graphical symbols. Those that have query capabilities at all provide a purely menu/textual form for the query language.

In addition, none of the non-relational databases have full relational query capabilities. The one relational graphical database, QBE, has full relational capabilities, yet it has the non-intuitive relational database form which forces the user to think in terms of relational tables, instead of a world of objects.

Thus, the following is a list of the important features that distinguish this system:

1. *Provides flexibility and wide capability in the area of queries.*

The flexibility and wide capability is there because GRAF-ASQ is based on the PROLOG programming language. It is designed to take advantage of PROLOG's extensive capabilities. It is designed to allow the user to store queries in a form called a function (as it takes input, processes it, and provides output). More complex queries can be built from the functions. It is also designed to allow the user to define functional attributes, which are attributes based on queries, instead of on straight data. The user can even create multi-definitions and recursive definitions for an attribute.

2. *Provides a means for searching on MacDRAW graphics items.*

In GRAF-ASQ, the user can assign attributes to MacDRAW graphics items, search on those items, and see what was chosen on the MacDRAW item.

3. *Provides a graphical language for querying that is simple and clear.*

The graphical query language is based on a small number of graphical constructs that can be used to create sophisticated statements. The user need not bother typing in queries. Instead, he chooses and puts together graphical constructs.

4. *Allows the user to view the database structure graphically.*

The user can view the schema graphically, can choose the central focus of the schema, and can choose whether or not to have atomic types be visible or invisible.

5. *Interfaces with other programs.*

GRAF-ASQ need not be used with MacDRAW, data can be brought in from other MacIntosh sources. Also, output can be sent to other systems for fancy formatting, such as to a generalized spreadsheet program.

6. *Has desirable features from both relational and entity-relationship databases.*

GRAF-ASQ is relationally complete, so it provides all the capability of a relational database language. It also provides the more intuitive view of objects, their attributes, and relationships between objects that is provided by an entity-relationship database.



## 5. The Use of N-Tuples

One of the goals of GRAF-ASQ is to provide flexible and wide access to data stored in the database. To aid in this, GRAF-ASQ uses the n-tuple as its basis. Using the capabilities of Prolog, each piece of information in the n-tuple is easily accessible.

To demonstrate this, let's define the term n-tuple as it applies in GRAF-ASQ. Then we will see how the form of the n-tuple allows wide access to data. In GRAF-ASQ, an n-tuple is defined as follows:

(<object> <attribute> {<qualifier>} <value>)

where {} means zero or more.\*

A 3-tuple is a sub-instance of an n-tuple and is defined as follows:

(<object> <attribute> <value>)

For the definition of <attribute>, <object>, and <value>, we will look only at 3-tuples.

Then, we will look at the more general n-tuple to define <qualifier>.

The definition of an <attribute> is based on relations. For example, let's look at the Person Relation.\*\* Person has the <attribute>s Name and Eye Color:

	<u>Name</u>	<u>Eye Color</u>
person-1	Frieda	blue
person-2	Joe	blue
person-3	John	brown
person-4	Sally	green

Table 1. Person Relation 1.

\* In the language of relational databases, an n-tuple is different. It is a row from a relation, consisting of n elements. Thus, looking at table 1, the n-tuple for person-1 would be (Frieda, blue). When n-tuples are mentioned in this paper, they are defined as above, unless explicitly specified otherwise.

\*\* Person-1, person-2, etc. are not items found in relations. They are placed here to help express the non-relational concept of object.

Thus, an <attribute> is the name of a set of members (e.g. Name for (Frieda, Joe, John, Sally) or Eye Color for (blue, blue, brown, green)), in terms of relations. Before describing an <attribute> in terms of a 3-tuple, let's look at what <object>s and <value>s are in terms of relations.

In the relation above, the <object>s are person-1, person-2, person-3, and person-4. They correspond to what the <attribute>s belong. A <value> is the specific <attribute> value that belongs to a specific <object>. For example, Frieda is a <value> in the relation above; it belongs to person-1 as the <attribute> Name's value.

Thus, a 3-tuple is a piece of a relation:

	<u>Name</u>	<u>Eye Color</u>
person-1	Frieda	blue
person-2	Joe	blue
person-3	John	brown
person-4	Sally	green

Table 2. Person Relation 1 with 3-tuple highlighted.

The piece of the relation highlighted is the 3-tuple (Name person-1 Frieda).

Another thing to note is that an <object> in one 3-tuple can be a <value> in another. Look at the following relation:

	<u>Name</u>	<u>Eye Color</u>	<u>Sibling</u>
person-1	Frieda	blue	person-2
person-2	Joe	blue	person-1
person-3	John	brown	
person-4	Sally	green	

Table 3. Person Relation 2 has atomic and non-atomic values.

One 3-tuple from this relation is (Sibling person-1 person-2), where person-2 is the <value>, yet another 3-tuple is (Name person-2 Joe), where person-2 is the

<object>. A <value> that is also an <object> in another 3-tuple is known as a <non-atomic value>. A <value> which is never an <object>, like Frieda in this example, is known as an <atomic value>.

We have only dealt with 3-tuples so far, so <qualifier>s have been ignored. Let's look at the <attribute> Average, which is the <attribute> of a 5-tuple, to produce a definition for a <qualifier>:

	<u>Team</u>	<u>Year</u>	<u>Average</u>
person-1	team-1	1982	.310
person-1	team-1	1983	.290
person-1	team-2	1983	.280
person-1	team-2	1982	.295
person-2	team-2	1984	.268

**Table 4.** Person Relation 3, showing 5-tuples.

Notice how the attribute Average is dependent on the attributes Team and Year. One <object>, person-1, has several different <value>s for Average based upon the <value>s for Team and Year. Thus, the 3-tuple (person-1 Average .310) is correct, but the 3-tuple (person-1 Average .290) is also correct and there is further database information to make the distinction between the two meaningful. The further information is the Team/Year combination that goes with each.

Thus, to give the most information about the <attribute>, Average should be a nested 3-tuple:

( ( ( person-1 Year 1982) Team team-1) Average .310).

An n-tuple (in this case, a 5-tuple) is a shortened form of a nested 3-tuple. To construct the n-tuple, the <attribute> would probably be given a more specific name like Team-Year-Avg and the 5-tuple for this example would be:

( person-1 Team-Year-Avg team-1 1982 .310).

Team-1 and 1982 are <value>s within the 3-tuples ( person-1 Team team-1)

and ( person-1 Year 1982), respectively, but are <qualifier>s within the 5-tuple shown above. They serve to further qualify the <object>-<value> relationship present in the 5-tuple.

Ordering of <qualifier>s can, but does not always, have significance. In the example given, Team comes before Year, but this is not significant because a person could play several years on one team or on several teams within one year. If, however, a person could play several years on one team, yet only be on one team within one year, this order would have significance. Team would qualify Year.

Now that n-tuples have been defined, let's see how they provide flexible, wide access to data. Suppose we have the following database of n-tuples:

```
(house-a has-room room-1)
(house-a has-room room-2)
(room-1 is-next-to room-2)
(room-1 has-window window-1)
(room-1 is-room-type livingroom)
(house-b has-room room-3)
(room-3 is-room-type livingroom)
```

the following information will be easily accessible:

Does house-a contain room-1?

Expressed as:

(house-a has-room room-1)

Retrieves:

yes

What rooms does house-a have?

Expressed as:  
(house-a has-room ?)

Retrieves:  
room-1  
room-2

How is house-a related to room-1?

Expressed as:  
(house-a ? room-1)

Retrieves:  
has-room

Which rooms are livingrooms?

Expressed as:  
(? is-room-type livingroom)

Retrieves:  
room-1  
room-3

Tell me about what relates to room-1.

Expressed as:  
( room-1 ? ?)

Retrieves:  
is-next-to room-2  
has-window window-1  
is-room-type livingroom

Tell me what objects are related by 'has-room'.

Expressed as:  
(? has-room ?)

Retrieves:  
house-a room-1  
house-a room-2  
house-b room-3

Tell me about what room-2 relates to.

Expressed as:  
(? ? room-2)

Retrieves:  
house-a has-room  
room-1 is-next-to

Retrieve all the information in the database.

Expressed as:  
(? ? ?)

Retrieves:  
house-a has-room room-1  
house-a has-room room-2  
room-1 is-next-to room-2  
etc.

That is, every permutation of missing object, attribute and/or value can be retrieved.

Thus, the n-tuple provides a means for easy access of not only objects and the values of their attributes, but also the more unusual information: how an object is related to a value, what objects and values have a certain relationship, etc. This is more generalized retrieval than a typical query language provides.

## 6. Relational Completeness of Language

According to Date [DA86], a relationally complete language must be able to perform the five basic relational operations: union, difference, product, selection, and projection. GRAF-ASQ is able to perform these operations. Note that in this section, tuples will be defined according to the relational language definition (see the section entitled N-Tuples).

In relational language, the union of relations A and B is relation C, the set of all tuples belonging to either relation A, relation B, or both. In GRAF-ASQ, inclusive-or can be used to access the same information as union would create in a new relation. As an example, here are three relations: Brothers, Sisters, and their union, Siblings:

	<u>Name</u>	<u>Has Sister</u>
person-1	Joan	person-2
person-2	Ann	person-1
person-3	Bill	person-1
person-3	Bill	person-2

	<u>Name</u>	<u>Has Brother</u>
person-1	Joan	person-3
person-2	Ann	person-3

	<u>Name</u>	<u>Has Sibling</u>
person-1	Joan	person-2
person-1	Joan	person-3
person-2	Ann	person-1
person-2	Ann	person-3
person-3	Bill	person-1
person-3	Bill	person-2

**Table 5.** The Sister, Brother, and Sibling Relations.

In GRAF-ASQ, the following functional attribute could be created:

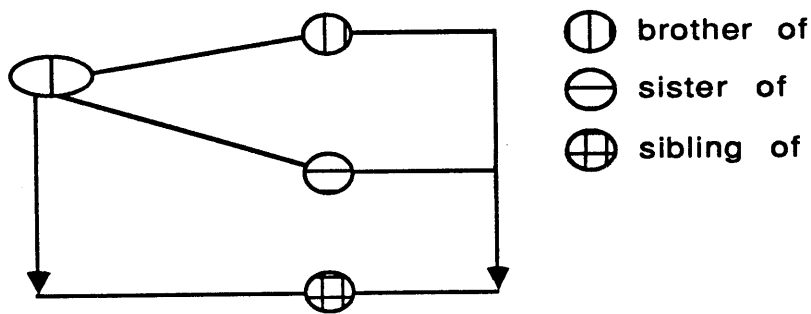
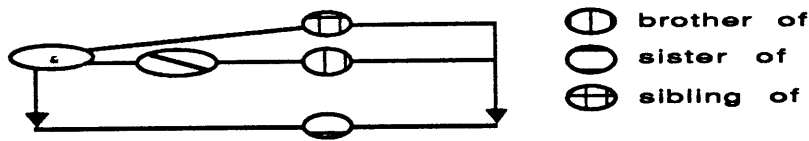


Figure 1. Union in GRAF-ASQ.

This would provide a "functional" attribute "sibling of" that would retrieve a brothers, sisters, or both of a given person. Thus, its effect would be like having sibling data created from the current brother and sister data. Via a menu command, used in conjunction with that functional attribute, the user could even request all the siblings be calculated and then entered into the database as straight data. If the user wishes, he could also indicate, via a menu item, that siblings should be entered into the database incrementally every time a user uses the attribute in a query.

Difference could be accomplished in a similar fashion. In relational language, the difference of relations C and A (C minus A) is relation B, the set of all tuples belonging to relation C, but not relation A. (If one calculates A minus C, the result is relation D, the set of all tuples belonging to relation A, but not relation C). Using the same relations as in Table 5, the Sister Relation is the difference when the Brother Relation is subtracted from the Sibling Relation. GRAF-ASQ has the ability to produce the equivalent difference. If the attributes "sibling of" and "brother of" already existed, the functional attribute "sister of" could be created to accomplish this:





**Figure 2.** Difference in GRAF-ASQ.

This "sister of" functional attribute is similar to the functional attribute "sibling of" described in the union example. "Sister of" in this example would be obtained from the calculation of the query "'sibling of' and not 'brother of'". Also, similarly, the "sister of" data could be entered into the database, via a menu item, as straight data.

The product of relations A and B is relation C, such that each tuple in relation A is concatenated with each tuple in relation B. In the relations shown below, the product of the Manufacturer Relation and the Part Relation is the Manufacturer/Part Relation:

	<u>Manufacturer Name</u>		
manufacturer-1	ABC Corp.		
manufacturer-2	Parts R Us		
	<u>Part Name</u>		
part-1	Gadget		
part-2	Widget		
	<u>Manufacturer Name</u>	<u>Part</u>	<u>Part Name</u>
manufacturer-1	ABC Corp.	part-1	Gadget
manufacturer-1	ABC Corp.	part-2	Widget
manufacturer-2	Parts R Us	part-1	Gadget
manufacturer-2	Parts R Us	part-2	Widget

**Table 6.** The Manufacturer, Part and Manufacturer/Part Relations.

In GRAF-ASQ, one can again create a functional attribute. Shown below is

a special functional attribute that takes two domains and connects each member of Domain A with each member of Domain B, via the specified attribute:

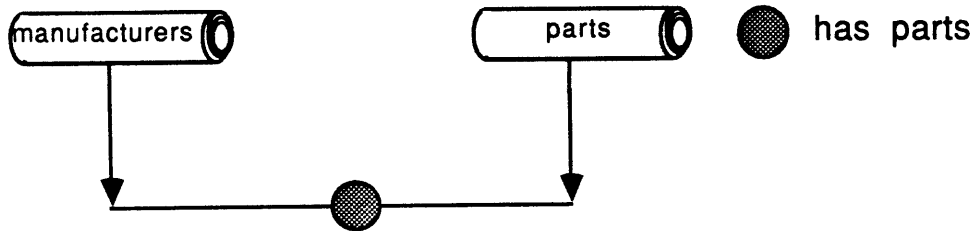


Figure 3. Product in GRAF-ASQ.

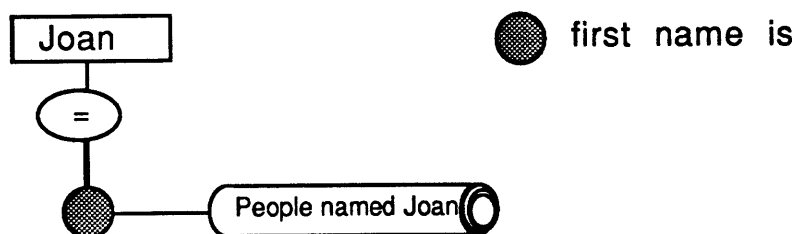
Thus, if manufacturer-1, manufacturer-2, part-1, and part-2 exist in the database, creating the functional attribute in Figure 3, and using a menu item, will cause the creation of the GRAF-ASQ n-tuples (manufacturer-1 has-part part-1), (manufacturer-1 has-part part-2), (manufacturer-2 has-part part-1) and (manufacturer-2 has-part part-2).

Selection is choosing only certain tuples from relation A based on a condition relating to one of the attributes in the relation. As an example, take the Sister Relation shown in Table 5. One selection on that relation would be to select on "Name = 'Joan'". Relation B, the resultant relation, is shown in Table 7.

	<u>Name</u>	<u>Has Sister</u>
person-1	Joan	person-2

Table 7. The Sister Relation where Name = "Joan".

In GRAF-ASQ, the following domain-creation query would accomplish the same thing:



**Figure 4.** Selection in GRAF-ASQ.

Although Figure 4 does not show anything about sisters, remember that in GRAF-ASQ, once one has access to a non-atomic value, as is every member of a domain, one can access all the attributes of that non-atomic value. In this case, sisters of Joan, as well as all other attributes of Joan, would be accessible using the Domain "People named Joan".

Projection is simply taking attribute(s) (i.e. column(s)) from relation A to create relation B. Thus, if one takes the Sibling Relation from Table 5 and projects attribute "Name", the following relation would result:

	<u>Name</u>
person-1	Joan
person-2	Ann
person-3	Bill

**Table 8.** The Name Relation.

In GRAF-ASQ, the following query would provide the equivalent information:

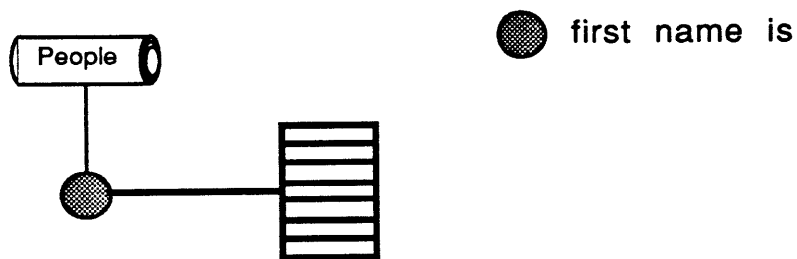


Figure 5. Projection in GRAF-ASQ.

This query would produce a list of names, not a new relation. However, the relation is essentially in the database anyway, in the form GRAF-ASQ n-tuples (person-1 first-name-is Joan), (person-2 first-name-is Ann), (person-3 first-name-is Bill). Performing this query would just let the user see this select information.

Thus, GRAF-ASQ can perform the five basic relational operations and is therefore relationally complete.

## 7. The System

The following sections will discuss the system in detail. Included is a section that describes the data model, a discussion of why Prolog was the chosen language to program the system in, and a collection of numerous scenarios and examples used to describe how the system works.

### 7-1. Data Model

This section describes how data should be viewed in GRAF-ASQ in the context of the main components that make up the system.

The GRAF-ASQ database contains objects which correspond to items found in the everyday world. Each object has a type. Each type has zero or more attributes associated with it. Thus, a "person type" might have the attributes "first name", "last name", "social security number", "spouse", etc. associated with it. Attributes can either be straight data or calculated based on other information. For example, "mother-in-law" might be calculated using the "spouse" and "mother" attributes. An attribute that is calculated is known as a functional attribute.

Each object can have a value associated with each of its type's attributes. All objects within a type do not have to have values for all its type's attributes, however. Each value also has a type. A value can either be atomic (has a type with zero attributes associated with it) or non-atomic (has a type with one or more attributes associated with it).

Object-attribute-value information is in the form of a 3-tuple which can be extended to an n-tuple. A 3-tuple has the form (<object> <attribute> <value>) and an n-tuple has the form (<object> <attribute> {<qualifier>} <value>). (See section 5 for

full definitions of 3-tuples and n-tuples).

The following describes domains, lists, and queries. Note that, in GRAF-ASQ, these components are graphical. (Examples of this are given in section 7.3). However, in this section, they will be discussed as if they were textual.

Domains are collections of objects, usually of the same type, but not necessarily. Domains can contain all the objects of the database, or some restricted portion. For example, the "person domain" might consist of all objects of type "person type". A further restriction might produce the domain "people named John", which would consist of objects of type "person type" with each of their "first name" attributes having the value "John". Domains are either in the system by default or are created via the use of queries (see section 7-3-3-5).

Lists are sets of sets of atomic values. The values within a set-within-a-set relate to each other. Thus, a list might consist of a set of "first name/last name" sets. In this case, the value for first name would belong to the same object as the value for last name in each set. The list might look like this: ((John,Smith),(Mary,Jones),...). Lists are created via the use of queries. In fact, lists (and sometimes domains) can be considered the results of queries.

A query is a request to retrieve data from the database. The request is built by using operators to associate n-tuples with one another. Either the query is requesting a confirmation of the existence of certain data in the database and no unknowns are needed or the query is requesting the retrieval of data and unknowns must be placed in the query request. For example, a query for confirmation of existence might be:

(joan,sibling,sue) and (sue,sex,female).

Notice there are no unknowns (variables) in this query. If data was desired, the query might be:

(joan,sibling,?) and (?,sex,female).

The result would be data, the list ((sue)).

A function is a storable form of a query. It appears in a reduced form so that the details of how the query works is not shown. However, it can be expanded back into full query form.

It was mentioned above that queries contain operators. An operator is one of such boolean operators as "and", "not", "inclusive or" and "exclusive or" or one of such relational operators such as "=", "<", and ">".

There are other constructs that are used in creating queries, functions, and functional attributes. They will not be mentioned, however, since this section was written to give a general idea of how data is stored and retrieved in the system and discussion of the other constructs is not necessary.

## 7-2. Choice of Prolog

This section includes a description of how Prolog works on a basic level and a discussion of why it is used for GRAF-ASQ.

Prolog is a programming language that was designed mainly for the development of artificial intelligence programs, especially expert systems. Its ability to store and retrieve information in a form whereby solutions to questions may be derived make it ideal for use with expert system applications. As a side effect, its storage and retrieval capabilities make it useful for database programming as well. Prolog works as follows:

A Prolog program consists of "facts" and "rules". A fact is straight data in the form of:

`<functor>.`

or

`<functor> (<argument> {,<argument>}),`

where {} means zero or more. An example of a fact would be "sister (joan,sue)." In this case, "sister" is the functor and "joan" and "sue" are arguments. (Note that "joan" and "sue" begin with lower-case letters because they are constants, not variables. This is explained further below).

If a programmer wanted a way to determine "sister" without there being any "sister" facts, he could create a rule. A rule has the form:

`<goal> :- <goal> {<separator> <goal>}.`

A <separator> is either a "," (which represents "and") or a ";" (which represents "or"). A goal has the same form as a fact, except its arguments can be variables instead of constants. The goal on the left side of the rule is the goal solved for and its arguments must be variables. The way Prolog distinguishes between constants and variables is by having constants begin with lower-case letters and having



variables begin with upper-case letters.

Below is an example of a sister rule:

```
sister (Sibling1, Sister2) :-  
    (sibling (Sibling1,Sister2); sibling (Sister2,Sibling1)),  
    female (Sister2),  
    not (Sibling1 = Sister2).
```

If given the goal "sister (Sibling1,sue)", Prolog would look for a fact that has the form "sister (<some constant>,sue)" or a rule with a left-side goal of the form "sister (Variable1,Variable2)". If the database contained the fact "sister (joan,sue)", Prolog would report "Sibling1 = joan" as the answer. If instead, it contained the sister rule stated above and the following facts:

```
sibling (joan,sue).  
female (sue).
```

the answer would also be "Sibling1 = joan", but the answer would need to be solved by Prolog, instead of just found.

The example goal of "sister (Sibling1,sue)" uses only one variable. However, a goal can have more than one variable. If given the goal "sister (Sibling1,Sister2)", Prolog would return:

```
"Sibling1 = joan"  
"Sister2 = sue".
```

A goal can have a variable for any or all of its arguments.

Now that a limited description of the workings of Prolog has been given, the following is a discussion of why it was chosen as the language for GRAF-ASQ. First of all, it provides a means to retrieve the eight permutations of n-tuples very easily (see section 5 on The Use of N-Tuples). Notice that in the fact "sister (joan,sue)", "joan" is similar to an object, "sue" is similar to a value, and "sister" is

similar to an attribute. Thus, using this form, objects and values can be searched for easily. This ability is not sufficient to allow searching on the eight permutations, however. There must be a way to search for attributes, too. Even though Prolog has no way to search on a functor directly, a trick can be used to solve this problem.

The way to solve the problem is to store the fact in a different way. Instead of storing "sister (joan,sue)", the programmer can store "data (joan,sister,sue)". Then a variable can be put in the "sister" (attribute) position, as well as the "joan" (object) and/or "sue" (value) positions and thus a search can be done on any of the eight permutations.

Another thing about Prolog that makes it useful for GRAF-ASQ is that the mechanism of rules facilitates the use of functions and functional attributes. A function is basically a storable query. In Prolog, a function can be stored as a rule. A functional attribute is a special attribute that is calculated. Similar to a function, it can be stored as a rule. (In this case, the attribute would have to be in the functor position, which would not correspond to the attribute position used when an attribute is part of a fact. However, a Prolog program can be written to get around this problem).

Since Prolog allows rules to have the same name, there can be different queries with the same name and different functional attributes with the same name. This allows for the creation of recursive functional attributes, as discussed in section 7-3-5-2.

Thus, there are numerous reasons for the choice of Prolog for this application. In fact, GRAF-ASQ is almost a graphical Prolog, taking advantage of its main capabilities. However, although the graphical query language has some correspondence to Prolog syntax, it does not directly imitate it.

### 7-3. Detailed Examples

This section gives a variety of scenarios and examples demonstrating how the system works and some of the more unusual and desirable actions it was designed to perform.

#### 7-3-1. Assigning Attributes

This section describes various ways in which attributes can be assigned to objects. There are three basic ways: simple attributing (assigning atomic values), complex attributing (connecting objects), and putting functional attribute results into the database.

##### 7-3-1-1. Simple Attributing

Figure 6 shows how MacDRAW objects are given simple attributes (attributes that have atomic values). First a user selects a MacDRAW object. Then he chooses the desk accessory "Attribute". The user then types in the type of object and the attributes and values he wants to give it. When finished, the data obtained from the desk accessory can be brought into the database by using the menu command "Get from MacDRAW".

##### 7-3-1-2. Complex Attributing

Figure 7 shows how MacDRAW objects are given complex attributes (attributes that connect objects to one another). First a user selects a MacDRAW

object. Then he gets into the Attribute Desk Accessory and defines the type, attributes, and values as desired. If he wants to assign an attribute that has a non-atomic value, instead of typing in a value, he selects the "Get Non-Atomic Value" button and is brought back to MacDRAW where he selects another object. Then he goes back to the desk accessory, where he can a type, attributes, and values to that second object. If he goes back to MacDRAW, chooses the first object again, and goes backs into the desk accessory, he will notice that the value of the attribute that needed a non-atomic value will be filled in with "object of type <second object's type>".

#### 7-3-1-3. Putting Functional Attribute Results in Database

Functional attributes (see section 7-3-5) are calculated every time a query uses them. There is an attributes menu command "Functional to Database", however, that allows the user to essentially make a functional attribute into a regular attribute. When the command is executed on a selected functional attribute, all the n-tuples which currently fulfill the requirements of the functional attribute will be created and placed into the database. Thus, if there was a functional attribute "sister", and by calculating it Ann was found to be John's sister and Jill was found to be Tom's sister, the data (John, sister, Ann) and (Tom, sister, Jill) would be entered into the database.

#### 7-3-2. Schema (or Meta Data)

To specify the view of a schema to be displayed, a user chooses the center object type by selecting the menu item "Choose Center". The user will be given a

choice of the possible object types from which he can choose the center object type. He can also choose how many levels from the center object type he wants displayed by choosing the schema menu item "Choose Number of Levels".

For example, if the chosen center was "person type" and it connected to "company type" via the attribute "works at", "company type" might be further connected to "location type". If the user requested to see one level, "location type" would not show, as it is part of level two. The exception to that would be if "person type" was also connected to "location type". Then "location type" would be part of level one and would show. The connection of "company type" to "location type" would also show in that case because all the connections between any types that are currently displaying will be shown.

Another option for the user is to use the menu item "Include Atomic" to specify whether or not types for atomic values should be shown. For visual distinction, atomic types are displayed in a lighter box than non-atomic types.

Figure 8 shows two example views of the same schema with the following options specified:

- (1) "person type" as the center, number of levels = 1, include atomic off.
- (2) "company type" as the center, number of levels = 2, include atomic on.

### 7-3-3. Queries

This section describes how to create queries and gives examples of a variety of them as a hint to what's possible. Examples of further possibilities are expressed in later sections.

### 7-3-3-1. Creating

Figure 9 shows the step-by-step creation of a query. First a user views the schema and chooses the attributes desired for the query. These attributes, with their tuple connectors already attached, are automatically put into the query window. Then the user selects the desired operators and connects them to the tuple connectors. Then the user selects the desired input/output symbols, attaches them where desired and the query is created. To execute the query, the user then chooses "Execute Query" from the query menu and the results appear in the list window.

### 7-3-3-2. The Eight Permutations of a 3-tuple

Figures 10 and 11 show a series of example queries which correspond to the eight permutations of a straight 3-tuple query as discussed in section 6. The permutations are:

(<object> <attribute> <value>)

(<object> <attribute> ?)

(<object> ? <value>)

(? <attribute> <value>)

(<object> ? ?)

(? <attribute> ?)

(? ? <value>)

(? ? ?)

### 7-3-3-3. The N-Tuple

Figure 12 shows an example of a query on an n-tuple of higher dimension than a 3-tuple (namely, a 5-tuple). The form of graphic used distinguishes the special relationship the non-attribute members of the n-tuple have. That is, how they are inter-related, and should, therefore, not be divided into separate 3-tuples. However, the graphic is also consistent, so that the similarity between a 3-tuple and n-tuple is seen. In a relational database, the distinction is not made and, in most other databases, the consistency is not there, some kind of additional construct is needed to express the special multi-relationship present in an n-tuple.

### 7-3-3-4. Complex Query

In context of GRAF-ASQ, "complex query" is a query that has at least one "and" and at least one "or" connecting tuples. The "and"s and "or"s used as part of an input do not count, Figure 13 is an example of a query that makes liberal use of "and"s and "or"s. Notice that the English statement almost directly conforms to the graphical version. The "or" divides the two phrases "located in Iowa and have female employees" and "located in Kansas and have employees that earn over \$25,000", just as in the English statement. "Retrieve all names of all companies" corresponds to the part of the graphical query which includes the "company name" attribute connected to the output list.

#### 7-3-3-5. Creating Domains

Figure 14 shows an example of a construct to create a domain. The construct is really a special type of query, so the process for producing it is the same as for a query. The user picks the attribute "year born", chooses the operators needed, then chooses the input/output symbols needed. The output is the "unnamed domain". Then, the user chooses "Execute Query" for the query menu. When the query is complete, the user can name and save the resultant domain by choosing the "Save Domain" menu item from the input/output menu.

#### 7-3-4. Functions

Functions provide a means to store and re-execute queries and to connect them to one another to produce more complex queries. The following sections will describe how to create a function, how to execute a function and how to link two functions together.

##### 7-3-4-1. Creating

Figure 15 shows, step-by-step, how to create a function. First the user creates a query. Then the user surrounds everything but the input/output part of the query with a function box. (The user chooses "Function Box" from the functions menu and is then able to draw a function box). Then he chooses the "Reduce Function" menu item to reduce the function, and finally he chooses the "Save Function" menu items to name and save it.



#### 7-3-4-2. Executing

A function is actually a query in a different form and is executed like a query. The user either creates a function as stated before or selects one by choosing the functions menu item "Select Function". After the function is brought into the query window, he chooses the menu item "Execute Query" to execute it.

#### 7-3-4-3. Linking

Figure 16 shows how to link one function to another to produce a more complex query. First the user selects the functions that are to be linked and they will appear in the query window. Then he chooses "Regular Connection" from the connections menu and uses it to connect the output of one of the functions to the input of the other. Then he chooses "Link Functions" from the functions menu and the two functions are linked together. A function can be linked to an ordinary query in the same way.

#### 7-3-5. Functional Attributes

Functional attributes are special attributes that are calculated instead of being part of n-tuples in the database. They are calculated based on a query that has no input/output symbols. They can be converted to regular attributes via a menu command, however (see section on Putting Functional Attribute Results in Database).

An example of how to create a functional attribute and an example of a

recursive functional attribute are shown below. Also shown is an example of how to create a relational product, which is done with the use of a functional attribute.

#### 7-3-5-1. Creating

Figure 17 shows how to create a functional attribute, step-by-step. First the user creates a query without input/output symbols. Then he selects "Undefined Attribute" from the attributes menu. He chooses "Define-Functional-Attribute Connector" from the connectors menu and uses it to connect the query without input/output symbols to the undefined attribute. Then the user chooses "Save Attribute" from the attributes menu and is prompted for a name. The functional attribute is saved with that name and a pattern is assigned to the attribute symbol.

#### 7-3-5-2. Recursive

Figure 18 shows an example of a recursive functional attribute. This is useful in producing generalized attributes in certain situations. For example, without recursion, one would have to create a grandparent attribute, a great-grandparent attribute, etc. to get at all the ancestors of a person. Using the recursive ability, one would only need to create an ancestor attribute, as shown. To create a recursive functional attribute, the user actually creates two definitions for the attribute. One terminates and one is recursive (refers to itself within its definition). When the attribute is executing, the terminating definition will be tried. The recursive definition will only be tried if the terminating definition fails.

The capability to create two definitions for a functional attribute is not restricted to the times when a recursive attribute is desired. A user can create more than one definition in any case.

### 7-3-5-3. Product

To create a product, the user creates a functional attribute definition whereby the ends of the undefined attribute connects to two domains. Figure 3 shows an example of this. Then the user names and saves the new functional attribute via the attributes menu item "Save Attribute". Then the user chooses the menu item "Functional to Database" from the attributes menu. This will cause each item from the first domain to be associated with each item of the second domain via the newly-defined attribute. Thus, a 3-tuple will be created for each first domain object/second domain object combination as follows:

```
(object-1-in-domain-1  newly-defined-attribute  object-1-in-domain-2)
(object-1-in-domain-1  newly-defined-attribute  object-2-in-domain-2)
.
.
(object-1-in-domain-1  newly-defined-attribute  object-n-in-domain-2)
.
.
(object-n-in-domain-1  newly-defined-attribute  object-1-in-domain-2)
(object-n-in-domain-1  newly-defined-attribute  object-2-in-domain-2)
.
.
(object-n-in-domain-1  newly-defined-attribute  object-n-in-domain-2).
```

### 7-3-6. Comparison Query

In a comparison query, two unknown values are compared. In figure 19, the unknown age of John is compared to the unknown ages of all people in the database to produce a list of the names of all people older than John. This is done by setting either an equal to, less than, and/or greater than symbol between the two unknowns.

### 7-3-7. Relational Database Only Query

Figure 20 shows a query that is difficult for non-relational databases to handle: "Retrieve the names of any departments that sell every item that some company makes". As shown, GRAF-ASQ can handle it. First the user creates a functional attribute which determines, for each department, what companies make items that the department does not sell. Then the user creates a query, using that functional attribute, that says "Retrieve the name of any department whereby there is some company that it does not 'not sell all items the company makes'". When the double negative is converted to a positive, this query translates to the original.

### 7-3-8. Modification

The discussion has focussed on querying and related operations thus far. Another important feature of a database is modification. The following sections will discuss how to modify the schema and how to modify data.

#### 7-3-8-1. Of Schema

At this point, schema data can only be added (via the attribute desk accessory), as shown in figure 6. Along with attribute and value data, type information is entered. Type and attribute information make up the schema. In most databases, a schema needs to set up beforehand in its entirety and cannot be added to easily, so it is an advantage to be able to add to a schema as one proceeds. However, there is no facility to delete or modify the schema in the same way. That can only be done by directly modifying the file that holds the schema information.

#### 7-3-8-2. Of Data

To modify data, the user executes a query. Then the user can choose the query menu item "Delete All Results" and all the n-tuples from which the results were obtained will be deleted. If the user chooses, "Delete Results One by One", the user will be prompted about whether or not to delete each n-tuple.

Similarly, the user can execute a query and then choose the query menu item "Modify All Results". In that case, the user will be prompted for a modification and all the n-tuples from which the results were obtained will be modified in that way. If the user chooses, "Modify Results One by One", the user will be prompted for a modification for each n-tuple.

After a user selects a MacDRAW object, he enters the attribute desk accessory.

Key in a Type (Otherwise the Last Selected Type is Used)

Menu of Attributes

Menu of Values

OK Get Non-Atomic Value Cancel

The user gives the object a type at the top and then puts in attributes and values. As he puts in attributes and values, they are placed into the "Menu of Attributes" and "Menu of Values". Then, whenever the user enters the same type (in this case, door) at the top, the attributes and values already keyed in will be available for the user to select from, instead of re-typing them.

door

Menu of Attributes

manufacturer is

Menu of Values

Stanley

height is

10.5

OK Get Non-Atomic Value Cancel

The user can bring this information into the database by issuing the menu command "Get from MacDRAW" from the query menu in GRAF-ASQ.

Figure 6. Simple attributing.

After a user selects a MacDRAW object, he enters the attribute desk accessory and assigns attributes and values as desired (see Figure 6).

door

Menu of Attributes

manufacturer is

Menu of Values

Stanley

height is

10.5

OK Get Non-Atomic Value Cancel

The user can enter an attribute that receives a non-atomic value by typing in the attribute and then pressing the "Get Non-Atomic Value" button. The user is then brought back into MacDRAW, where he selects another object to be the value of the attribute.

door

Menu of Attributes

manufacturer is

height is

Menu of Values

Stanley

10.5

next to

OK **Get Non-Atomic Value** Cancel

When the user chooses the first object again in MacDRAW and then returns to the attribute desk accessory, he will notice that a description of the non-atomic value has been put in.

door

Menu of Attributes

manufacturer is

height is

next to

Menu of Values

Stanley

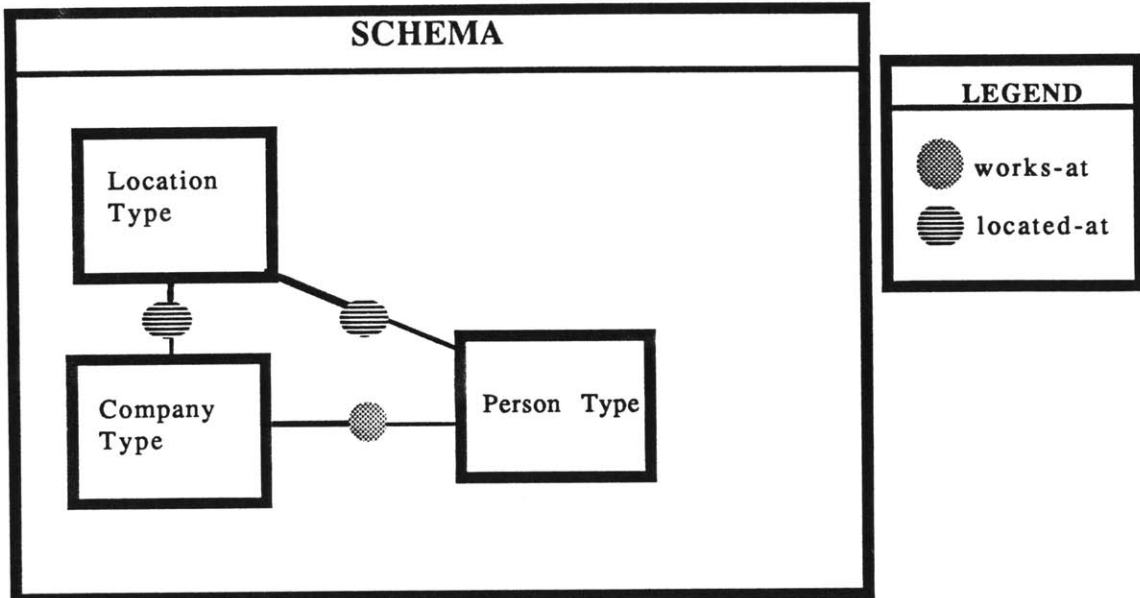
10.5

<object of type window>

OK Get Non-Atomic Value Cancel

Figure 7. Complex attributing.

Schema view with "person type" as the center, number of levels = 1, and include atomic off.



Schema view with "company type" as the center, number of levels = 2, and include atomic on.

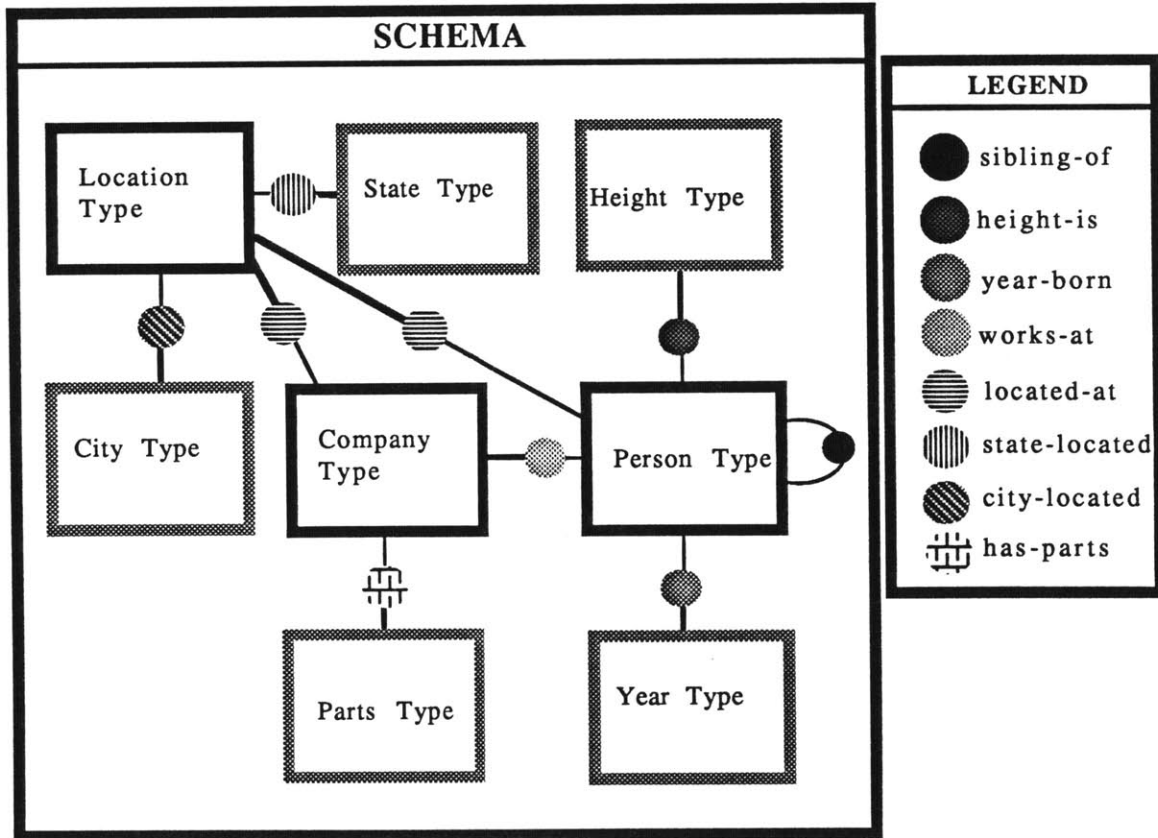
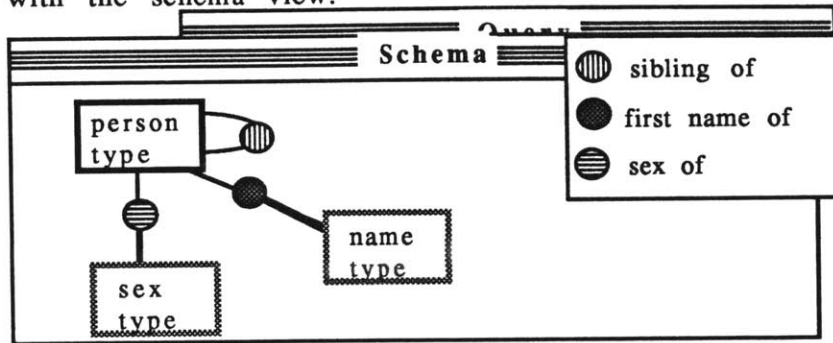


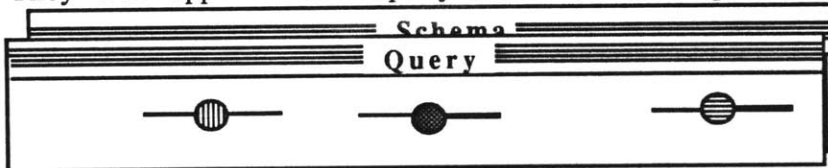
Figure 8. Schema Views



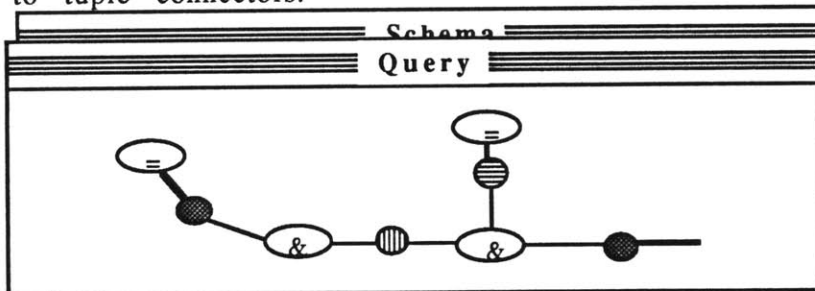
Choose attributes from schema via a menu that is created with the schema view.



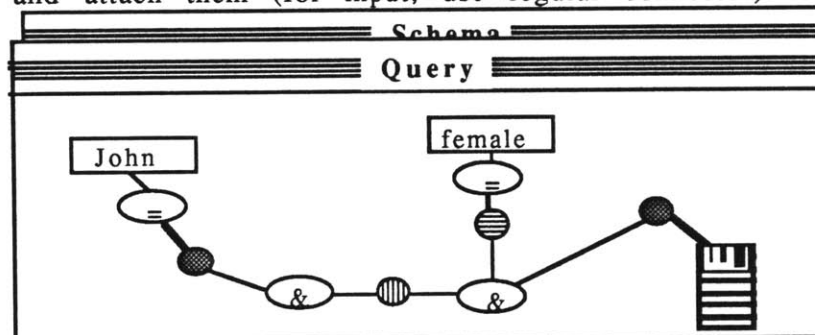
They will appear in the query window, with tuple connectors.



Choose operators from operators menu and attach them to tuple connectors.



Choose appropriate input/output from input/output menu and attach them (for input, use regular connector).



User may now execute query and have results appear in the list window.

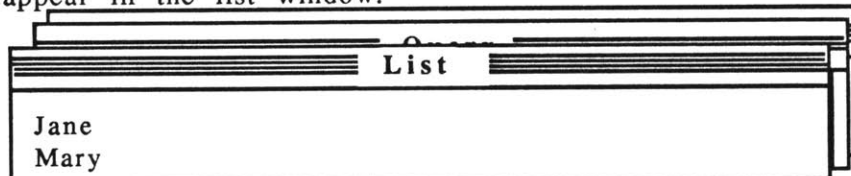
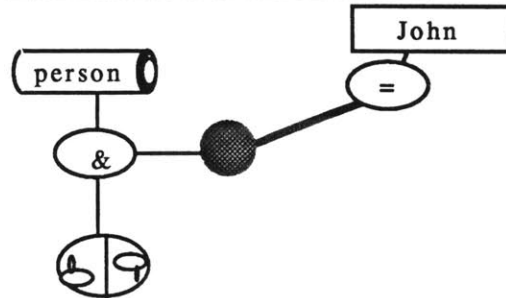
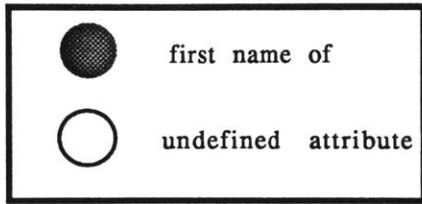
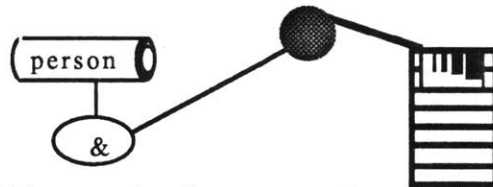


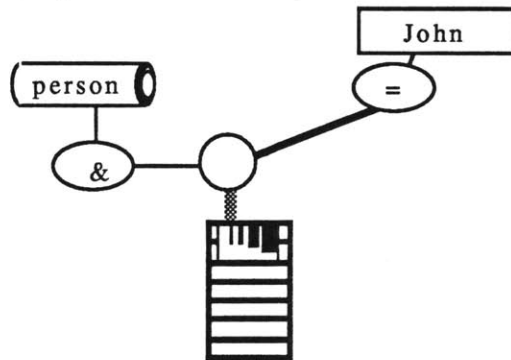
Figure 10. Steps to create the query:  
"Retrieve the first names of all John's sisters".



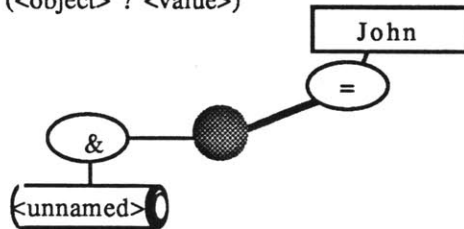
Does a person with the first name John exist?  
 (<object> <attribute> <value>)



What are the first names of people in the database?  
 (<object> <attribute> ?)

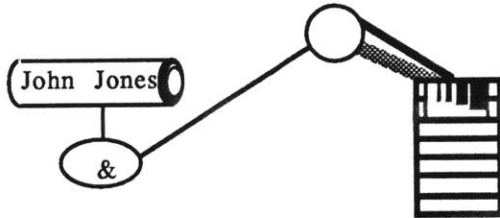
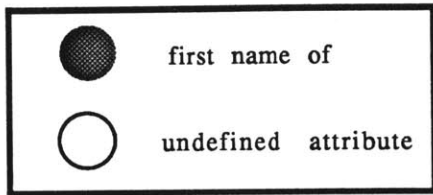


How are people related to the value "John"?  
 (<object> ? <value>)

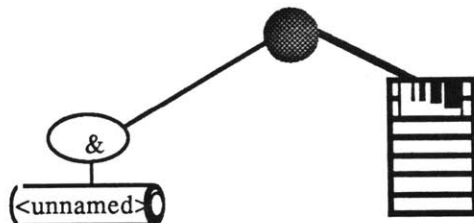


Create a domain of objects with a  
 first name of "John".  
 (? <attribute> <value>)

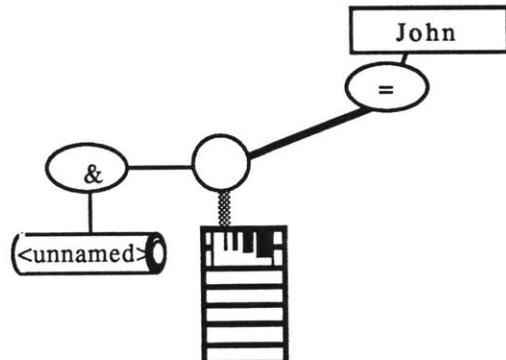
Figure 10. The first four of the eight permutations of straight 3-tuples.



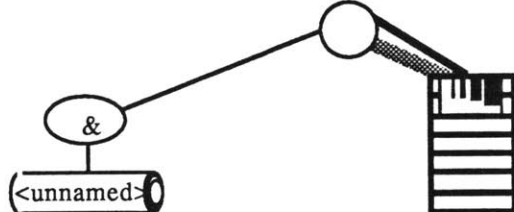
What are all the attributes and values John Jones?  
 (<object> ? ?)



What objects and values are related by the attribute "first name of"?  
 (? <attribute> ? ?)



What objects with what attributes have the value "John"?  
 ( ? ? <value> )



What are all the objects and all attributes and values in the database?  
 ( ? ? ? )

Figure 11. The second four of the eight permutations of straight 3-tuples.

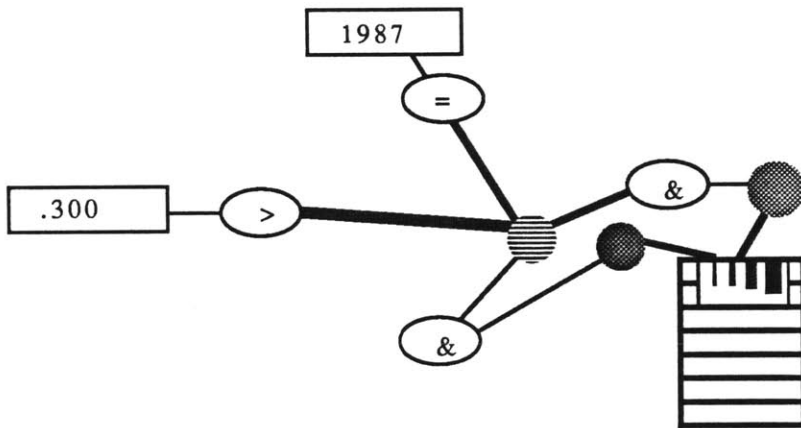
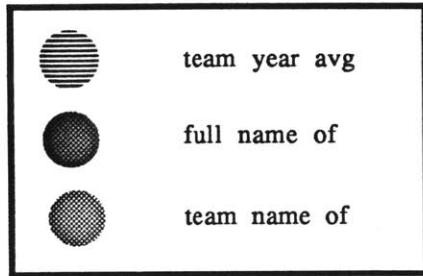
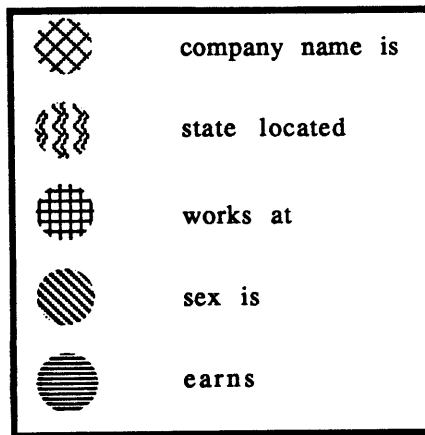


Figure 12. 5-tuple query:  
 "Retrieve the full names of all players with an average over .300 and the names of the teams they played on in the year 1987".



Note, this would not count as an "or" in the definition of a complex query as a query that has at least one "and" and at least one "or".

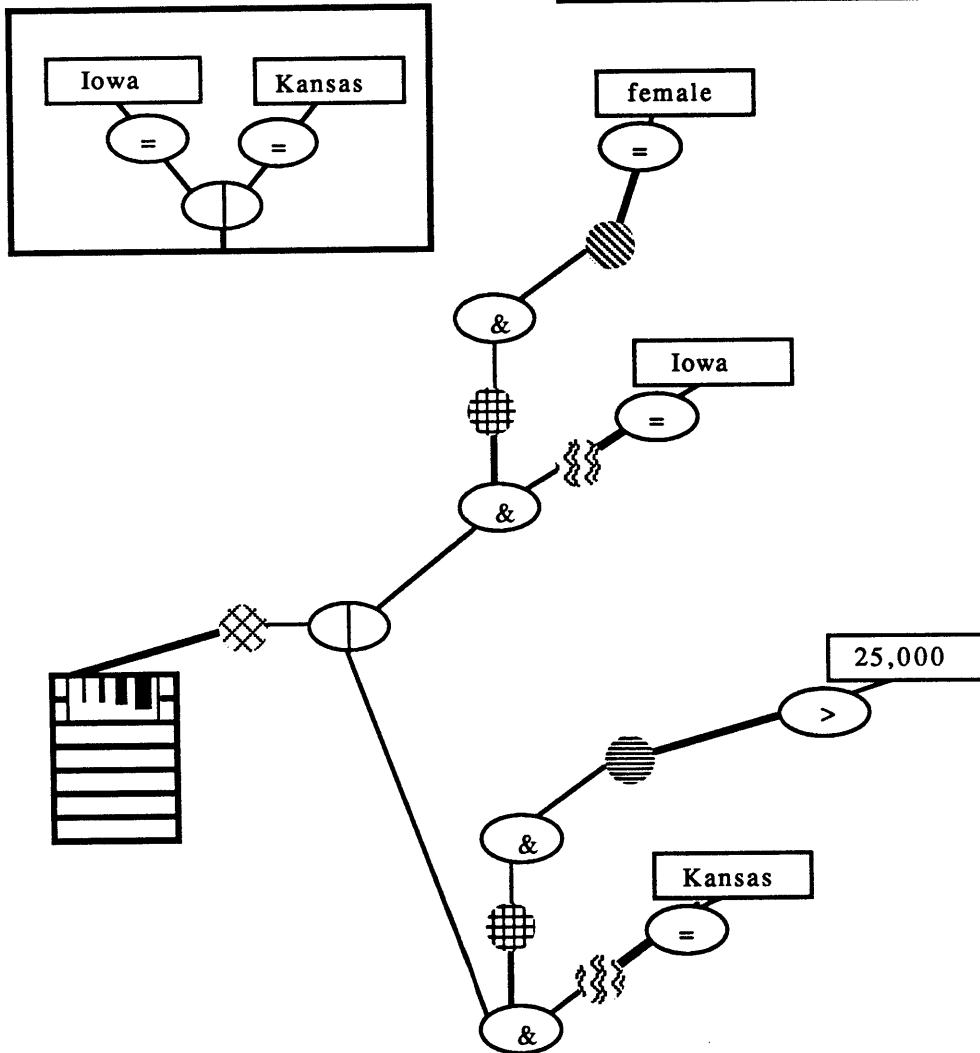
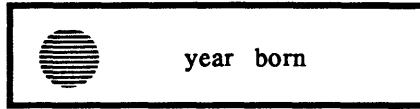
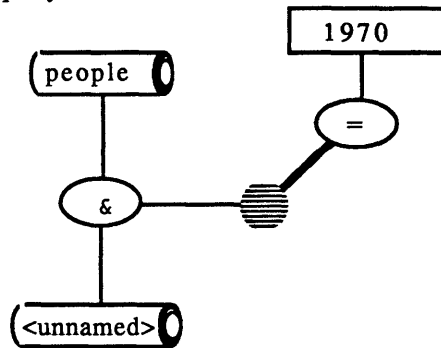


Figure 13. Complex query:  
 "Retrieve the names of all companies which are:  
 located in Iowa and have female employees  
 OR  
 located in Kansas and have employees that  
 earn over \$25,000".



Create by the same methodology as a query, as this a special type of query.



Execute the query via the menu item "Execute Query".  
Store and name the domain via the menu item "Save Domain".

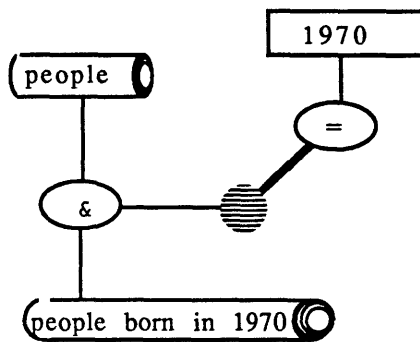
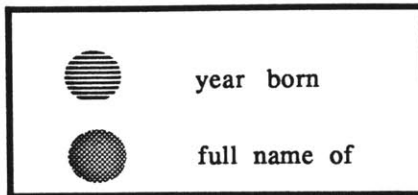
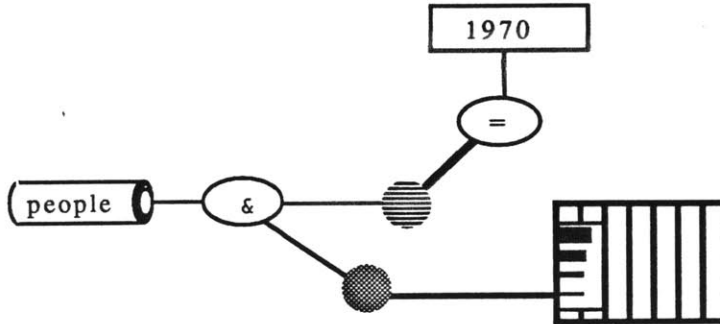


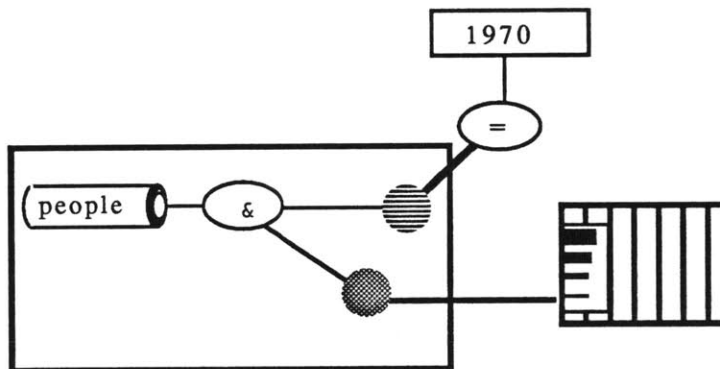
Figure 14. Creating the domain "People born in 1970".



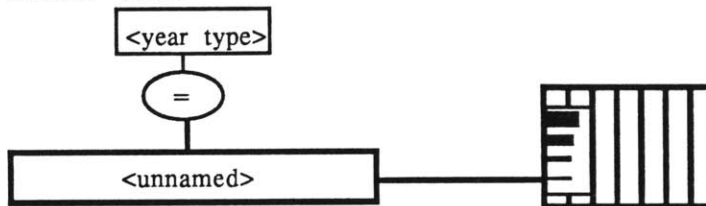
Create a query.  
 (below is "Retrieve the full names of people born in 1970").



Choose "Function Box" from the functions menu and draw a function box around non-input/output.



Choose the menu item "Reduce Function" to put it in reduced form.



Choose the menu item "Save Function" to name and save the function.

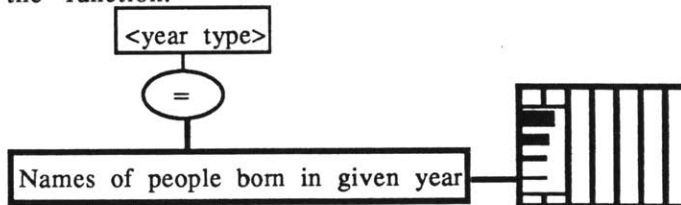
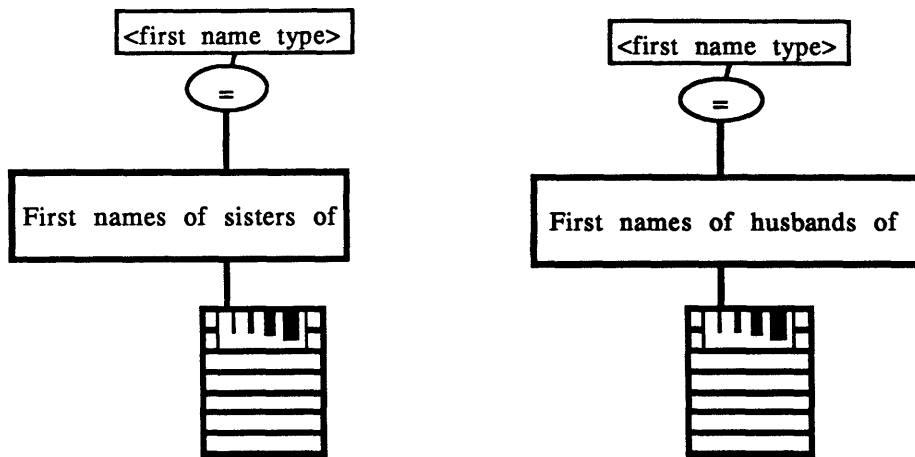
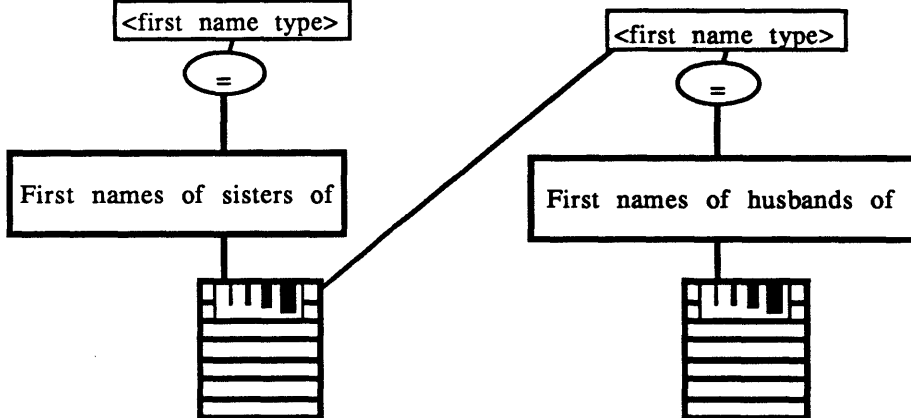


Figure 15. Creating the function "Names of people born in given year".

Choose two functions.



Connect the output of one with the input of another with a regular connector.



Choose the menu item "Link Functions" from the functions menu to get the following:

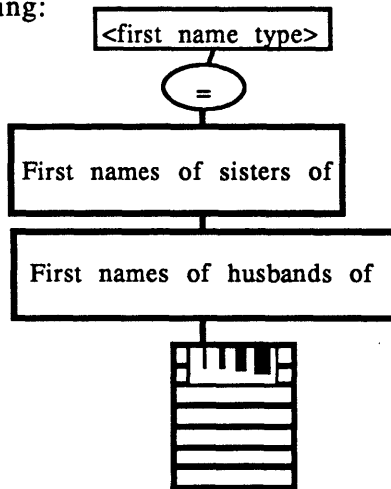
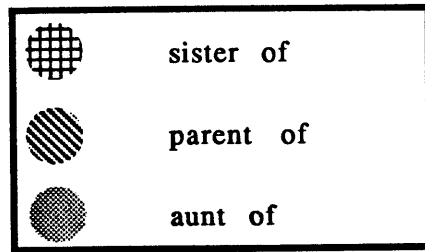


Figure 16. Linking Functions





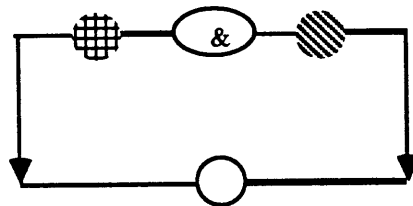
User creates a query without input/output.



User selects "Undefined Attribute" from the attributes menu.



User connects the two using the define-functional-attribute connector.



User chooses "Save Attribute" from the attributes menu and is prompted for a name. After the functional attribute is named, its symbol is assigned a pattern.

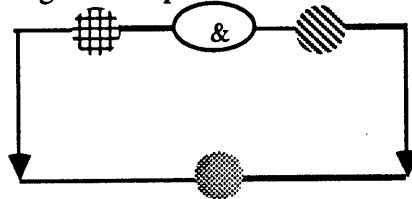
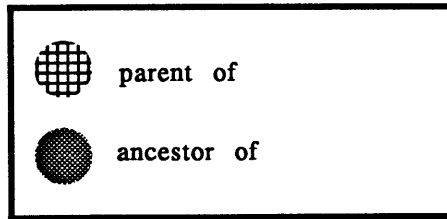
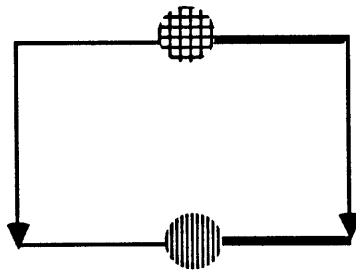


Figure 17. Creating the functional attribute "aunt of"



Create the terminating functional attribute  
(which translates to "parent of", in this case).



Create the recursive functional attribute  
(which translates to "parent of ancestor of",  
in this case).

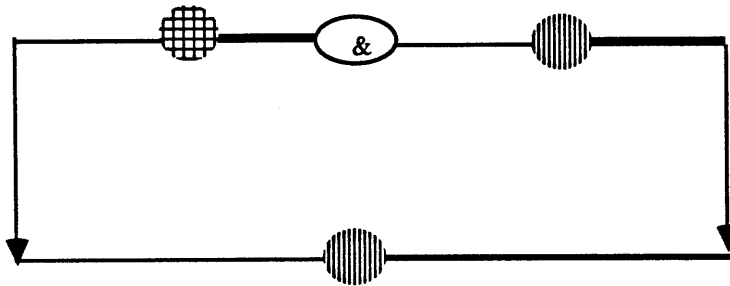


Figure 18. The recursive functional attribute  
"ancestor of"

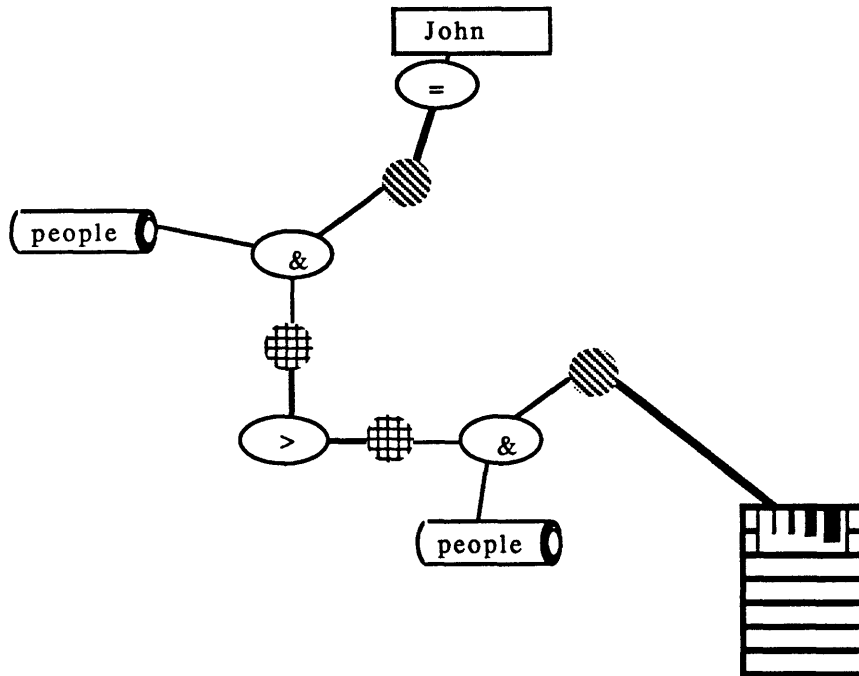
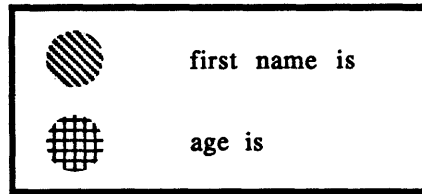
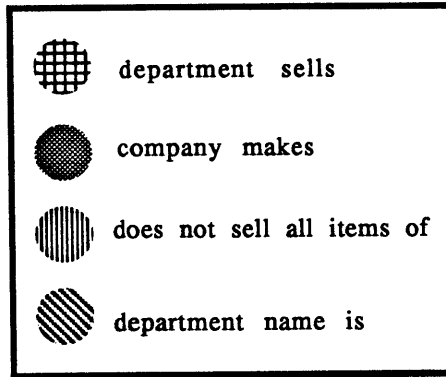
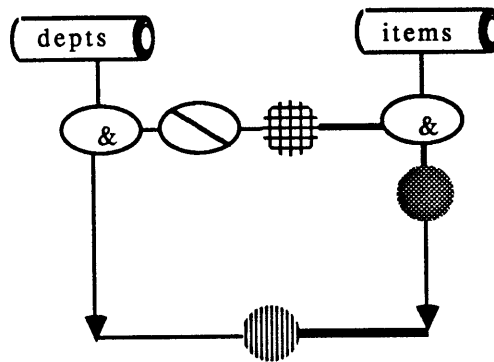


Figure 19. Comparison query:  
Retrieve names of all people older than John.



First, create the functional attribute "does not sell all items of".



Use this functional attribute to produce the desired query.

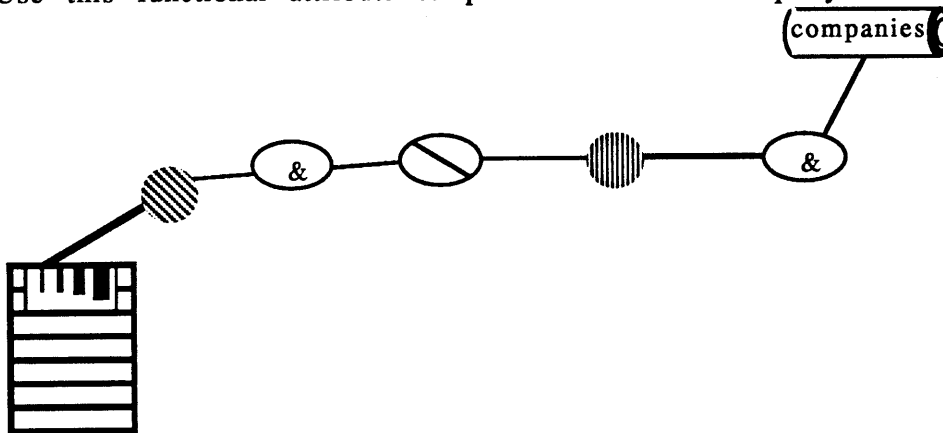


Figure 20. Relational Database Only Query:  
 "Retrieve the names of any departments that sell every item some company makes."

## 8. Conclusion

GRAF-ASQ is designed as a graphical query system that contains a limited number of symbols to provide simplicity and to provide the user with the ability to access data not normally accessible in entity-relationship databases yet in an entity-relationship form. It is also designed to interface with other systems, particularly systems like those for computer-aided design (CAD), which need, yet normally do not provide, extensive search abilities.

The implemented parts include:

- the ability to view the schema with different central foci and with atomic attributes made invisible.
- the ability to get simple attributed data from MacDRAW and to query and search for data via menus.
- the ability to highlight found objects within MacDRAW.

The usefulness of having the schema view has been seen in other languages (see Section 3). The ability to interface with MacDRAW is also useful. It allows a user to create graphical drawings in a familiar environment and to store and retrieve data corresponding to those drawings, using that same environment. The ability to query and search for items via menus gives a taste of the query capabilities the final graphical query language would contain.

It will take a full implementation to know the usefulness of the graphical query language itself. On paper, a variety of queries can be put together in a very simple, comprehensible form. The queries that are possible include those often atypical for an entity-relationship type database, yet the user can still think in terms of entities (objects) and relationships (attributes) when constructing the queries.

The potential of the function and functional attribute constructs were explained to some extent. These constructs relate to known constructs, such as functions and procedures, found in programming languages. They are not as often found in query languages, yet can add greatly to ease of use. They, among other things, allow a user to test pieces of queries as they go along and to store them so that the same queries will not have to be reproduced over and over again.

The big question is how quickly a naive user would take to this form. The simple graphics and the entity-relationship form are definite plusses, yet are not guarantees. GRAF-ASQ is definitely a new language and how quickly a naive user would catch on to it is unknown. Only testing with a fully implemented version would make this clear.

## Bibliography

### Graphical Database Interfaces

The following all use the Entity-Relationship model and let the user have a graphical view of the database structure. Each annotation will point out a key idea(s) brought out in the particular paper.

- [EL85] Elmasri, R.A., J.A. Larson,  
"A Graphical Query Facility for ER Databases",  
Proceedings of the Fourth International Conference on  
Entity-Relationship Approach, pp. 236-45,  
Oct. 1985, Chicago, IL,  
IEEE Comput. Soc. Press, Silver Spring, MD.
- This database interface (GORDAS) allows users to see a hierarchical view of the structure based on any entity the user selects.
- [FO84] Fogg, D.,  
"Lessons from a 'Living in a Database' Graphical Query",  
ACM SIGMOD, vol. 14, no. 2,  
Proceedings of the Annual Meeting, Jun. 1984.
- This database interface (LID) emphasizes browsing, starting from a particular entity and branching off to entities related to it.
- [GO85] Goldman, K.J., S.A. Goldman, P.C. Kanellakis, S.B. Zdonik,  
"ISIS: Interface for a Semantic Information System",  
SIGMOD Rec., vol. 14, no. 4, pp. 328-42, Dec. 1985, USA,  
Proceedings of the ACM-SIGMOD 1985 International Conference on  
Management of Data, pp. 28-31, May 1985, Austin, TX.
- This database interface provides a consistent way to view both the database structure and the data itself. It allows the results of a query to be added to the database structure in the form of a sub-class. A sub-class is actually a collection of the entities that fit the query criteria.
- [KI84] King, R.,  
"Sembase: A Semantic DBMS".  
Proceedings of the First International Workshop on Expert Database  
Systems, Kiawah Island, South Carolina, Oct. 1984.
- This database interface (SKI) shows the user the relationship between parts of the database that are not directly related.

- [WO82] Wong, H.K.T., I. Kuo,  
"GUIDE: Graphical User Interface for Database Exploration",  
Proceedings of Very Large Data Bases, Eighth International  
Conference, pp. 22-32, Sep. 1982, Mexico City, Mexico,  
VLDB Endowment, Saratoga, CA.

This database interface allows a user to examine the results of partial queries before creating the complete query. It also allows the user to view the database structure at various radii, and can get aggregate results of retrieved data such as sums and averages. However, it specifically handles statistical applications and only uses numeric data.

- [WO83] Wong, H.K.T., W.-L. Yeh,  
"Graphical Query Systems for Complex Statistical Databases",  
Computer Science and Statistics: Proceedings of the Fifteenth  
Symposium on the Interface, pp. 35-49, Mar. 1983, Houston, TX,  
North-Holland, Amsterdam, Netherlands.

More about GUIDE (see [WO82]).

- [ZD86] Davison, Jay W. and Stanley B. Zdonik,  
"A Visual Interface for a Database with Version Management",  
ACM Transactions on Office Information Systems, vol. 4, no. 3.,  
July 1986, pp. 226-256.

#### Other Related Database Facilities

- [CA80] Cattell, R.G.G.,  
"An Entity-based Database User Interface",  
Proceedings of the ACM SIGMOD Conference on Management of Data,  
May, 1980.

This paper speaks about SDB, and includes discussion of the advantages of entity-relationship databases.

- [HA81] Hammer, M., D. McLeod,  
"Database Description with SDM: A Semantic Database Model",  
ACM TODS vol. 6, no. 3, Sep. 1981, pp. 351-387.

This paper gives a detailed description of an entity-relationship model, and brings out two interesting constructs: a *grouping* (a super-type) and an *aggregate* (a subset of entities that the user groups together manually).



## Database Background

- [CO72] Codd, E.F.,  
"Relational Completeness of Data Base Sublanguages",  
Database Systems, ed. R. Rustin,  
Prentice Hall, NJ, 1972, pp. 65-98.
- This paper presents a methodology to determine the relational completeness of a language.
- [DA83] Date, C.J.,  
An Introduction to Database Systems, Vol. 2,  
Addison-Wesley Publishing Company, 1983.
- With volume 1, this is a good introductory text on databases.
- [DA86] Date, C.J.,  
An Introduction to Database Systems, Vol. 1, 4th ed.  
Addison-Wesley Publishing Company, 1986.
- With volume 2, this is a good introductory text on databases.
- [KE78] Kent, William,  
Data and Reality,  
North-Holland Publishing Co., 1978, Chapter 10.
- This paper contains an enlightening discussion of n-tuples.
- [SC84] Schiel, Ulrich,  
"A Semantic Data Model and its Mapping to an Internal Relational Model",  
Databases -- Roles and Structures, ed. P.M. Stocker, P.M.D. Gray,  
M.P. Atkinson,  
Cambridge University Press, 1984, pp. 373-400.
- This paper talks about the difference between models that distinguish between attributes and relationships and those that do not.

## Prolog

- [BR86] Bratko, I.,  
Prolog Programming for Artificial Intelligence,  
Addison-Wesley Publishing Co., Inc., 1986.

Prolog is described in the context of artificial intelligence applications. It gives a good, basic introduction to the language and how to think about programming with it.

- [CO86] Covington, M.A.,  
"Expressing Procedural Algorithms in Prolog",  
Research Report 01-0021,  
Advanced Computational Methods Center, Univ. of Georgia,  
May, 1986.

This paper gives helpful hints for procedural programming using Prolog.

- [LA86] Lanam, D.H.,  
Advanced A.I. Systems' Prolog Reference Manual,  
Version M-1.0 and 1.10,  
Advanced A.I. Systems, Inc., Mountain View, CA., 1986.

This is the manual for the Prolog being used for the thesis project.

- [MA86] Marcus, C.  
Prolog Programming: Applications for Database Systems, Expert Systems, and Natural Language Systems,  
Arity Corporation,  
Addison-Wesley Publishing Company, Inc., 1986.

## MacIntosh

- [AP85] Apple Computer, Inc.  
Inside MacIntosh, Vol. 1,  
Addison-Wesley Publishing Company, Inc., 1985.

This manual gives information about MacIntosh facilities needed for the thesis project, such as QuickDraw, the ToolBox and the windowing facilities.

- [CH85] Chernicoff, S.  
MacIntosh Revealed, Vol. 1 -- Unlocking the Toolbox  
Hayden Book Company, 1985.

This manual was used as a supplement to Inside MacIntosh. Especially useful is its detailed listing of traps.

- [Ch85] Chernicoff, S.  
MacIntosh Revealed, Vol. 2 -- Programming with the Toolbox  
Hayden Book Company, 1985.

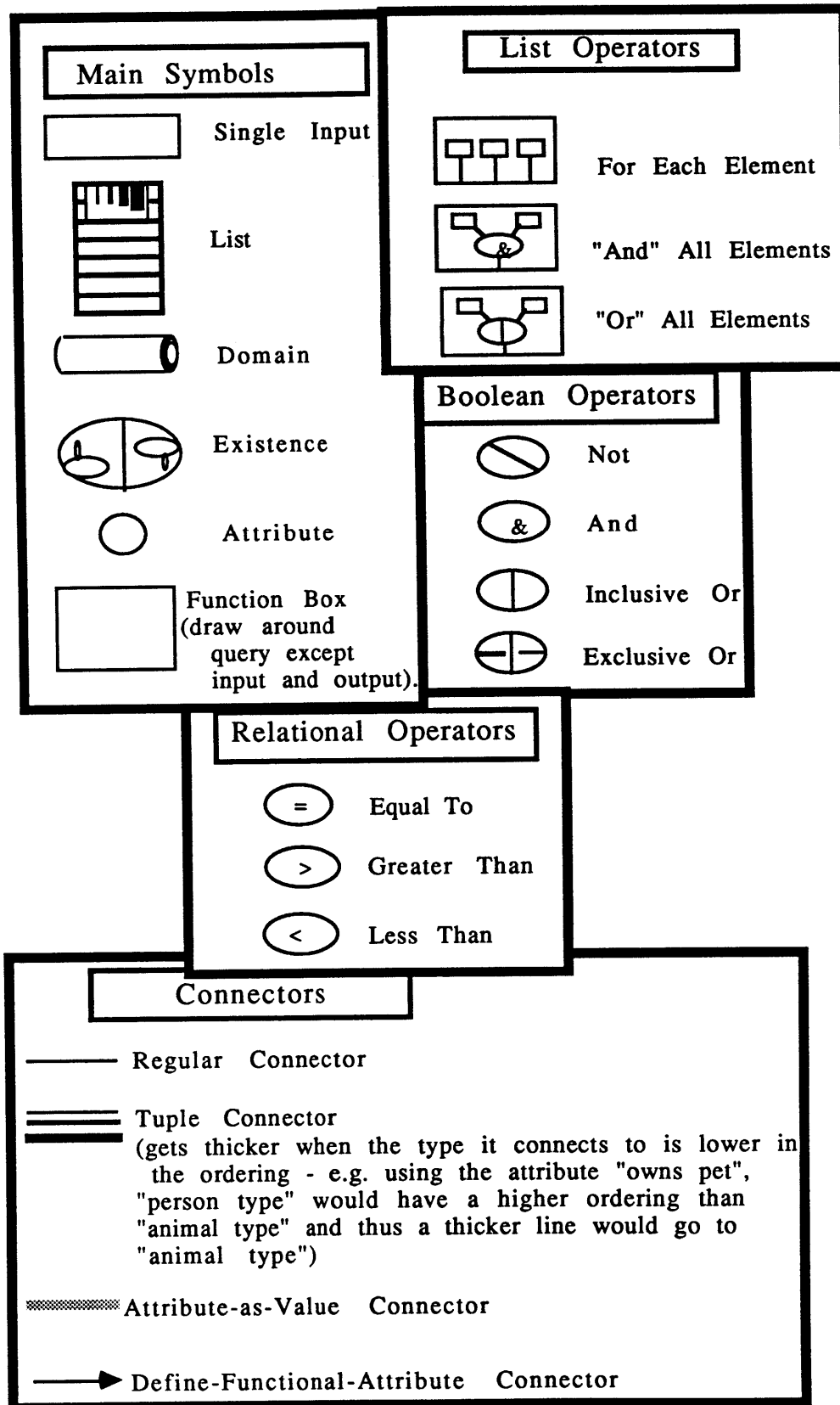
This manual was used as a supplement to Inside MacIntosh. Especially useful is its detailed listing of traps.

## AutoCAD

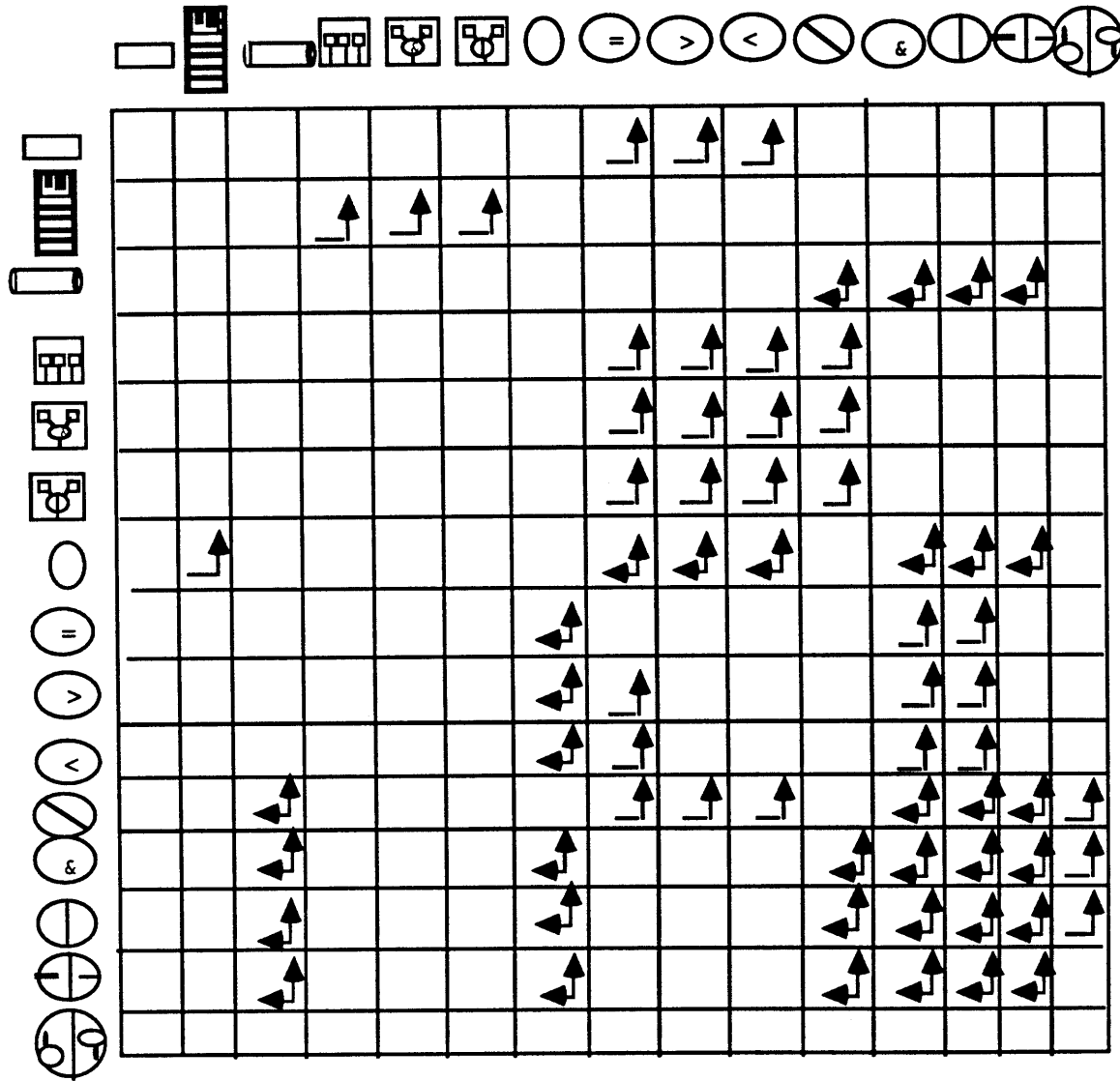
- [AU87] The AutoCAD Drafting Package Reference Manual,  
Publication TD106-009,  
Autodesk, Inc., 1987.

This manual was used as the information source on attributing using AutoCAD.

# Appendix I. Symbol List



## Appendix II. Legal Connections



**Key**

- Connect in one direction  
 - Connect in both directions

## Appendix III. Syntax

Key	
	Or
[]	Optional
{}	One or More
()	Connected Symbols

---

**Action**            :: Query | Create-Domain | Func-Attr-Def | Func-Def

---

**Symbol**            :: any piece of syntax that can be expressed  
                      as a single graphic item, string, or number



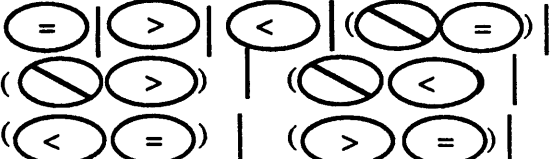





---

The following pages contain the syntax, divided as follows:

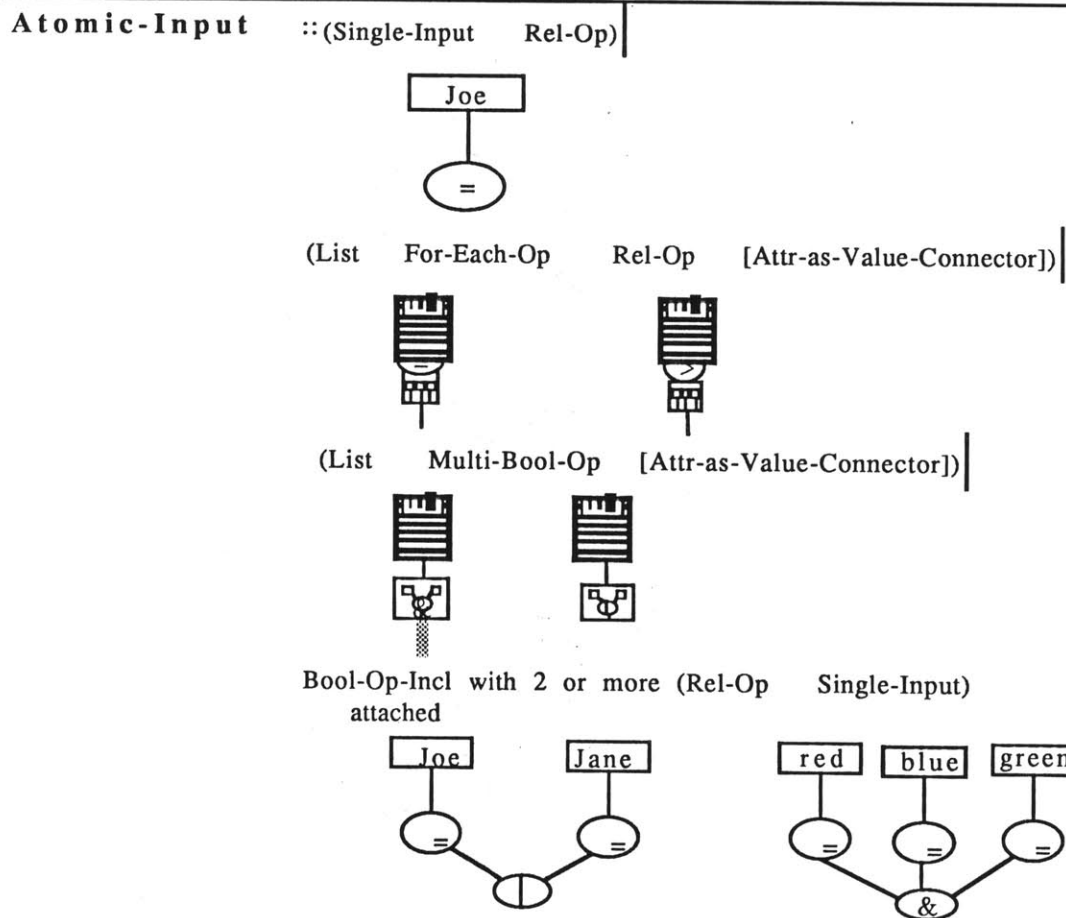
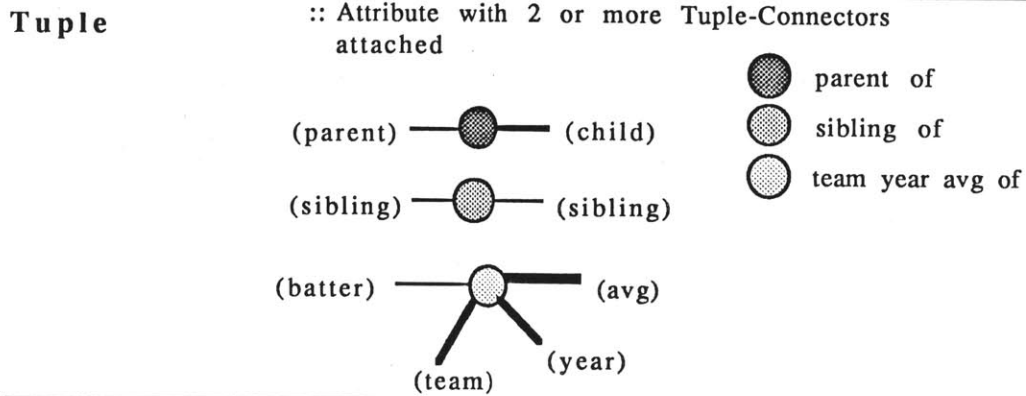
**Lowest Level Syntax - Symbols**  
**Lower Level Syntax - Tuples and Input/Output**  
**Higher Level Syntax - Object Groups**  
**Highest Level Syntax - Actions**

**Note:** All symbols are connected by Regular-Connectors, unless a Special-Connector is explicitly specified (see Lowest Level Syntax page for syntax definitions of Connectors).

### Appendix III. Lowest Level Syntax - Symbols

<b>&lt;Atomic Value&gt;</b>	::	<value> which never serves as <object> (see chapter entitled "N-Tuples") is either character string or number
<b>Single-Input</b>	::	<div style="border: 1px solid black; padding: 2px; display: inline-block;">&lt;Atomic Value&gt;</div>
<b>Multiple Boolean Operator (Multi-Bool-Op)</b>	::	
<b>For-Each Operator (For-Each-Op)</b>	::	
<b>Relational Operator (Rel-Op)</b>	::	
<b>Boolean Operator (with Inclusive Or only) (Bool-Op-Incl)</b>	::	
<b>Boolean Operator (Bool-Op)</b>	::	Bool-Op-Incl   
<b>Not-Op</b>	::	
<b>Regular-Connector</b>	::	—————
<b>Tuple-Connector</b>	::	—————   —————   ————— etc.
<b>Attribute as Value Connector (Attr-as-Value-Connector)</b>	::	—————
<b>Define Functional Attribute Connector (Def-Func-Attr-Connector)</b>	::	↓
<b>Special-Connector</b>	::	Tuple-Connector   Attribute-as-Value Connector   Def-Func-Rel Connector
<b>Attribute</b>	::	○
<b>Input-Domain</b>	::	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Named ○</div> (contains objects)
<b>Output-Domain</b>	::	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Unnamed ○</div> (empty until search done)
<b>List</b>	::	
<b>Existence</b>	::	

### Appendix III. Lower-level Syntax - Tuples and Input/Output



**Non-Atomic-Output** :: Output-Domain | Existence

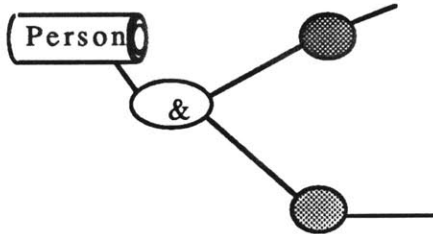
**Atomic-Output** :: List | (Attr-as-Value-Connector List)



### Appendix III. Higher-Level Syntax - Object Groups

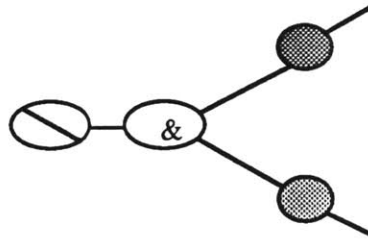
---

**Positive Object Group**:: Tuple |  
**(Pos-Obj-Group)** Bool-Op with optional Input Domain and 2 or more  
 Tuples attached



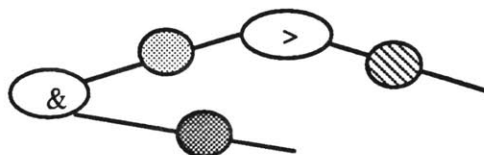
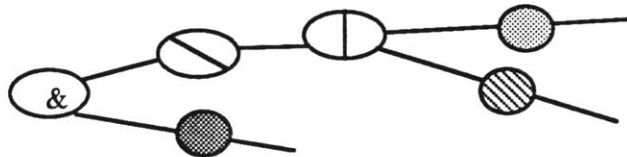

---

**Object Group** :: Pos-Obj-Group |  
**(Obj-Group)** (Not-Op Pos-Obj-Group)



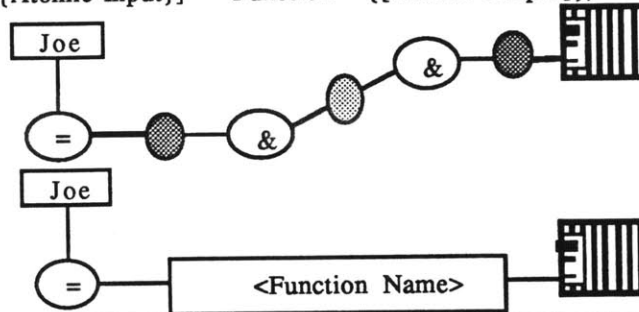

---

**Multiple Object Group**:: Obj-Group |  
**(Multi-Obj-Group)** Bool-Op with 2 or more Obj-Groups attached |  
 Rel-Op with 2 or more Obj-Groups attached

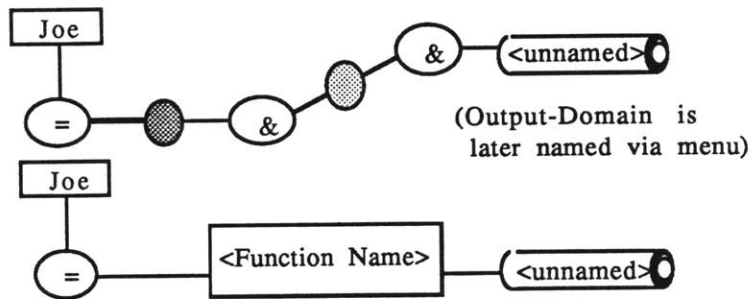


### Appendix III. Highest-Level Syntax - Actions

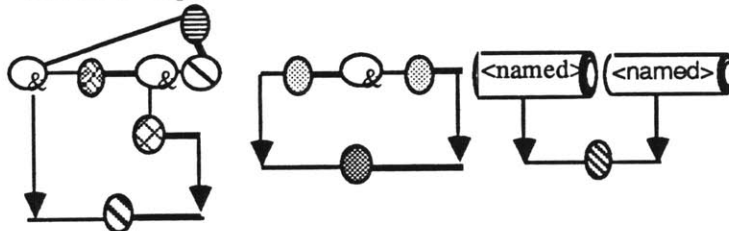
**Query** :: ([[Atomic-Input connected to Tuple-Connector of <Atomic Value>]]  
Multi-Obj-Group  
[[Atomic-Output connected to Tuple-Connector of <Atomic Value>]] ) |  
([[Atomic-Input]] Function [[Atomic-Output]])



**Create-Domain**:: ([[Atomic-Input connected to Tuple-Connector of <Atomic Value>]]  
Multi-Obj-Group  
[[Output-Domain connected to Bool-Op]] |  
([[Atomic-Input]] Function Output-Domain)



**Functional Attribute Definition (Func-Attr-Def)** :: Multi-Obj-Group with Def-Func-Attr-Connectors attached to each stray Tuple-Connector or Bool-Op | Domain with Def-Func-Attr-Connector pointing to each end of a Tuple



**Function Definition (Func-Def)** :: Multi-Obj-Group reduces to <Function Name>

which can be called as the function and expanded again via menu

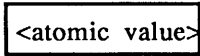


## **Appendix IV. Semantics of Symbols**

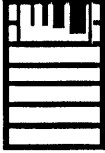
The following pages contain the semantics of the symbols, divided as follows:

- Semantics - Main Symbols and Connectors**
- Semantics - Operators**

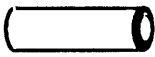
## Appendix IV. Semantics - Main Symbols

**Single Input:**

used to input a single atomic value into the query.

**List:**

a list of sets of atomic values retrieved from the query, each set containing one or more atomic values (e.g. first and last names referring to the same person would be in the same set). The box containing the lines of different thicknesses is where the incoming tuple connectors come into, depicting ordering, the narrower the line, the higher in the ordering. Thus, if the ordering was to be first name, then last name, the tuple connector for "first name of" would connect to the narrowest line and that for "last name of" would go to the second narrowest line.

**Domain:**

collection of objects, usually with some common trait, such as being of the same type and/or having a common attribute value.

**Existence:**

query output which indicates whether or not what was searched for exists.

**Attribute:**

the attribute of an n-tuple.

**Function Box:**

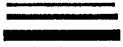
this is drawn around a query, except its input and output. Then, the query is reduced to a function box that has the input and output coming off of it. This is now known as a function which can be named, stored, and executed like the original query. It can also be connected to other functions and to queries to create more complex queries.

## Appendix IV. Semantics - Connectors



### **Regular Connector:**

the connector most commonly used to connect non-connector symbols.



### **Tuple Connector:**

the connector that is an off-shoot to an attribute. It is used to indicated the ordering of the other items in the n-tuple. The lighter the line, the higher the ordering. The object of the n-tuple will always connect to the lightest line. However, the value of the n-tuple could be equally light if there is no ordering. For example, if the attribute is "sibling is", the ordering is equal, because the either the object or the value could be in the object position. If the attribute is "parent of", however, only the object and value would have to have the given order and the value would be connected by a darker line.



### **Attribute-as-Value Connector:**

in the case where there is a query to find out a list of attributes this connector is connected from an attribute to a list symbol to avoid confusion with a tuple connector, which also connects to an attribute.



### **Define-Functional-Attribute Connector:**

this is used to connect a query with a new attribute which will define the new attribute as a functional attribute, to be calculated based on the query.

## Appendix IV. Semantics - Operators

### Relational Operators

**Equal To:**

determines whether or not the input value is equal to the corresponding n-tuple value.

**Greater Than:**

determines whether or not the input value is greater than the corresponding n-tuple value.

**Less Than:**

determines whether or not the input value is less than the corresponding n-tuple value.

### Boolean Operators

**Not:**

negates the results of any of the other boolean operators, or any of the relational operators.

**And:**

determines whether or not all the object-groups (see Appendix III) or input values connected to it are true.

**Inclusive Or:**

determines whether or not at least one of the object-groups (see Appendix III) or input values connected to it are true.

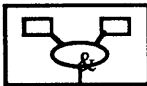
**Exclusive Or:**

determines whether or not only one of the object-groups (see Appendix III) or input values connected to it are true. The lighter line within the symbol would be connected to the object-group that should be tested first. If more than two object-groups were connected to the exclusive or, more lines would appear.

### List Operators

**For Each Element:**

treats each element of a list as an individual input value for the query.

**"And" All Elements:**

"ands" together all the elements of a list as if each element were a separate input value being "anded" together with the others.

**"Or" All Elements:**

"inclusive-ors" together all the elements of a list as if each element were a separate input value being "inclusive-ored" together with the others.

## Appendix V. Menues

---

### Main Menu

Schema	Query	Operators	Input/Output	Connectors	Attributes	Functions
--------	-------	-----------	--------------	------------	------------	-----------

---

### Schema Menu Items

**Display Schema:**

Schema will be displayed in the Schema Window with the specifications as indicated by the other menu items.

**Choose Center:**

A menu of object types will appear and one can be chosen as the center. The current center will have a check mark beside it.

**Include Atomic:**

If check mark beside it, indicates that atomic value types should be displayed on the schema display. Otherwise, they will be invisible.

**Choose Number of Levels:**

User will be able to indicate the number of levels from the center object that should be displayed on the schema diagram.

---

### Query Menu Items

**Create Query:**

Opens up window for query creation.

**Execute Query:**

Once the query has been constructed, choose this to actually perform the query.

**Delete All Results:**

Once the query has executed, choose this to delete all the items chosen from the database.

**Delete Results One by One:**

Once the query has executed, choose this to walk through each item selected and decide individually whether or not to delete it from the database.

**Modify All Results:**

Once the query has executed, choose this to modify all items chosen from the database. The user will be prompted for the modification.

**Modify Results One by One:**

Once the query has executed, choose this to walk through each item selected and decide individually whether or not to modify it. The user will be prompted for the modification.

**Get from MacDRAW:**

Put the latest n-tuples created through the Attribute Desk Accessory in MacDRAW into the database.

**Show in MacDRAW:**

Highlight any MacDRAW objects selected via the most recent query when enter MacDRAW.

---

### Operators Menu Items

The following operators will be in the menu for the user to choose from and user for a query:

 equal to  less than  greater than

 not  and  exclusive or  inclusive or

---

## Appendix V. Menues Continued

---

### Input/Output Menu Items:

User can select an empty input/output or select one already saved as part of creating a query.



empty input value



empty list



empty domain



existence

#### Save List:

User will be prompted for name for domain and then it will be saved.

#### Select List:

User will be prompted to select a list from the ones saved.

#### Save Domain:

User will be prompted for name for list and then it will be saved.

#### Select Domain:

User will be prompted to select a domain from the ones saved.

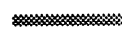
---

### Connectors Menu Items:

User can choose a connector to connect other symbols or modify tuple connector:



regular connector



attribute-as-value connector



define-functional-attribute connector

#### Modify tuple connector:

Lighten or darken the chosen tuple connector to change the ordering.

---

### Attributes Menu Items:

User can choose undefined attributes to use in a query, or can choose one of the other items.



undefined attribute

#### Save attribute:

User will be prompted for a name for the attribute and then it will be saved.

#### Select attribute:

User will be prompted to select an attribute from all that exist.

#### Select attribute from Schema View:

User will be prompted to select an attribute from all that exist within the current schema view.

#### Modify attribute:

User will be prompted to change name of selected attribute. If the attribute is a functional attribute, the user will be able to modify the query associated with it.

#### Delete attribute:

All n-tuples that contain the currently selected attribute will be deleted.

#### Functional to Database:

The selected functional attribute will calculate all its members and enter them into the database.

---



## Appendix V. Menues Continued

---

### Functions Menu Items:

User can choose function box in order to draw one around a query, or can choose one of the other items.



function box

### Reduce Function:

Reduces the query, surrounded by a function box, to a function box with the input and output attached to it.

### Save Function:

User will be prompted for name for the current function and then the function will be saved.

### Select Function:

User will be prompted to select a function from the ones saved.

### Delete Function:

Deletes the current function.

### Modify Function:

Expands the current function into its original query form and allows the user to modify the current function.

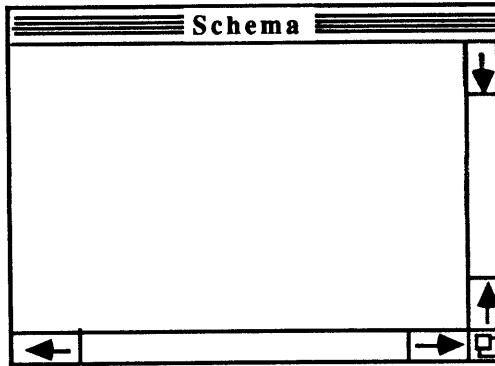
### Link Functions:

Two functions are chosen by the user and then the output from one is connected to the input of the other via a regular connector. The user then chooses "Link Functions" to link them together.

---

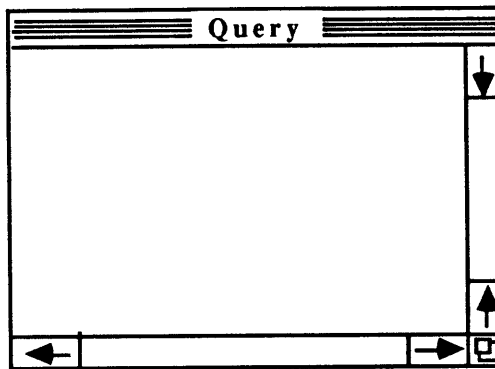
## Appendix VI. Windows

---



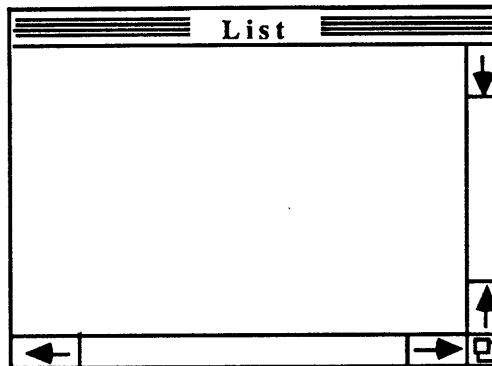
Window within which the schema is shown.

---



Window within which the user can create and execute a query (including functions).

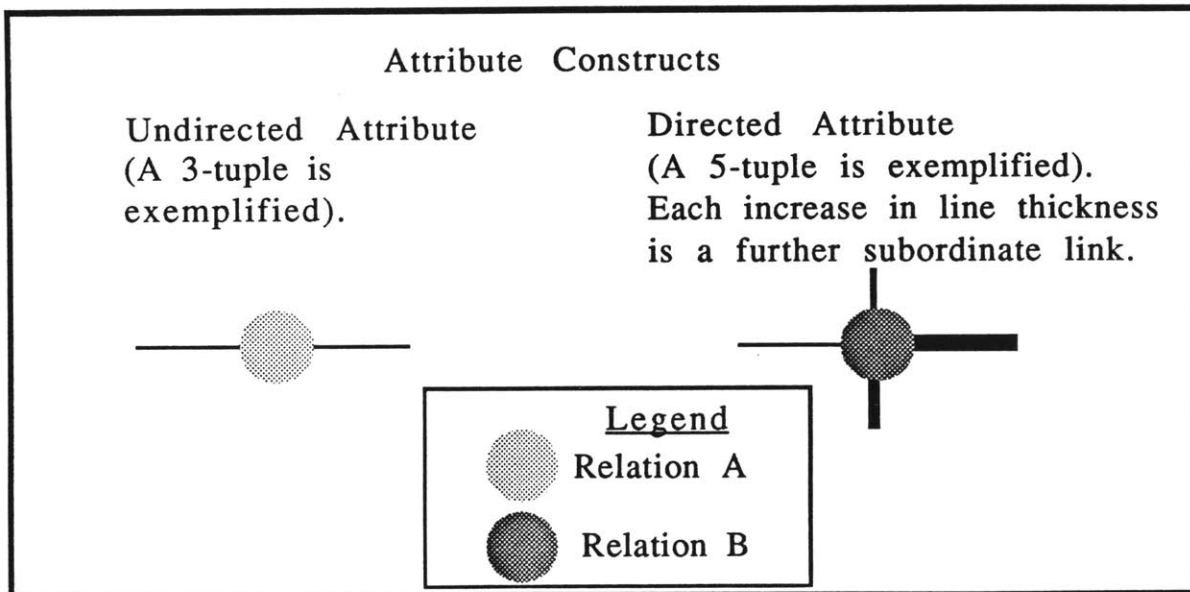
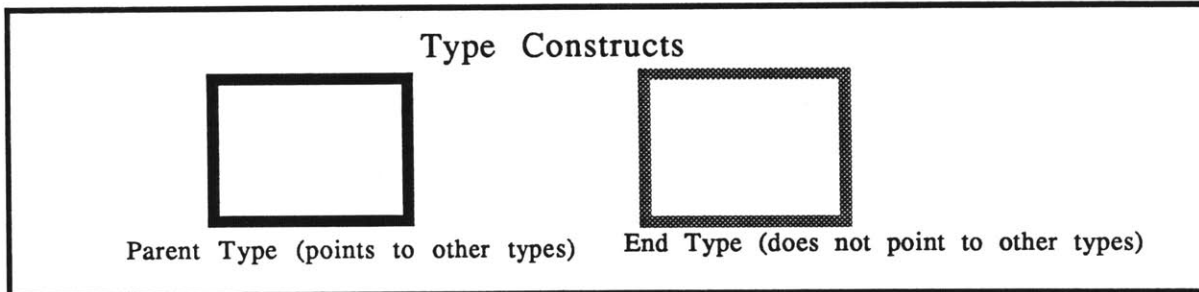
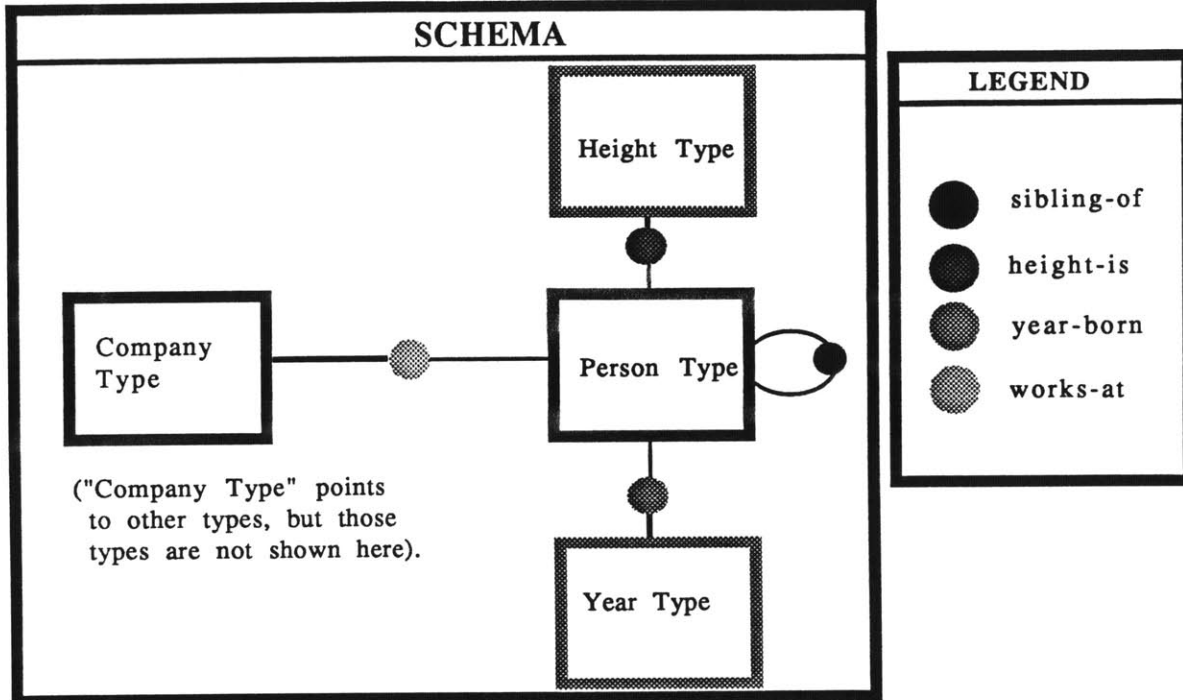
---



Window within which the list output is shown.

---

## Appendix VII. Schema



## Appendix VIII. Attribute Desk Accessory

The user chooses an object from MacDRAW, then gets into the Attribute Desk Accessory. When in the desk accessory, he keys in the object's type (if the type has already been keyed in, the type is already shown). The user can then assign attributes and values by choosing from the old ones in the above scrolling region, or type in new ones below.

Key in a Type (Otherwise the Last Selected Type is Used)

<Type is put in here>

Menu of Attributes	Menu of Values
<p style="text-align: center;">&lt;Old attribute is shown here&gt;</p> <div style="border: 1px solid black; height: 20px; width: 100%;"></div> <div style="border: 1px solid black; height: 20px; width: 100%;"></div> <div style="border: 1px solid black; height: 20px; width: 100%;"></div>	<p style="text-align: center;">&lt;Old value is shown here&gt;</p> <div style="border: 1px solid black; height: 20px; width: 100%;"></div> <div style="border: 1px solid black; height: 20px; width: 100%;"></div> <div style="border: 1px solid black; height: 20px; width: 100%;"></div>
<p style="text-align: center;">&lt;New attribute is put in here&gt;</p> <div style="border: 1px solid black; height: 100px; width: 100%;"></div>	<p style="text-align: center;">&lt;New value is put here&gt;</p> <div style="border: 1px solid black; height: 100px; width: 100%;"></div>

OK

Get Non-Atomic Value

Cancel

The user can assign a non-atomic value by assigning an attribute and not assigning a value for it, but press the "Get Non-Atomic Value" button instead. Then the user will be brought back to MacDRAW, where he can choose the non-atomic value by choosing another object.