(2)

# NAVAL POSTGRADUATE SCHOOL
## Monterey, California



# THESIS

**USER INTERFACE OF DFQL:
AN OBJECT-ORIENTED APPROACH**

by

Li, Chang-Tsun

May 1992

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited.

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S) | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS | 7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School |
| 6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000 | 7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c. ADDRESS (City, State, and ZIP Code) | 10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)
USER INTERFACE OF DFQL: AN OBJECT_ORIENTED APPROACH

12. PERSONAL AUTHOR(S)
Li, Chang-Tsun

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| Master's Thesis | FROM 02/90 TO 06/92 | June 1992 | 181 |

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | query language, dataflow programming, object-oriented programming, human factors |
| | | | |
| | | | |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In recent years, many graphical approaches have been proposed to lift the inconvenience of text-based query language among end-users. The new query language called DFQL (DataFlow Query Language) is a fully graphical interface to the relational model based on a dataflow paradigm. It only requires users to connect some well-defined operators which have been given an equivalent one-to-one correspondent functionality (or construct) in traditional query language (SQL in this case). All of the power of current query languages and sufficient expressive power and functionality are retained. But some shortcomings of DFQL user interface still exist.

This thesis is to introduce a more ease-to-use and ease-to-learn user interface, so the shortcomings we found in DFQL user interface can be lifted and the productivity and power of the new version of DFQL can be increased. We have adopted object-oriented programming approach in our implementation and the benefits of using object-oriented programming in our development are also discussed.

| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT | 21. ABSTRACT SECURITY CLASSIFICATION | |
|---|---|---|
| [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS | UNCLASSIFIED | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL: C. Thomas Wu | 22b. TELEPHONE (Include Area Code): (408) 655-5687 | 22c. OFFICE SYMBOL: CS/XX |

Approved for public release; distribution is unlimited

# USER INTERFACE OF DFQL: AN OBJECT-ORIENTED APPROACH

by
Li, Chang-Tsun
Captain, Taiwan, R.O.C. Army
Chung Cheng Institute of Technology, Taiwan, R.O.C., 1987

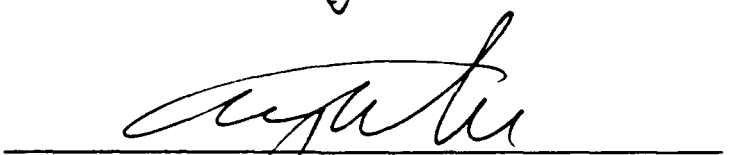
Submitted in partial fulfillment of the
requirements for the degree of

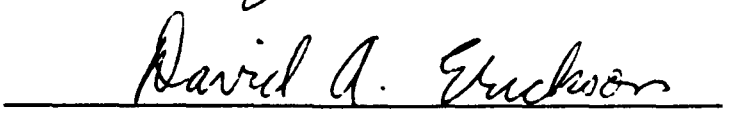
MASTER OF COMPUTER SCIENCE

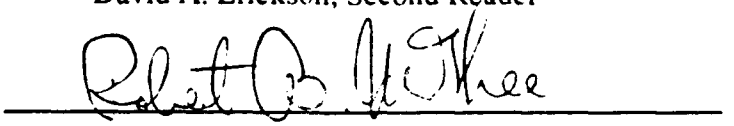
from the

NAVAL POSTGRADUATE SCHOOL
June 1992

Author: 
Li, Chang-Tsun

Approved By: 
C. Thomas Wu, Thesis Advisor


David A. Erickson, Second Reader

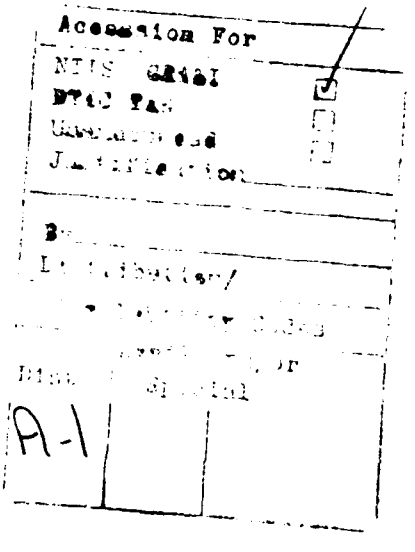

Robert B. McGhee, Chairman,
Department of Computer Science

# ABSTRACT

In recent years, many graphical approaches have been proposed to lift the inconvenience of text-based query language among end-users. The new query language called DFQL (DataFlow Query Language) is a fully graphical interface to the relational model based on a dataflow paradigm. It only requires users to connect some well-defined operators which have been given an equivalent one-to-one correspondent functionality (or construct) in traditional query language (SQL in this case). All of the power of current query languages and sufficient expressive power and functionality are retained. But some shortcomings of DFQL user interface still exist.

This thesis is to introduce a more ease-to-use and ease-to-learn user interface, so the shortcomings we found in DFQL user interface can be lifted and the productivity and power of the new version of DFQL can be increased. We have adopted object-oriented programming approach in our implementation and the benefits of using object-oriented programming in our development are also discussed.





# TABLE OF CONTENTS

# LIST OF FIGURES

# I. INTRODUCTION

## A. BACKGROUND

In the past twenty years, relational database management systems have been accepted extensively for database implementation. To interact with a database, users of a database management system such as Ingres develop application programs by embedding a data manipulation language(e.g., ESQL) in a regular programming language(e.g., C). But those specialized text-based languages are somewhat unfriendly to inexperienced end-users (Codd, 1988) (Codd, 1990, chpt.23). In recent years, many graphical approaches have been introduced to lift the inconvenience among end-users (Wu, 1991) (Wu, 1986) (Wong, 1982) (Zloof, 1977) (Miyao, 1986).

A new query language called DFQL (DataFlow Query Language) was proposed in (Wu, 1991). DFQL is a fully graphical interface to the relational model based on a dataflow paradigm. Instead requiring users to use traditional text-based query language, DFQL only require users to connect some well-defined operators which have been given an equivalent one-to-one correspondent functionality (or construct) in traditional query language (SQL in this case). All of the power of current query languages and sufficient expressive power and functionality are retained. With an easy to use facility for extending the language, users are allowed to define their own **user-defined operators** by encapsulating existing **primitive operators** and/or **user-defined operators** previously defined by users. The user-defined operators become part of DFQL but can also be deleted from DFQL if they are thought of to be no good to exist.

As have been presented previously in other papers (Angelaccio, 1990 and Sckut, 1991), the following goals are met in DFQL:

- Employ a fully graphical environment as an user friendly interface to the database.
- Sufficient expressive power and functionality, including relational completeness.
- Ease-of-use in learning, remembering, writing and reading the language's constructs.
- Consistency, predictability, and naturalness (in both syntax and function).
- Simplicity and conciseness of features.
- Clarity of definition, lack of ambiguity.
- Ability to modify existing queries to form new queries incrementally.
- High probability that users will write error-free queries.
- Operator extensibility--allow the user to create new operators in terms of existing ones, analogous to defining a function in a programming language.

Partial implementation of DFQL was done in (Clark, 1991). This thesis improves some shortcomings in Clark's implementation. For example, as you can see in (Clark, 1991), a new operator always be created at the upper left corner of the drawing area. There is no way to create an operator by directly typing at wherever users like it to be. After being created, users are required to drag the operator to a new location where it is supposed to be. Several tedious steps of operation are required to simply delete an existing operator. Without creating an operator in the drawing area first, no on-line help message about this operator can be get. After

users saved the help message of a new user-defined operator, the content of the help message can not be changed. So, should the help message of an user-defined operator need to changed, the only way to do is to delete the operator permanently and then redefined it and type in the help message carefully without any more mistake. No more than one DB INTEFACE window (an interface window in which users construct queries) are allowed at the same time is also a severe limitation of productivity to DQFL. If more than one **DB INTERFACE** window is allowed to exist at the same time then the reference to other existing queries may be more convenient during the construction of a new query. There are some more shortcomings which we will discuss in later chapter can be improved. All of these drawbacks increase users' overhead to construct a query and limit the productivity of DFQL.

The main concern of our new development of DFQL as we will discuss in the later chapters is to introduce a more ease-to-use and ease-to-learn user interface, so the shortcomings we found in Clark's DFQL interface need to be lifted in order to reduce the overhead of the construction of queries. We believe that some features also need to be incorporated into the new DFQL so that the performance can be improved and the productivity and power of the new version of DFQL can be increased. For example, instead of rigidly creating a DFQL object at the upper left corner of the drawing area of the query window, users should be allowed to create a new DFQL object simply by clicking the mouse anywhere in the drawing area and entering the text right on the screen. Multiple **DB INTERFACE** window (in fact, as you will see that in our new DFQL, **DB INTERFACE** window is replaced with

Query Window) and modification of the on-line help messages of existing user-defined operators are allowed.

Since the interest of this thesis is focused on the user interface, the method of intermediate code generation and linkage to the existing backend database management system (DBMS) is not covered. For those who are interested in this issue can consult (Clark, 1991).

## B. OBJECTIVES

. We started from the human factor aspects and utilized the techniques of object-oriented programming. Since, as stated in last section, the main concern of our new development of DFQL is to introduce a more ease-to-use and ease-to-learn user interface the following interface principles (Wu, 1990) are to be met:

- Interface Principle 1: Be able to provide more information when asked.
- Interface Principle 2: Be able to display multiple information at the same time.
- Interface Principle 3: Be able to recover from the unintended or erroneous operation.
- Interface Principle 4: Be able to perform the same operation in more than one way.
- Interface Principle 5: Be able to prevent severe problem from happening.
- Interface Principle 6: Be able to prevent modifications that are not supposed to be made.

We'd also like to discuss some important features and benefits of object-oriented programming such as reusability, message passing, responsibility driven,

inheritance, polymorphism, etc. we utilized in our implementation and present some experiences and lessons we learned during the development of our new DFQL.

## C. OVERVIEW

Chapter II describes the concepts of the previous DFQL user interface done by Gard J. Clark, The concepts of our new DFQL is basically the same with those introduced in this chapter. We also discuss the details of how shortcomings we found in the previous DFQL interface can be improved.

Chapter III describes our new DFQL user interface and introduces some important features which we have added to the new version of DFQL. The human factors analysis of this new interface, and how the interface principles we mentioned in last section are met is also covered in this chapter.

Chapter IV presents a brief introduction to the programming language we use -- Prograph and describes why we have adopted the object-oriented approach and evaluates the object-oriented design of our new DFQL user interface implementation and the benefits of class, message passing and some other features of object-oriented programming technique.

Chapter V describes some important lessons we have learned during the development of the new DFQL interface. A summary of this thesis is also given in this chapter.

# II. PREVIOUS DFQL INTERFACE

## A. CONCEPTS

As its name says, DFQL is a DataFlow Query Language. Queries are defined by the user connecting the desired **objects** graphically in the drawing area of a query window. Upon completion of a graphically constructed query, the query can be translated to the equivalent text-based SQL query by the DFQL interpreter and sent off to the backend database management (DBMS) for execution. After execution, the results will then be sent to the **Query Results Window** displaying on the computer screen.

Two categories of objects are defined in DFQL interpreter: **DFQL operators** and **text objects**. Data are flowing from one operator to another along the query. Text objects are used as input data only, so there is no data flowing in to a text object. Operator execution is controlled by the presence of the input data for that operator. When all the data required become available the operator may execute or fire.

### 1. DFQL Operators

There are two categories of operators defined in DFQL interpreter: **primitive operators** and **user-defined operators**. An user-defined operator is one that has been constructed by the user from primitive operators and possibly other previous created user-defined operators. So we can say that an user-defined operator is a compact query with some essential input data unspecified.

All operators in both categories except operators **DISPLAY** and **SDISPLAY** (we will mention them later) have the same appearance. A sample operator named **select** is shown in Figure II.1 below. Each operator is made up of three components: the body with the name of the operator, the input nodes and the

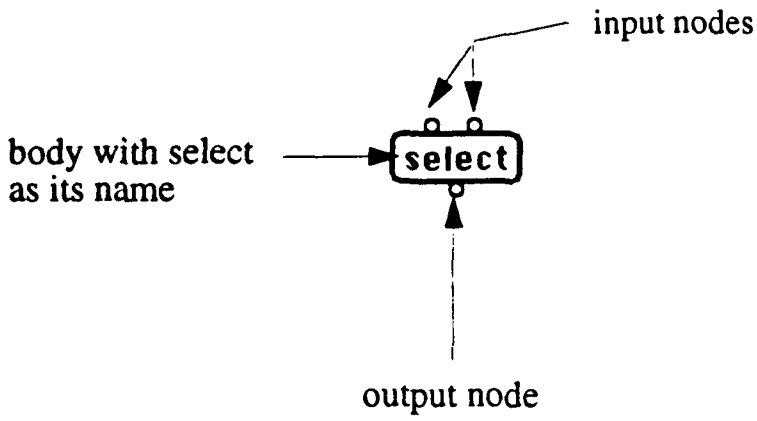



**Figure II.1 Operator Construction**

output node. They can have multiple input nodes but there is only one output node for each operator.

The body of the operator is the rounded-rectangle with the input nodes and output node attached to it. The input nodes and output node are represented by small circles. The input nodes are where the data from other operator or text object flowing into or fed into the operator.The output node is where the result of the execution of this operator flowing out of the operator. The intermediate sult of a operator may then be passed to other operator(s) by connecting the operator's output node to other operator's input node(s). An output node can be connected to multiple input nodes whereas an input node can only be connected to exactly one output node. The output from each operator is always a relation.

The appearance of operators DISPLAY and SDISPLAY is shown in Figure II.2. We are intended to make them different from other operators because as we will


**Figure II.2 Construction of DISPLAY and SDISPLAY**

see later that these two operators are not provided to execute query functions but to allow user to print the contents of relations on the computer screen. They create no results.

### *a. Primitive DFQL operators*

There are fifteen DFQL primitive operators provided in DFQL interpreter. Among them, there are six basic operators. These six primitive DFQL operators and a corresponding translation into SQL are shown in Figure II.3. The other primitive operators are shown in Figure II.4.

### *b. User-Defined Operators*

With user-defined operators, users can construct their own operators for situations that are unique to their query needs and give those new defined operators names that reflect the functions they perform. For example, instead of creating a query as shown in Figure II.5, we can define an operator sel-project and construct the query as shown in Figure II.6, where operator sel-project is defined by

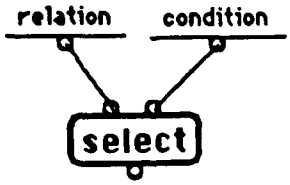
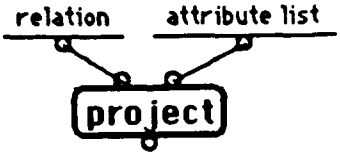
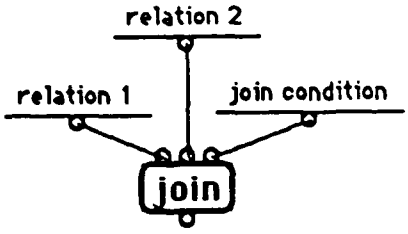
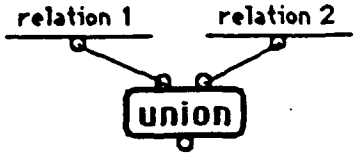
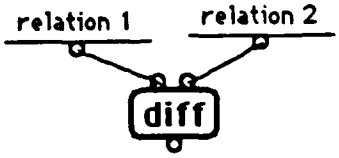
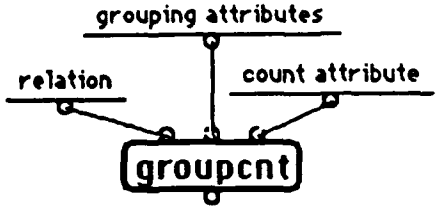
| DFQL Operator | SQL |
|---|---|
|  | SELECT DISTINCT *<br>FROM relation<br>WHERE condition |
|  | SELECT DISTINCT *<br>FROM relation |
|  | SELECT DISTINCT *<br>FROM relation1 r1, relation2 r2<br>WHERE join condition |
|  | SELECT DISTINCT *<br>FROM relation1<br>UNION<br>SELECT DISTINCT *<br>FROM relation2 |
|  | SELECT DISTINCT *<br>FROM relation1<br>MINUS<br>SELECT DISTINCT *<br>FROM relation2 |
|  | SELECT DISTINCT grouping attributes<br>COUNT(*) count attribute<br>FROM relation<br>GROUP BY grouping attributes |

**Figure II.3 DFQL Basic Operators**

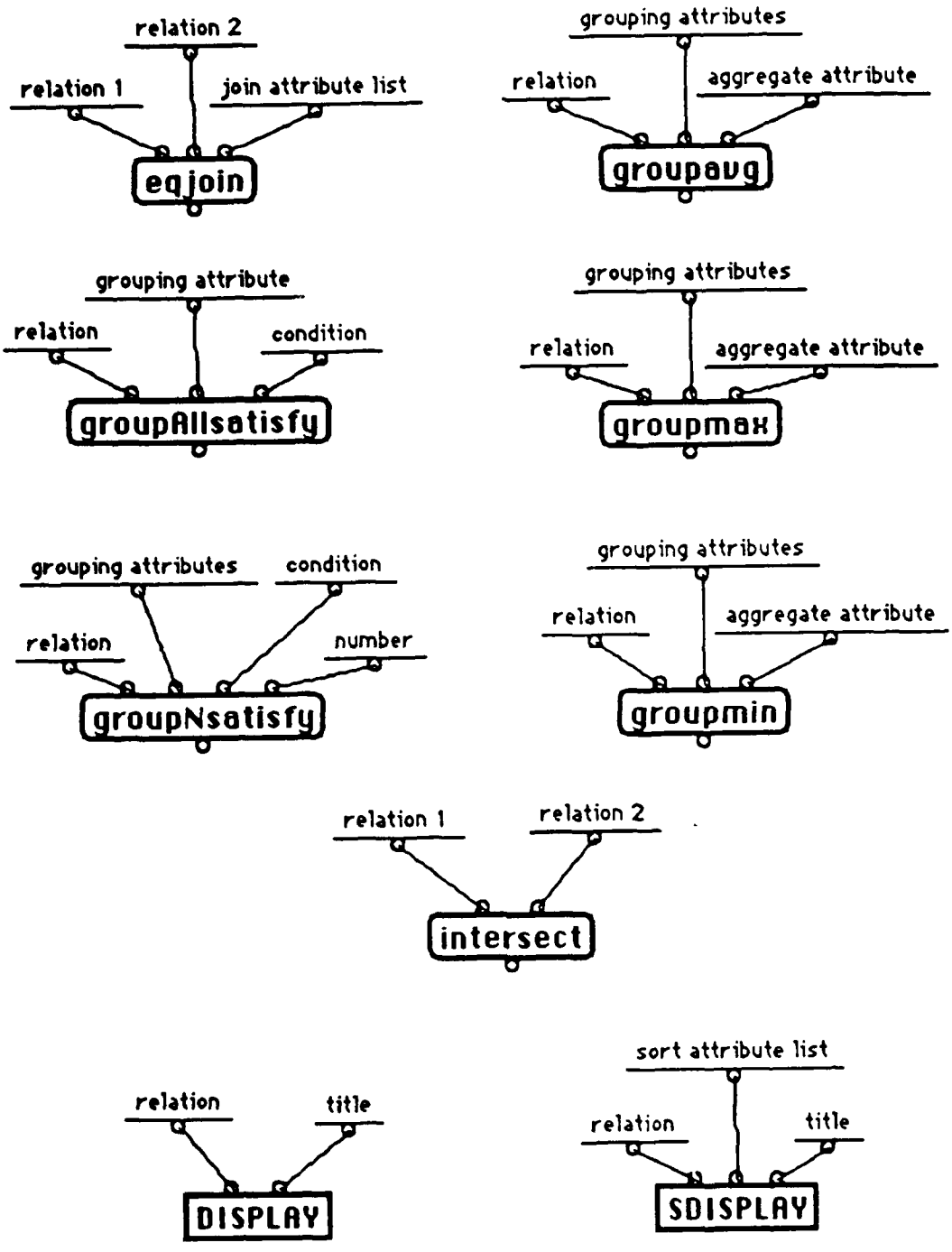



**Figure II.4 Other DFQL Operator**

combining primitive DFQL operators select and project. We will discuss how to define an user-defined operator.

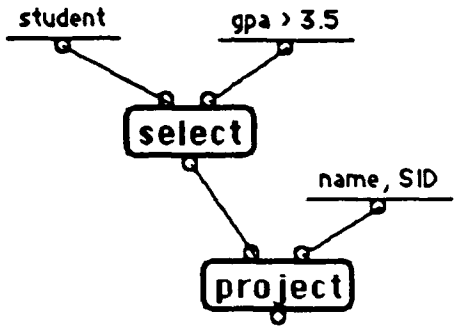



**Figure II.5 A Sample Query**

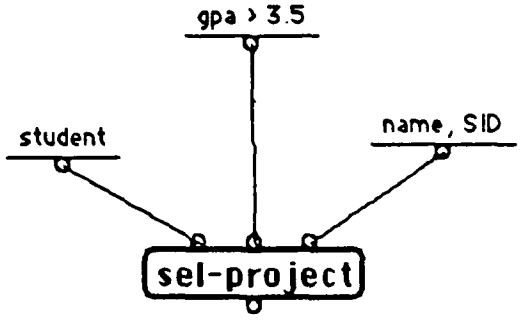



**Figure II.6 An User-Defined Operator**

The most significant advantage gained from the utilization of user-defined operators is that abstraction of complicated queries into a single user-defined operator is allowed. This make it easier to understand and use operators correctly. It also conserves space of the drawing area in the query window.

## 2. Text Objects

A special notation is used to provide textual input to the DFQL operators. Text entered by the user shows up in the query window as an object with the text attached to an output node as shown in Figure II.7. The text object can be interpreted

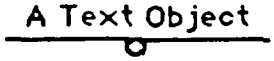



**Figure II.7 Text Object**

in two different ways. If the text is the name of a relation, the output at the root can be thought of as an instance of that specific relation. If the text represents a condition, a list attributes, or some other textual input to another DFQL operator, then the text is passed on to that operator as a textual argument.

## 3. DFQL Query Construction

All DFQL queries exist as a dataflow program in which text objects and operators are connected by dataflow paths. The data flow paths are represented as the lines in the DFQL query that connect the input and output nodes of the DFQL

objects. Execution of the query can be visualized as flowing from the top of the diagram to the bottom. When the input arguments to an operator are available, that operator may execute, or fire, producing its output which will then flow on to the other connected operators. Since text objects have no inputs, they may fire at any time. Execution of the query continues until all input has been exhausted. The general idea behind DFQL query construction have been presented in (Clark, 1991). Instead of getting into all the details, we give an sample query as shown in Figure II.8 below.

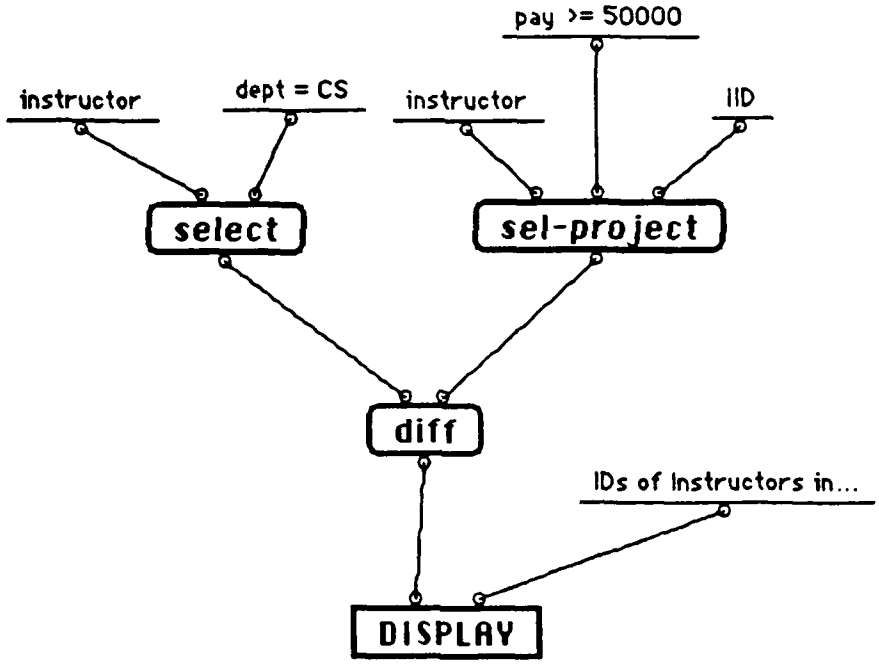



**Figure II.8 A Sample Query**

This query uses the diff operator to return the IID of instructors in CS department whose pay is lower than 50,000 dollars. In this query the user-defined operator sel-project from Figure II.6 is used.

## B. SHORTCOMINGS OF THE PREVIOUS DFQL INTERFACE

The previous DFQL interface as shown in Figure II.9 is made up of a DB INTERFACE window and a menu with eight menu items. The DB INTERFACE

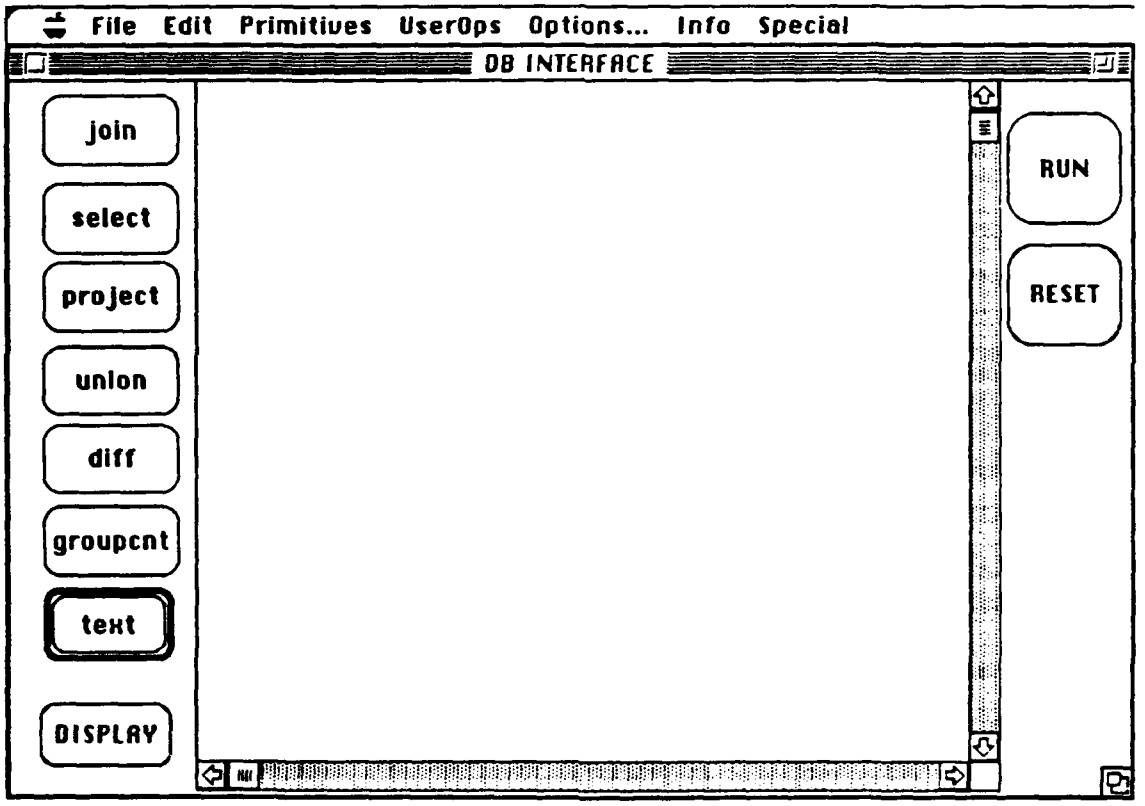



**Figure II.9 DB INTERFACE**

window is the main interface from which users can construct their queries. The description of these components can be find in (Clark, 1991). We just present some shortcomings we found in it and discuss what we can do about them.

### 1. Tedious Query Construction

The most significant drawback in the Clark's DFQL interpreter is the way to create an object in the **DB INTERFACE** window. Every time an object's button (either DFQL operator or text object) on the left portion of the **DB INTERFACE** window is clicked or an item (operator's name) from menu item **Primitives** of the main menu (Figure II.10) is selected, DFQL always creates it on the very upper left

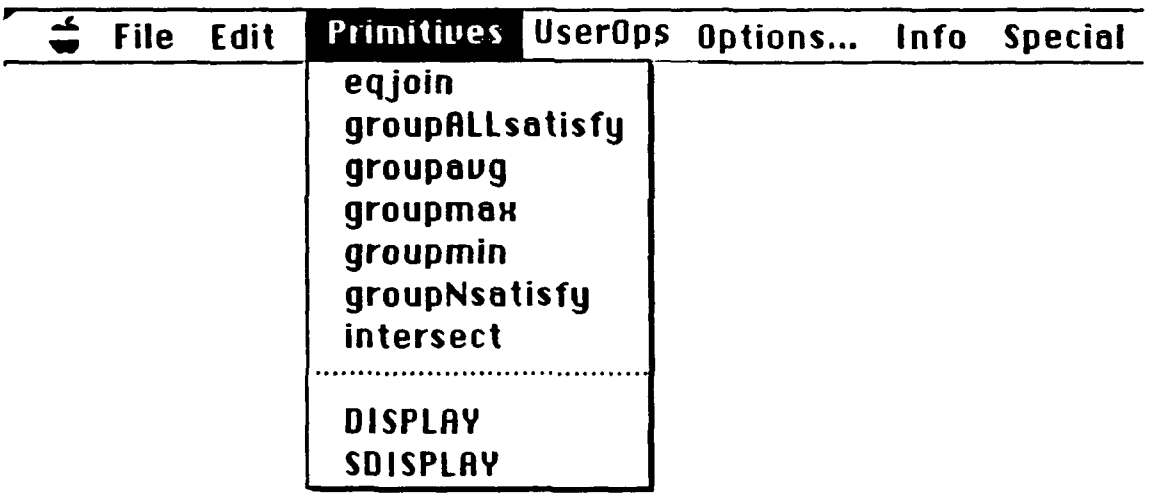



**Figure II.10 Primitives Menu**

corner of the drawing area in the **DB INTERFACE** window (Figure II.11, an operator named **union** is created in the **DB INTERFACE**). In order to put it on the right place where the object is supposed to be, the user need to click the object and

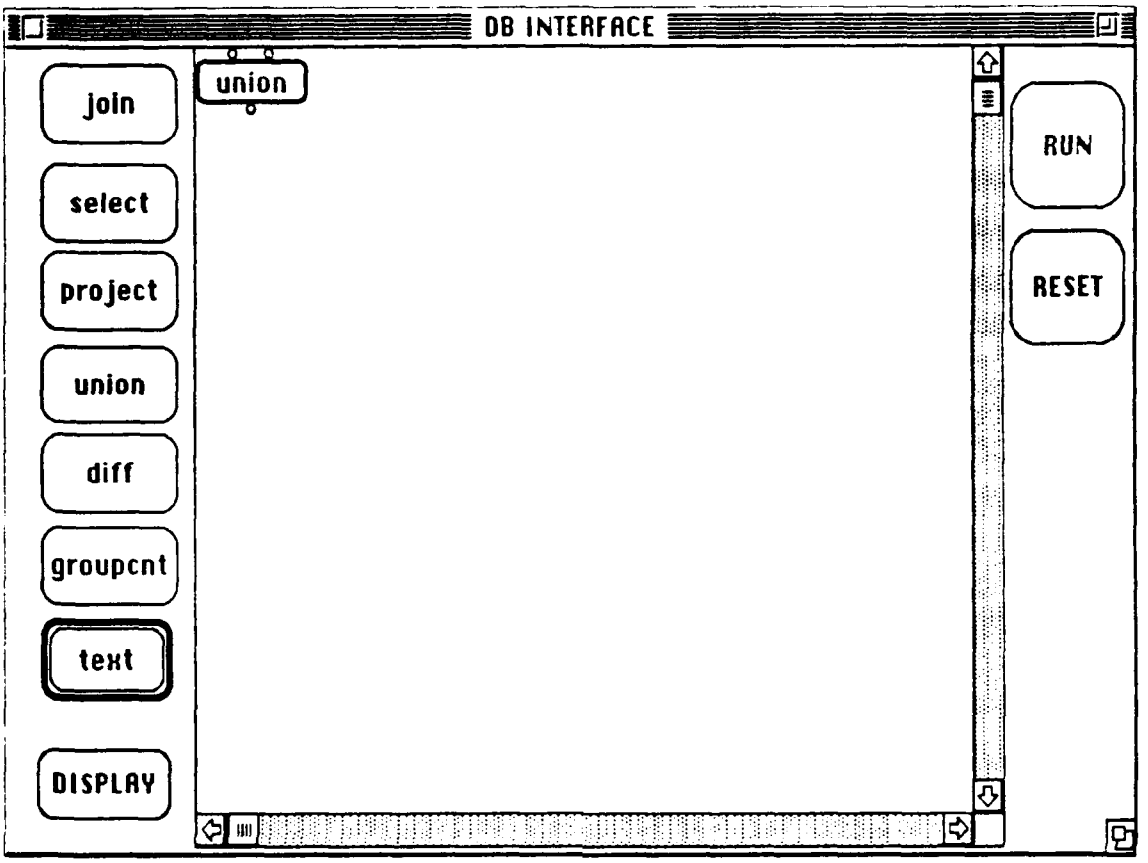



**Figure II.11 Creation of an Operator**

drag it. But if the checkable menu item **Select** in the **Edit** menu (Figure II.12) is checked then all of sudden, when the user click the object in order to drag it, she(he) will find that instead of dragging it, the object's color is converted,i.e., the object is selected to be deleted or deselected. So the user need to uncheck, or turn off Select and then drag the object. Figure II.13 shows a selected operator

Not being able to create a text object by directly typing on the drawing area is another drawback. So we believe that allowing the user to locate the exact location

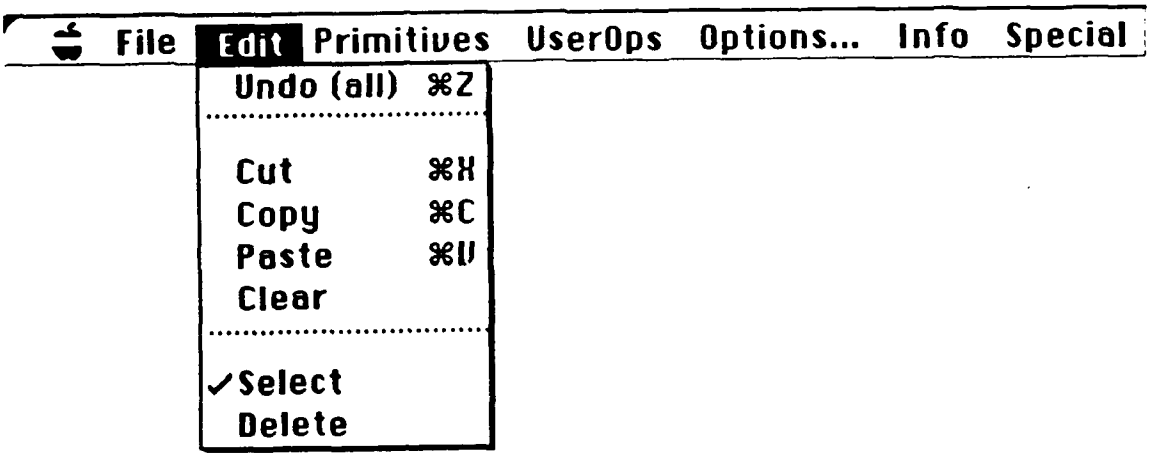



Figure II.12 Edit Menu


Figure II.13 A Selected Operator

for the next object by simply clicking anywhere in the drawing area and then typing in the text or an existing operator's name directly will ease the user's duty.

To create an user-defined operator in the drawing area, an awkward process requires the user to select the menu item **Select** from menu **UserOps** (Figure II.14)

and selects the specific user-defined operator from a dialog box (Figure II.15).Allowing the user-defined operators to be created the same way as primitive operators are would be a better solution. We will see how this procedure can be eased in our new implementation.

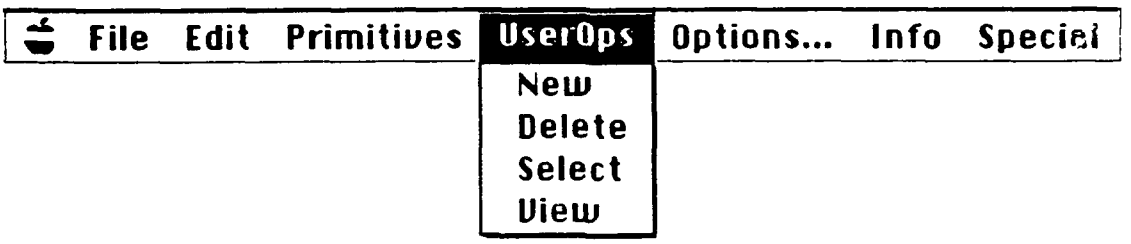



Figure II.14 UserOps Menu

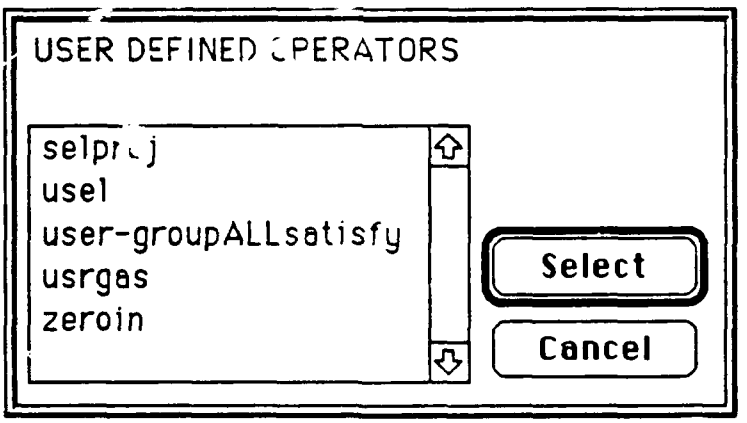



Figure II.15 Dialog Box for User-Defined Operator Selection

### 2. Tedious Delete Operation

To delete an object from the drawing area of the **DB INTERFACE** window, the user also need to check, or turn on **Select** first and then select the menu item **Delete** from menu **Edit**. To make the delete operation easier, we will point out another way to delete objects without to check **Select** first. That is to say that **Select** is no longer needed in our new implementation of DFQL interpreter and can be remove.

### 3. No Concurrent Query Constructions

No more than one **DB INTEFACE** window are allowed at the same time is also a severe limitation of productivity to DQFL. If more than one **DB INTERFACE** window is allowed to exist at the same time then the reference to other existing queries may be more convenient during the construction of a new query and users are also able to construct multiple queries in different windows at the same time.

### 4. Rigid User Operator Definition

To create a new user-defined operator, previous DFQL disables the object creation and deletion abilities. So the desired internal structure for the new user-defined operator must be exist in the drawing area before getting into the operator definition mode. This means that if the user want to define a new operator during the process of query construction, he (she) needs to delete some operators unneeded in the structure of the new operator. While in the operator definition mode, if the user find that some operators need to be added to or deleted from the internal structure of the new operator being constructed, he(she) need to give up all the efforts done before and return to the query construction mode then add or delete some operators.

We believe that all the object creation and deletion abilities should be retained in operator creation mode so the user does not need to go back and forth between query construction and operator definition modes.

### 5. Reference Information Exit Too Short

After viewing the internal structure of an user-defined operator, the window in which the internal structure of the user-defined operator is displayed need to be closed before the user can proceed to do other stuff. The window and its contents are not allow to stay available. This means that every time when the user need to view the internal structure of an user-defined operator, the viewing procedure need to be repeated. This is also a drawback needed to lifted.

### 6. Restricted Way of Getting Help

The previous DFQL provides help information describing each operator by requiring the user to double-click on an operator existing in the drawing area. This is the only way to get the information of an operator. So it is impossible for an user to get the information of any operator before it is created in the drawing area. But it is always the case that users need to consult the help messages of operators from time to time, especially for user-defined operators. This need is more obvious when the number of user-defined operators gets larger. So when users are not really sure what the functionality of an operator they may want to use is, we can not expect them to tolerate the inconvenience of creating an operator, double clicking it to get the help message, finding out that it is not what exactly they want and then go through the same procedures again to try other operators.We will added some more features of on-line help to our new DFQL interface.

# III. NEW DFQL INTERFACE

Like the previous DFQL, new DFQL is also implemented on an Apple Macintosh. Basic operations of this implementation depend heavily on use of the mouse and pull-down menus. In this section we present an detailed discussion on how the user interacts with the DFQL interpreter to construct and execute queries.

## A. STARTING THE PROGRAM

Upon start-up, the user is presented with the screen shown below as Figure III.1. We do not use **DB INTERFACE** as the title of the main window, instead, we use **Query Window** followed by an number indicating the sequential order. That is because we allow multiple **Query Windows** to exist at the same time in our new DFQL. So what we see upon start-up is a main window entitled **Query Window 1** along with a pull-down menu.

## B. QUERY WINDOW AND ITS ITEMS

A Query Window as shown in Figure III.1 can be moved, resized, closed and roomed by utilizing the basic functions of Apple Macintosh. We assume that readers are all aware of those functions.

### 1. Buttons

The Run button executes the query which is currently constructed in the drawing area. Run will first check that the query is correctly constructed. Then the query will be sent off to the backend DBMS for processing. Query results returned from the database will be displayed in a separate **Query Result Window** (Figure III.2). We will mention Query Result Window later.

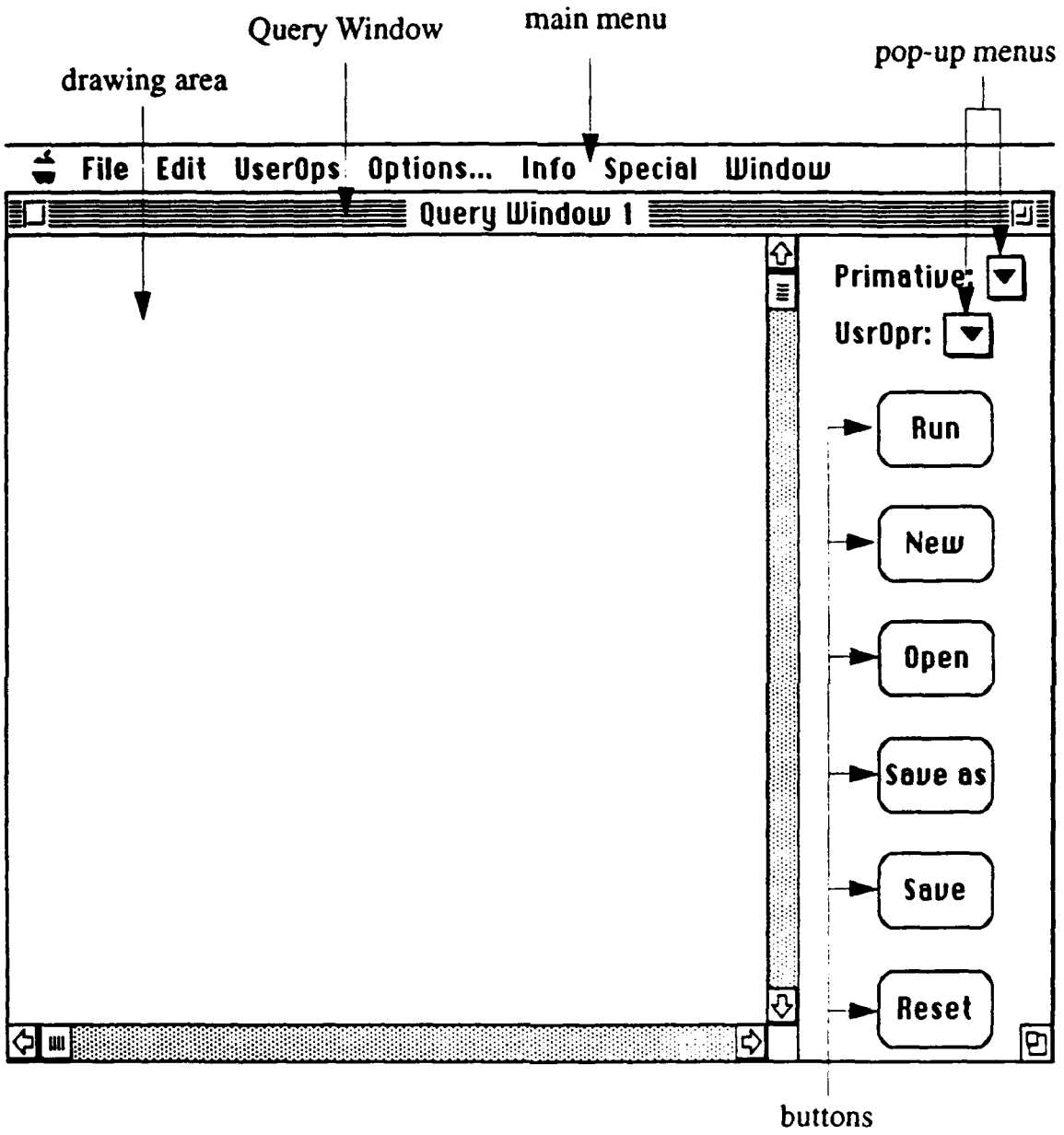
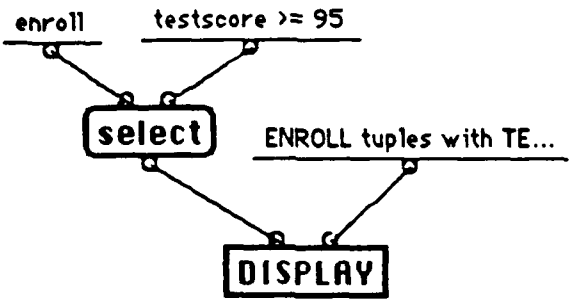



**Figure III.1 Query Window**




Query Results

```
==============================
ENROLL tuples with TESTSCORE >= 95
==============================

SID   CID    GRADE  TESTSCORE
------ -------  ----------  ----------------
S1    CS05   A                98
S2    CS10   A                95
9 records selected.
```

**Figure III.2 A Query and its Query Results Window**

**The New** button creates a new **Query Window N**, where N is the number indicating that this is the Nth Query Window opened since the start-up of the DFQL interpreter. When a new Query Window is opened, a query filename **Untitled #N**, where N is the same number as that in the title of the Query Window. All Query Windows retains all capabilities as the first one does. By creating new Query Windows, the user is allow to construct queries in different Query Windows concurrently. This feature make the DFQL more productive.

**The Open** button allows the user to retrieve a previously saved query file from disk. When Open is clicked, a dialog box is presented from which the user can select the stored query file for retrieval. Before a selected query file can be retrieved onto a Query Window, DFQL will check if it has been opened in any Query Window or if there is any query with the same file name exists in any Query Window. If this situation is true, then DFQL will prompt a message telling the user that in which *Query Window the query file selected* has been opened. This feature agrees with **Interface Principle 5: Be able to prevent severe problem from happening.** If the selected query file has not been opened, it is immediately retrieved onto the Query Window and the query (the content of the newly opened file) appears in the drawing area. The previous filename assigned to the Query Window will be replaced by the new one.

The **Save** button stores the current query onto disk with the name that is currently assigned to the query. The **Save as** button allows the user to store the current query with a new name. When **Save as** is clicked, a file naming dialog box displayed. The user can enter the new file name for the current query to be saved. If the entered file name matches an existing file in any Query Window, the user will be

prompt that a file with this name is currently opened in a specific Query Window and the save operation is abandoned and the file naming dialog box appears again asking the user to enter a new filename. On the other hand, if the entered filename matches an existing file in the disk, the user will be asked whether or not he(she) really want to replace the previous stored file. If not, the user can either abandon the saving attempt or enter a new name. When an appropriate name has been given the query will be saved to disk.

The **Reset** button clears the current query from the drawing area and from the computer's memory. When the user has no desire to save the current query, Reset can be used to set up a new query. This is also a shortcut to delete all objects in the drawing area and give the user a blank drawing area to start constructing a new query. After Reset, **Untitled #N** is assigned as the file name to the **Query Window N** again. This feature which prevent the user from "destroying" an existing query file by simply clicking the mouse on the **Reset** button is also based on **Interface Principle 5: Be able to prevent some severe problem from happening.**

## 2. Drawing area

The **drawing area** is the portion of the Query Window that is bounded by the horizontal and vertical scroll bars. This area starts out blank and is used to graphically construct the DFQL query. As the query becomes larger so that portions of drawing area are hidden from the user's view, the scroll bars may be used to bring the hidden portions into view. There are two ways to create an object: First, the user clicks the mouse anywhere he(she) wants the object to appear in the drawing area. **An "input box"**, the gray rectangle, with a flashing cursor at the center appears on where the mouse is clicked waiting for the user to enter any text from the keyboard.

After the input box shown up, the user can select either an operator from the pop-up menu **Primitive** or an user-defined operator from the other pop-up menu **UsrOpr** by clicking on the up-side-down triangle. The input box will be replaced with the selected operator or user-defined operator. The second way is also clicking in the drawing area to create an input box first and then keep typing in text. Should there be any typing mistake, the **delete** key on the keyboard can be used to delete the mistyping characters. As the **Return** key is pressed, the text input ended and the DFQL interpreter checks whether or not the entered text is in the name list of the primitive operators or user-defined operators. If it is, then the operator whose name is entered will appears in the drawing area and centered at where the mouse is clicked. If the entered text is not in the name list of the primitive operators or user-defined operators, a text object with the text entered will appear and is also centered at where the mouse is clicked. There is an example shown in Figure III.3. The feature allowing operators to be created in more than one way supports the **interface Principle 4: Be able to perform the same operation in more than one ways.**

The DFQL interpreter is case sensitive. If the entered text is **Select** instead of **select** (the name of a primitive operator) and we do not have an user-defined operator named **Select**, then it is created as a text object, not an operator. If no characters are entered, after pressing the **Return** key, the "input box" will disappears.

In order to construct a DFQL query, the query objects must connected with the desired data flows. Data flows are the lines connecting output node of any given object to the input node of another object (or objects). To draw these lines, the user must click the mouse on either an output or input node. Once the mouse button has

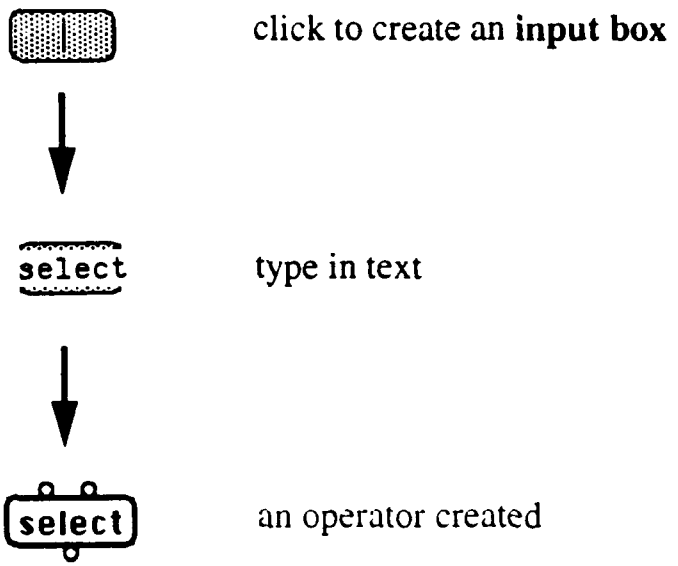



**Figure III.3 Creating an Operator by Typing**

been released, a rubber-band line will be drawn from that node to the current position of the mouse. Clicking on the input node or output node of another object will connect the dataflow line from the originating node to the newly indicated node if the connection makes sense. The following attempted connections do not make sense:

- From input to input
- From output to output
- Between nodes of the same operator
- Making a cycle

Should any of these nonsense connections be detected, an error message is presented stating that the attempted connection is not allowed. While the rubber-band line is "on", clicking the mouse in an blank portion of the drawing area will turn off the rubber-band line if the user has decided not to make a connection after all. Since an input node may have only one input flow, if the user connects a dataflow line to an input node that already had one, the previous dataflow line is deleted automatically.

In order to move an object within the drawing area, the user clicks the mouse on the object and drags it to the desired position while holding down the mouse button. If the user clicks the object and release the mouse button immediately, the object will be selected, i.e., its color will be converted. Selecting a DFQL object existing in the drawing area has two effects. First, it allows the selected object to deleted. Secondly, selecting a DQFL operator allows the user to retrieve intermediate results from the query. When an operator is selected and the **Run** button is clicked in the Query Window, The query will be executed up to and including the selected operator. The result of this partial query will then be displayed in the Query Result Window. To delete a selected object, the user can select the menu item **Delete** from the menu **Edit**. The equivalent operation to delete a selected object is simply pressing the **Command** and **D** keys on the keyboard simultaneously. The user is also allowed to select several objects and then delete them all at one time. However, an object selected can still be moved around in the drawing area. Simply clicking the mouse on a selected object will deselects it.

Double-clicking either on an primitive operator or on an user-defined operator will bring up a help window describing that operator. The **Help Window** is

shown in Figure III.4. The left portion of the Help Window is a scrolling list of the

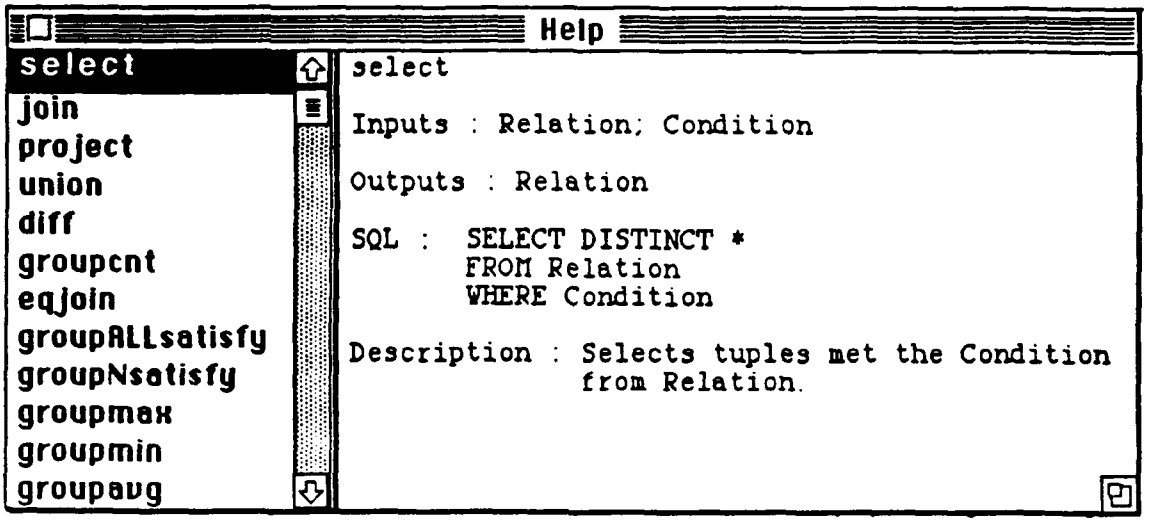



**Figure III.4 Help Window**

names of all operators. The highlighted name is the one we double-clicked on. The message shown in the right portion of Help Window is the description of the highlighted operator.

Double-clicking on a text object opens up an editor for that object's text string. All of the Macintosh 's normal text editing functions such as cutting, copying, and pasting text from the Macintosh clipboard are supported in this editor. When the OK button is clicked, the previous text for the object is replaced with the new string.

If the mouse is double-clicked on an output node, the columns of the relation flowing out of that node are displayed.In this way, the user can determine what attributes may be used by operators subsequent to that point in the query graph. This assistance is very important in the construction of large queries in which the attributes become hard to keep track of. Also, when user-defined operators are used,

it is important to be able to easily determine what the names of the attributes are that the operator produces. There is an example shown in Figure III.5.

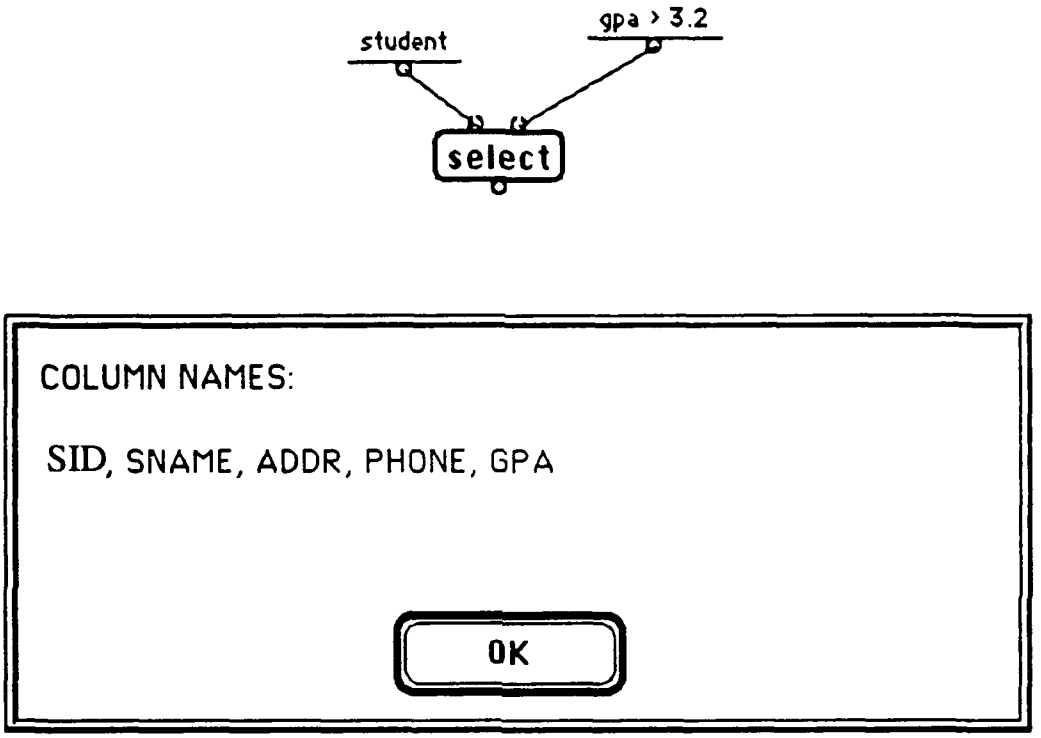



**Figure III.5 Columns of a Relation at the Output of select**

### 3. Pop-up menus

There are two pop-up menus in the Query Window: **Primitives** and **UsrOpr**. Primitives contains the names of all primitive operators while UsrOpr

contains the names of all the user-defined operators. When a new user-defined operator is created, its name will be attached to the end of UsrOpr.

These two pop-up menus provide another way to create an operator. After the user clicks the mouse on the drawing area to create an input box, instead of entering text from the keyboard, the user can also click on the reversed triangle of either pop-menus depending on what kind of operator he(she) want to create and then selects the specific operator from the pop-up menu list. The input box will be replaced with the selected operator.

## C. OPERATOR DEFINITION WINDOW

When the menu item **New** in menu **UserOps** is selected, the **Operator Definition Window** as shown in Figure III.6 is displayed on the computer screen. The Operator Definition Window is very similar to Query Window. The difference between these two windows are that they have different set of buttons and there is an "input bar" in the drawing area of the Operator Definition Window.

The drawing area plays exactly the same role as that in the Query Window does. The input bar is used to define where the input data to the user-defined operator will be sent internally. Clicking the mouse on the input bar will create additional input nodes for the user-defined operator. If too many input nodes are created by mistake, they can be removed by clicking the mouse on the **Delete Input** button. Each click deletes one input nodes from the input bar. Once the desired number of input nodes are created, they must be connected to the desired operators in the drawing area. All input nodes of the operators inside the user-defined operator must be connected. Also, there may be only one unconnected output node in the user-defined operator. This single node becomes the output node for the entire user-defined operator.

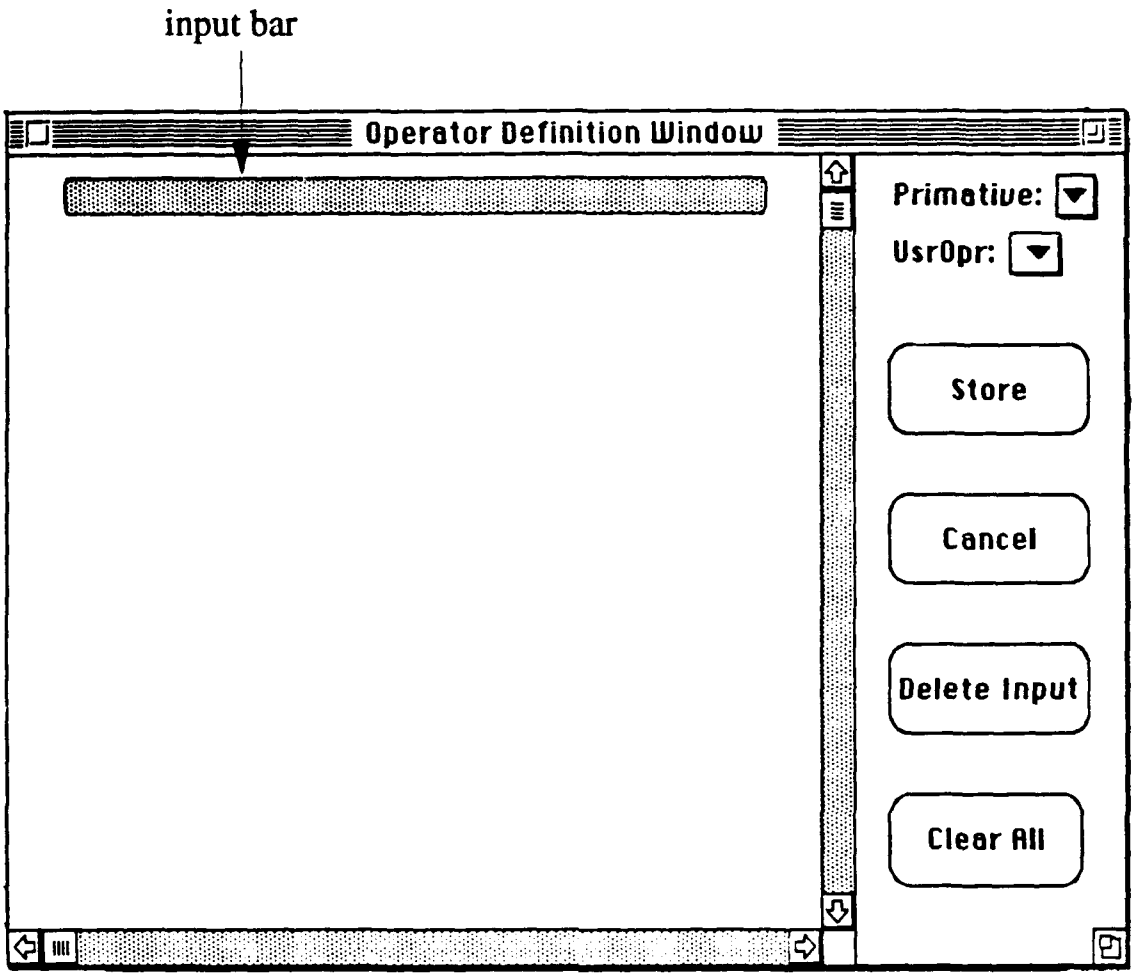



Figure III.7 Operator Definition Window

When the **Store** button is clicked, DFQL interpreter first check the internal structure of the user-defined operator to ensure that all necessary connections have been made and query criteria have also been met. Then the user will be asked a name for the new defined operator and a description that will be used as help message for the operator. This feature agrees with **Interface Principle 1: Be able to provide more information when asked**. The operator's name is checked for uniqueness

among all existing operators. If the uniqueness is ensured, the name of the new user-defined operator will be added to the end of the scrolling lis of Help Window and the UsrOpr pop-up menu list so the user can use it immediately. New DFQL allows users to define user-defined operators successively as many as their want.

The **Clear All** button clears the drawing area and allows the user to reconstruct the user-defined operator. The **Cancel** button cancels all the operations the user has done since the start of the definition of a new operator.

## D. MENU ITEMS

The items listed in the main menu (Figure III.1) usually remain the same throughout the whole session of the application no matter what window is currently displayed. Any items that are not applicable at a given time are made unselectable and are displayed at reduced intensity, commonly known as being "grayed out". We also provide equivalent key-combinations for some menu items to speed up the user's operations. This feature supports **Interface Principle 4: Be able to perform the same operation in more than one ways**. The menu items are discussed below.

### 1. Apple

Apple is a standard Macintosh menu that has no relation to DFQL. It provides access to Macintosh utilities called "Desk Accessary" and should be accessible at all times (Apple, 1985, p. I-54). The only DFQL specific item in this menu is the "**About...**". When this item is selected, a brief information about DFQL interface is displayed.

## 2. File

As shown in Figure III.7, the **New** item executes exactly the some function as the **New** button in the Query Window. It is very possible that the user may close the only Query Window unintendedly. Without this **New** item, the user will not be able to open any new Query Windows and need to quit DFQL unwillingly. This **New** item is provided basing on **Interface Principle 4: Be able to perform the same operation in more than one ways** and **Interface Principle 3: Be able to recover from the unintended or erroneous operation.**

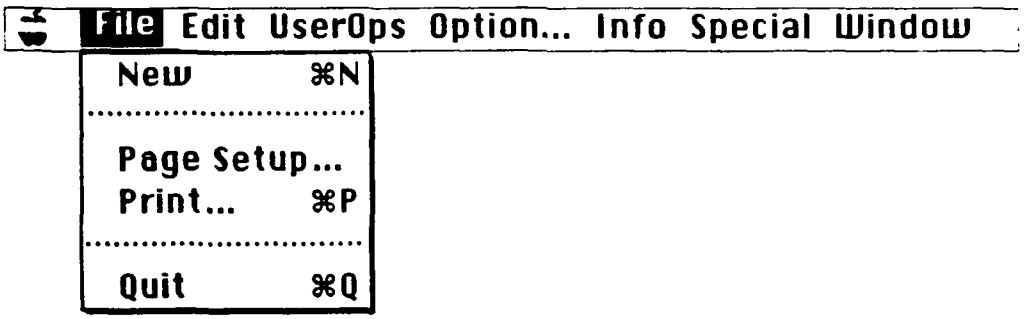



**Figure III.7 File Menu**

The **Page Setup...** item is also a standard Macintosh **File** menu item which allows the user to change printer parameters such as the size of paper, print quality and orientation. The Print... item is provided to print out the information of the front window of the DFQL.

The **Quit** item closes all opened windows and terminates the DFQL interpreter execution. Before closing each Query Window, DFQL will check whether or not the query in the drawing area of this Query Window has been changed since last **Save**. If there are new changes since last **Save**, a dialog box pops

up asking the user if he want to save or not. The user should click either Discard or Save in order to proceed. This precaution agrees with **Interface Principle 5: Be able to prevent severe problem from happening.**

### 3. Edit

The **Edit** menu as shown in Figure III.8 is also a standard Macintosh menu. It provides the text editing functions of **Cut**, **Copy**, **Paste**, and **Clear**. They are available whenever the user is editing text items.

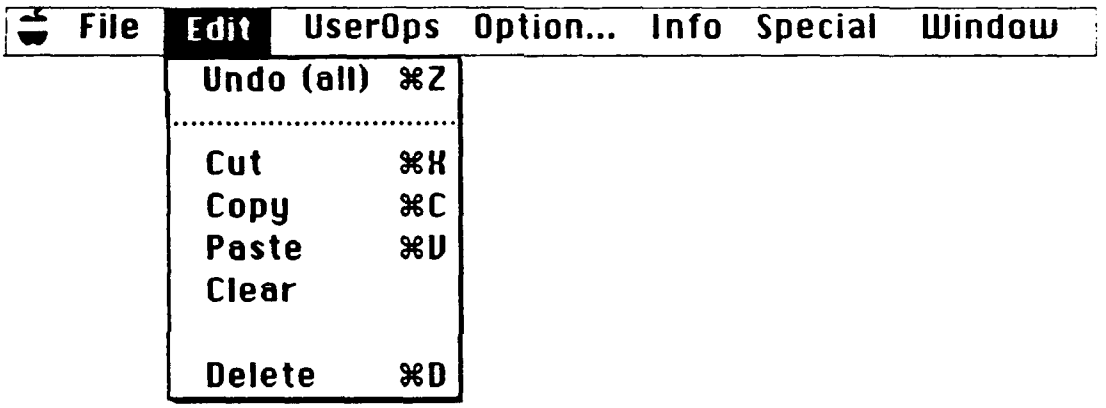



**Figure III.8 Edit Menu**

The **Delete** item deletes the selected object(s). The **Undo(all)** item recovers the deleted object(s) and put them back to the drawing area. This feature is based on **Interface Principle 3: Be able to recover from the unintended or erroneous operation. Undo(all)** is only active immediately following the deletion of object(s).

### 4. UserOps

The **UserOps** menu as shown in Figure III.9 is designed for manipulating user-defined operators. The New menu item is to open the Operator Definition

Window so that the user can define new user-defined operators. The Delete menu item allows the user to delete existing user-defined operators from the DFQL interpreter. When **Delete** is selected. the user is presented with a dialog box containing a scrolling list of user-defined operators, as shown in Figure III.10. When the desired operator is selected, by either double-clicking on its entry or single-clicking on its entry and then pressing the **Select** button next to the scrolling list. Once an user-defined operator is deleted, it will disappear from both the scrolling list of the Help Window and the UsrOpr pop-up menu list in Query Window and Operator Definition Window. The View menu item allows the internal structure of

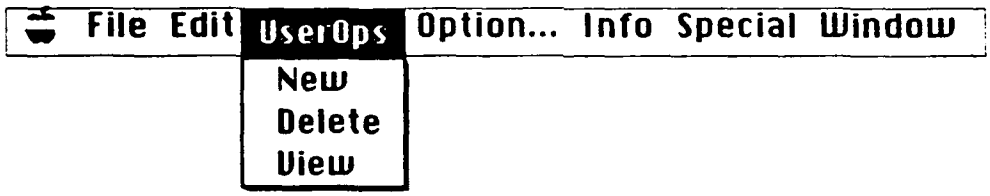



**Figure III.9 UserOps Menu**

an existing user-defined operator to be displayed. When this item is selected, the user is provided a selection dialog box as shown in Figure III.11. After the user chose one of the existing user-defined operators from the box, an **User-Defined Operator** window shows up and displaying the internal structure of that selected user-defined operator. An example of this display is shown in Figure III.12. This display is especially useful if the user-defined operator was provided by someone else. The user is not allowed to modify the internal structure of user-defined operator. In this

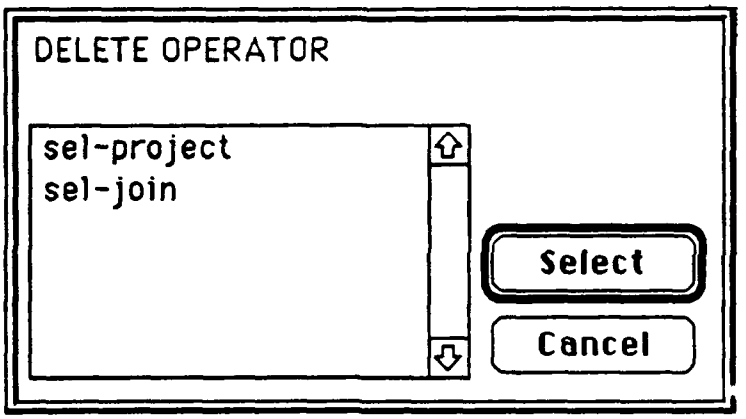



**Figure III.10 Selection Box for User-Defined Operator Deletion**

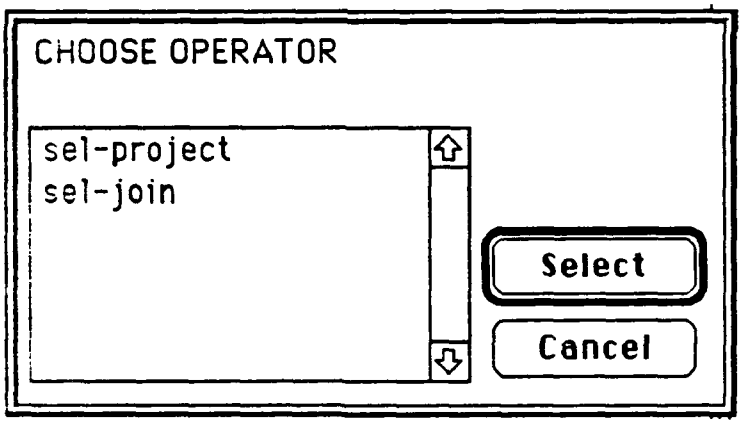



**Figure III.11 Selection Box for User-Defined Operator Viewing**

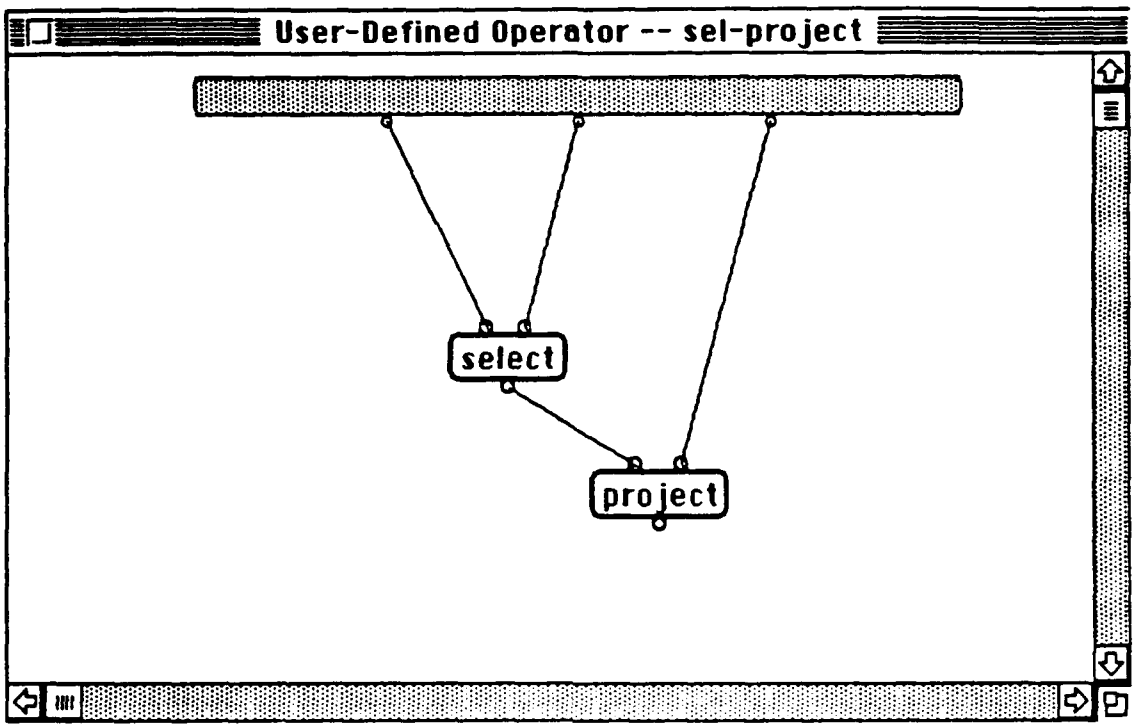



**Figure III.12 User-Defined Operator Window**

way the integrity of the operator is preserved while still allowing some access to the internal for the user's purpose.

### 5. Options...

This menu, as shown in Figure III.13, is imported from (Clark, 1991). We did not introduce any new features to it. All of the items provided in **Option...** menu are toggle items. When the item is active, or "turned on", a check mark is presented next to the item. For example, in Figure III.13 the **Sound** item is turned "on", whereas the **Display Last** and **Show SQL** are "off". When the **Display Last** is turned on, the output of the last DFQL operator executed will be displayed in the

Query Results Window when the query is run. This is useful when incrementally constructing queries because it causes the display of the results without having to use a display operator. Show SQL causes the intermediate SQL code that is generated from the DFQL query graph to be displayed in the Query Results Window along with the results of the query. This display can be used to troubleshoot any execution errors that are not directly apparent from the DFQL query graph. Also, this option allows the DFQL interpreter to be used as a translator in which a DFQL query is input and a DFQL query is output which could then be run on any SQL database system. When selected, the Sound option causes certain easily recognizable sounds to be played at different key points during processing of the query.

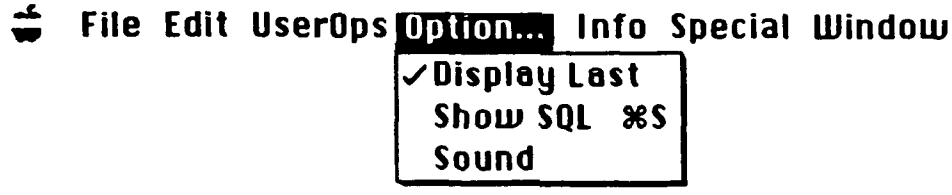



Figure III.13 Option... Menu

### 6. Info

The **Table** item in the Info menu, as shown in Figure III.14 allows the user to retrieve information about what attributes exist for tables in any given relation in

the database. When **Table** is selected a selection dialog box (Figure III.15) is displayed from which the user can choose which table he is interested in. This action will bring up a dialog box displaying the attributes of the selected table as shown in Figure III.16. The Help menu item opens the Help Window. We will cover all the details about Help Window in a specific section later.


**Figure III.14 Info Menu**

### 7. Special

This menu (Figure III.17) is also the original work of (Clark, 1991). The only item, ***ORACLE*Shell*** stars up a separate application to provide the user direct access to the backend DBMS (in this case ORACLE). Since this separate application is not our concern in this presentation, so we do not get into its details.

### 8. Window

There is only one menu item, Cascade Win (Figure III.18) in **Window** menu. As the user proceeds, there may be several windows opened on the computer screen at the same time. Some of them may be overlaid by the others. When **Cascade Win** is selected or its equivalent key combination, **Command-W** is

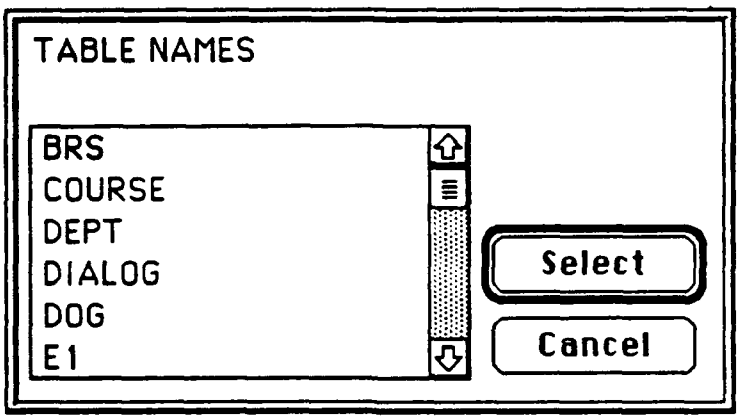



**Figure III.15 Selection Box for Table**

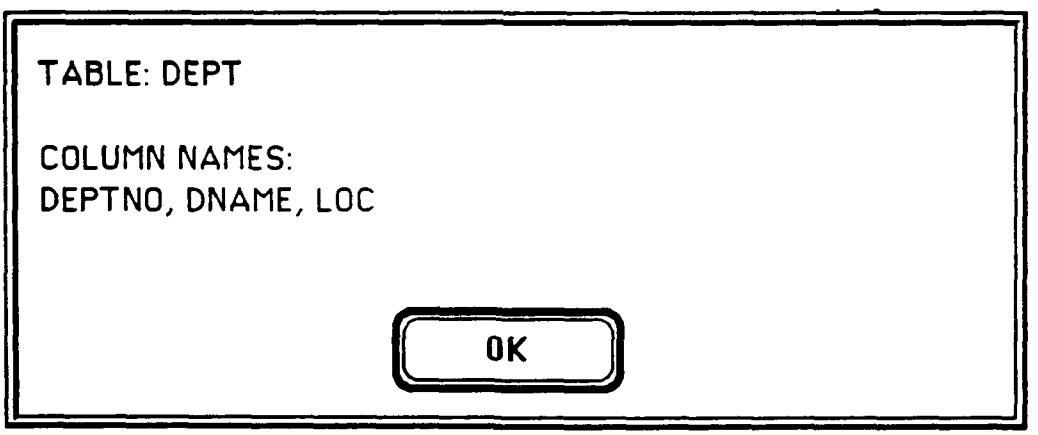



**Figure III.16 Table Information**

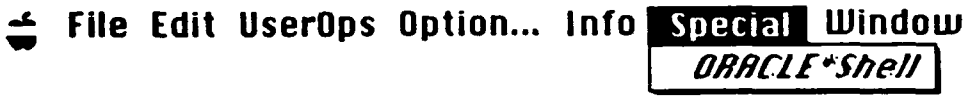



Figure III.17 Special Menu

pressed, all the windows will be relocated as the example shown in Figure III.19 so the banners of all windows can be seen. This feature allows the user to easily activate a completely overlaid window.

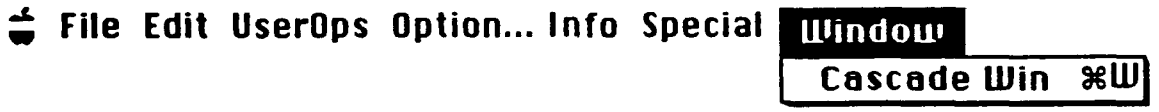



Figure III.18 Window Menu

## E. HELP WINDOW

There are two ways to open the Help Window as we described before: double-clicking on an operator displaying in the drawing area and selecting menu item Help from menu Info. Upon opening Help Window, if we want to view the descriptions

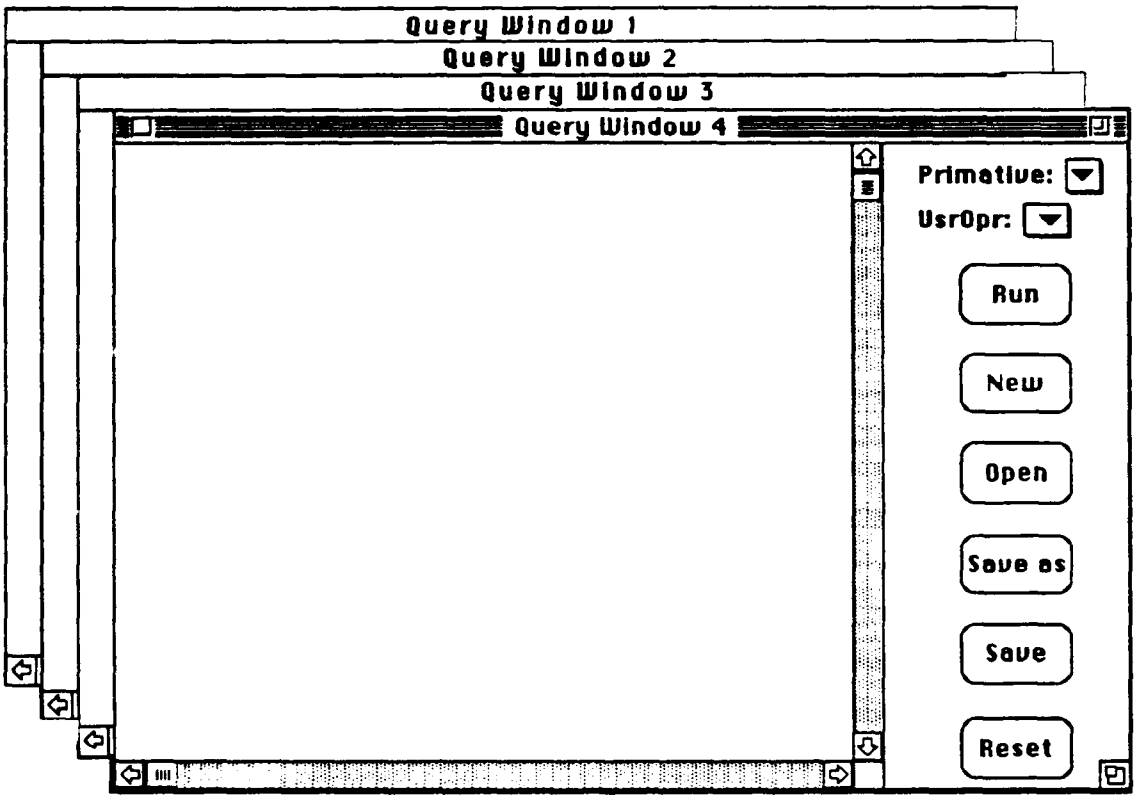



**Figure III.19 Cascaded Windows**

of any other specific operator in the operator list, we can simply click that operator in the scrolling list, the description of that selected operator will be provided on the right. This feature agree with **Interface Principle 1: Be able to provide more information when asked**, **Interface Principle 2: be able to display multiple information at the same time** and **Interface Principle 4: Be able to perform the**

**same operation in more than one ways**. In fact in order to open the Help Window, we do not even need to double click on any operator. We can simply select the menu item Help from menu Info whenever we need to consult the help message. Once the Help Window is opened, we can view the description of any existing operator.

It is possible for the user to find that the description of user-defined operators in the Help Window need to be modified after the operator is defined. To modify the descriptions, the user can double-click the specific operator whose description is to be modified to open an editor and enter what description appropriate. Upon completion of the modification, clicking the **OK** button will save the new description of that operator. This feature supports **Interface Principle 3: Be able to recover from the unintended or erroneous operation**. Since the primitive operators are basic set of operators, DFQL does not allow their descriptions to be modified. So double-clicking on the primitive operators cause nothing happened. **Interface Principle 6: Be able to prevent modifications that are not supposed to be made** is honored by this feature.

# IV. PROGRAPH AND OBJECT-ORIENTED PROGRAMMING

## A. LANGUAGE -- PROGRAPH

Since the previous DFQL interpreter was implemented in Prograph of version 2,02, to make things consistent and to import some of the previous code into our new implementation, we decide to use Prograph as our programming tool. Another reason to use Prograph is that, as stated in (TGSS, Tutorial, 1990, chapt. 1), Prograph integrates four key trends emerging in computer science:

- Visual programming.
- Object-oriented programming.
- Supporting dataflow specification of program execution.
- Providing application building toolkit.

### 1. Visual Programming

Contrary to text-based programming, visual programming uses graphical operations to accomplish the programming task. Figure IV. 1 shows the definition of methods **newPerson**, **setAttributes** and **introduce** for class **Person**. In method **newPerson**, the hexagon-shaped operation creates an instance of a specified class object (in this case a **Person**). This newly created **Person** then flows into the operation named **setAttributes** which is a method defined in class **Person**, After attributes being set by the method **setAttributes**, the **Person** with his attributes set flows into the operation named **introduce**, which is also a method defined in class **Person** to have the new person self-introduced. Method **setAttributes** asks the user to enter the person's name and where the person is from, then assigns those two

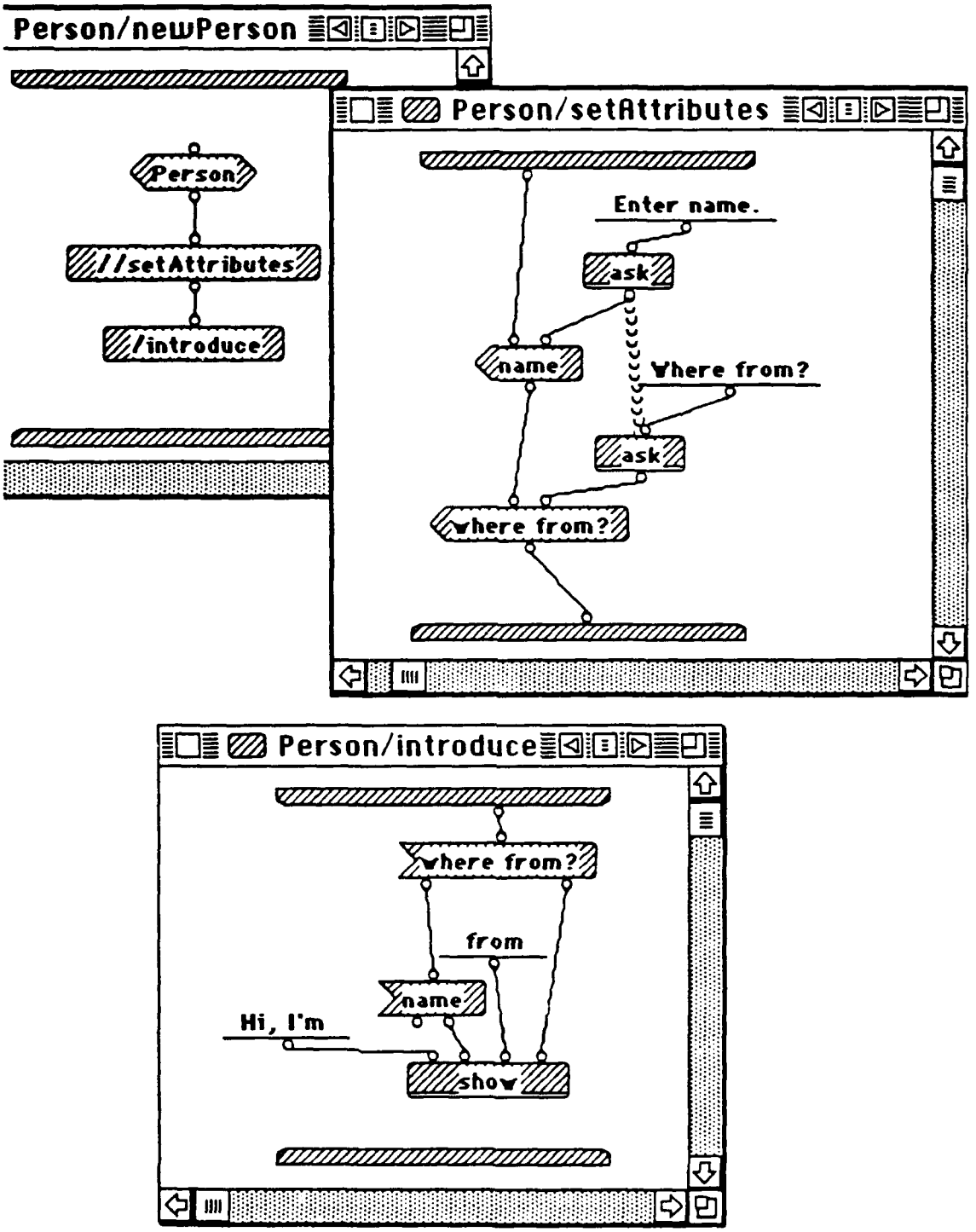



Figure IV.1 Methods

personal data from the user as the values of the new person's attributes, **name** and **where from?**. The new person introduces himself in method **introduce**,

There are four ways of method reference in Prograph: The form of *context-determined method reference* is "**//method**". This reference indicates that the named method can be found in the same class as the method containing (calling) this named method. In method newPerson, Figure IV. 1, method **setAttributes** is referenced in this form. The form of *explicit-class method reference* is "**classname/method**". Method **introduce** is referenced in this form. The third form, *Universal method reference*, is "**method**". The containing method looks for the method in the Universal methods pool. The last form is *data-determined method reference*, "**/ method**". Prograph looks for the method in the class to which the object arriving at the first input terminal of the method belong. This powerful method referencing form supports **polymorphism** of object-oriented programming and late binding. Different classes can each have a method of the same name with totally different function. When a method is referenced in this form, Prograph does not know from which class the specified method can be reference until the class object arrives on the input terminal of the specified method at runtime. That is why we say that it supports late-binding.

## 2. Object-Oriented Programming

Prograph provides a good foundation for object-oriented programing, system classes. Each class has two components: **attributes** and **methods**. **Attributes** identify instances of classes. By passing **messages (calling methods)** to objects (instance of class), we can make them act accordingly. In prograph, objects of same categories are group into classes, that is to say that a class is an abstract

description of a collection to similar objects. In object-oriented programming, objects communicate by passing messages. The situation is similar in Prograph but objects flow into operations to initiate some actions, rather than stationary objects sending messages back and forth (TGSS, Tutorial, 1990, chapt. 4). Figure IV. 2 shows a class window displaying the classes hierarchy, the attributes and methods window of class **Person**. In Figure IV. 1, we have shown one way to assign attribute values and to pass message asking the person to introduce himself/herself. The line between classes represent inheritances. For example, classes **Student** and **Instructor** are both descendants of class **Person** They both inherit all the attributes and behaviors (methods) of their ancestor **Person**. But they can have their own additional attributes and methods (Figure IV. 3). In attribute window, Figure IV.3, there is a downward arrow in each inherited attribute icon. Method overriding is also allowed in Prograph. For example class **Instructor** has its own method **introduce** whose functionalities can be totally different from **Person**'s.When an instance of **Instructor** is "asked" to introduce himself, instead of referencing the method **introduce** in class **Person**, the one in class **Instructor** is to be referenced. But since class **Student**, has no its own method **introduce**, so when an instance of class **Student** is "asked" to introduce herself, the inherited method **introduce** defined in its parent class **Person** is to be referenced. Programmers can also make their created classes descendants of system classes so a great deal of effort can be saved.

## 3. Dataflow Programming

In Prograph, each operation can have zero or more input *terminals* and zero or more output *roots* representing by small circles attaching to the top or bottom of operations (Figure IV.4). Data flow along arcs connecting terminals and roots. When

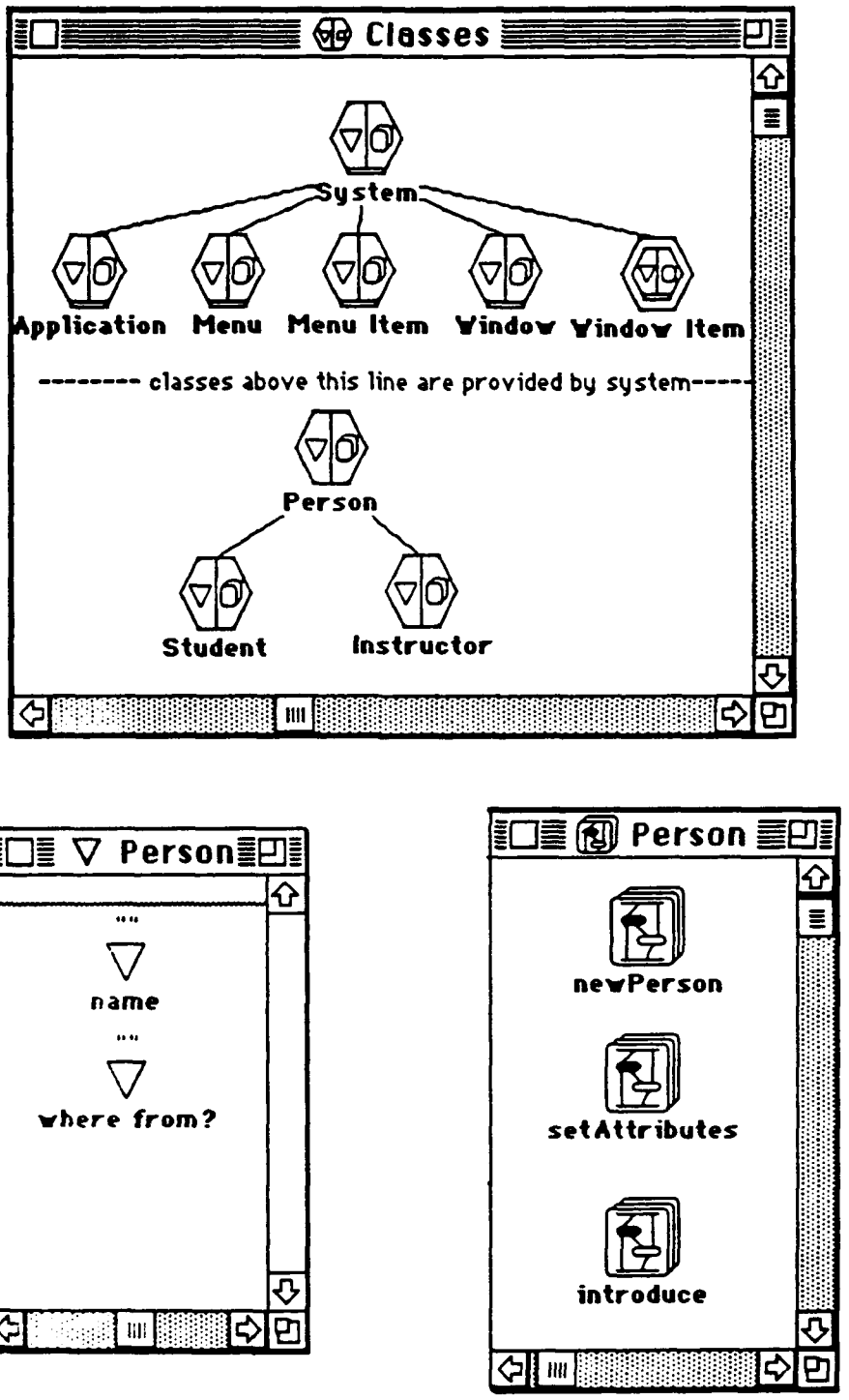



Figure IV.2 Class Hierarchy and Class Components

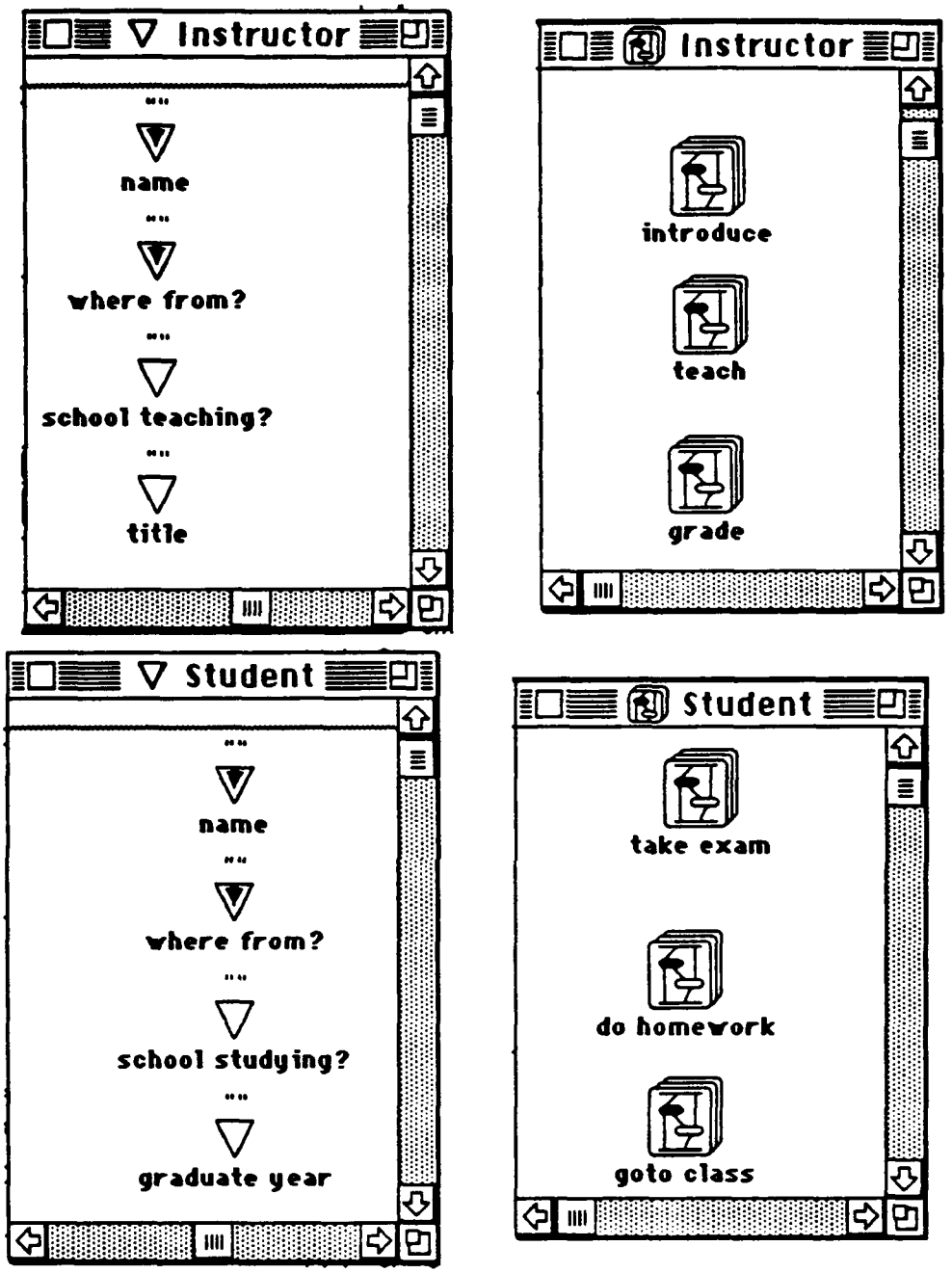



Figure IV.3 Classes Instructor and Student

all required input data become available at an operation's terminals, the operation can not execute or fire. Specific order of execution is not prescribed in Prograph programs. This situation implies that concurrent execution of several operations is likely to occur. To enforce the order of execution, synchros are provided as shown in Figure IV.4. In Figure IV.4, there is a direct arc connecting the root of **operation1** and one terminal of **operation2**, so **operation 2** must execute after the execution of **operation 1**. Same situation between **operation 1** and **operation 3**. But, since there are no direct arc between **operation 2** and **operation 3**, so they can execute whenever their required input data are available. Execution order of **operation 1** and **operation 2** is not prescribed without the synchro. With the synchro emerging in Figure IV.4, the execution of **operation 3** is enforced to wait until the execution of **operation 2** is done. In method **setAttributes**, Figure IV.1, we used a synchro to ensure that a person's name was asked before where he/she is from is.

By combining the dataflow specification of program execution and some powerful debug facilities, Prograph allows programmers to debug programs by inspecting the data flowing in the programs from debug mode. With the inspection of the dataflow, programers can see how programs work and why it does not work the way as expected if there are some errors in the programs.

## 4. Application Building Toolkit

The application building toolkit in Prograph supports independent specification and management of the interface through high-level tools and facilities. For example, the **application**, **menu**, **window**, and several **window item** editors are some powerful and easy to use tools provided in Prograph. **application** editor (Figure IV.5) serves as a library of menus and windows. The **menu** editor

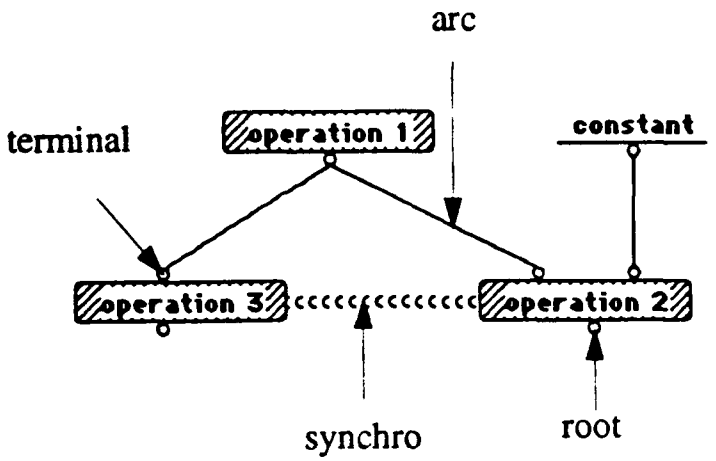



**Figure IV.4 Terminals, Roots, Arcs and Synchro**

(Figure IV.6), **window** editor (Figure IV.7), and **window item** editors (Figure IV.8) are used to specify the "look and feel" of the menus and windows created and organized using the **application** editor. So instead of developing the windows, menus, and their items we need in the DFQL user-interface from scratch, we can just specify some essential informations in each relevant editor and the application builder will creates what we need according to the informations we specified. To refine any portion of them, we just return to the relevant editor and respecify some relevant informations. No recoding and recompilation needed.

Since there are some new features and facilities added to the 2,02 version, so our program is implemented in Prograph of version 2.5 (e.g., the **pop-up menu**

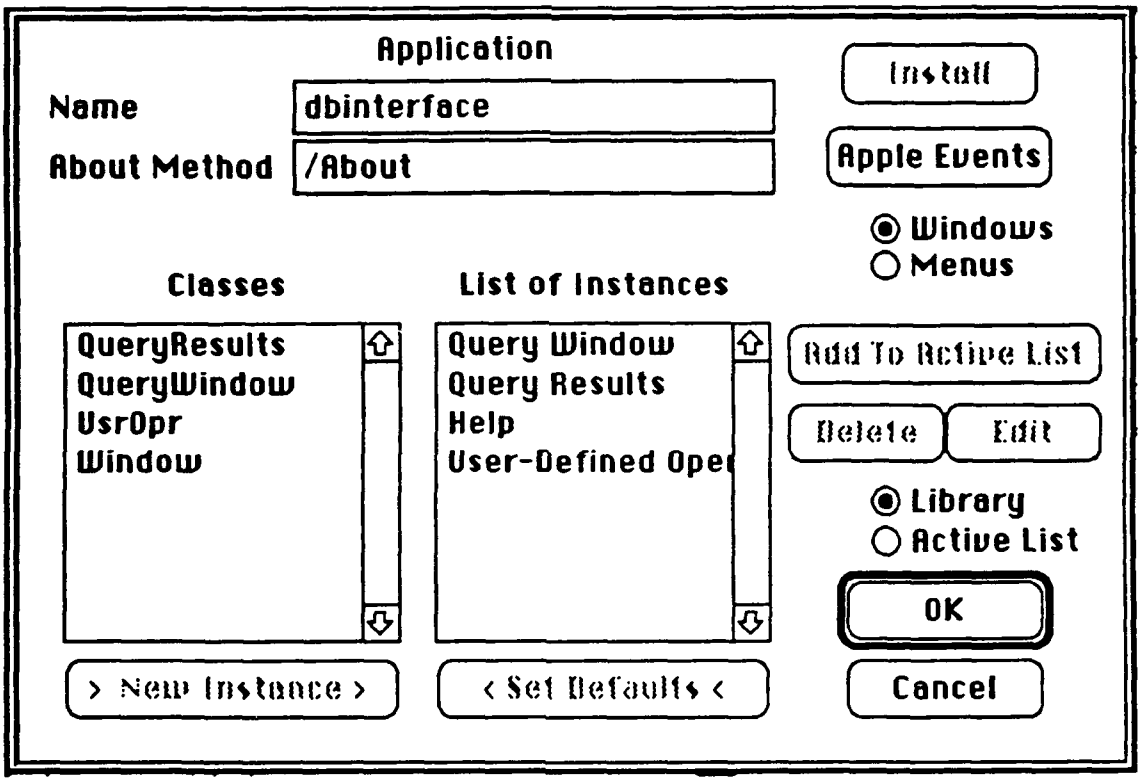



**Figure IV.5 application Editor**

we used in the **Query Window** is a new added facility to version 2.5). We do not intend to get into the details of Prograph. The following references by The Gunakara Sun Systems Limited (TGSS) are recommended for those who are interested in.

- Prograph: Tutorial, second printing, 1990.
- Prograph: Reference, second printing, 1990.
- Prograph: 2.5 Updates, first printing, 1991.

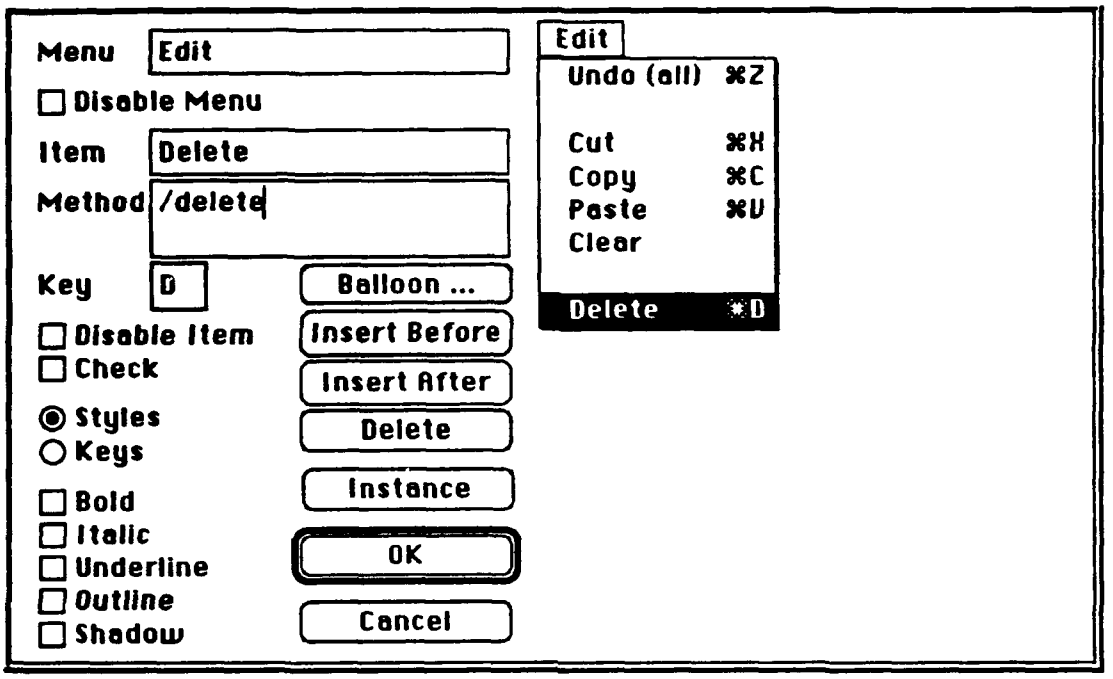



Figure IV.6 menu Editor

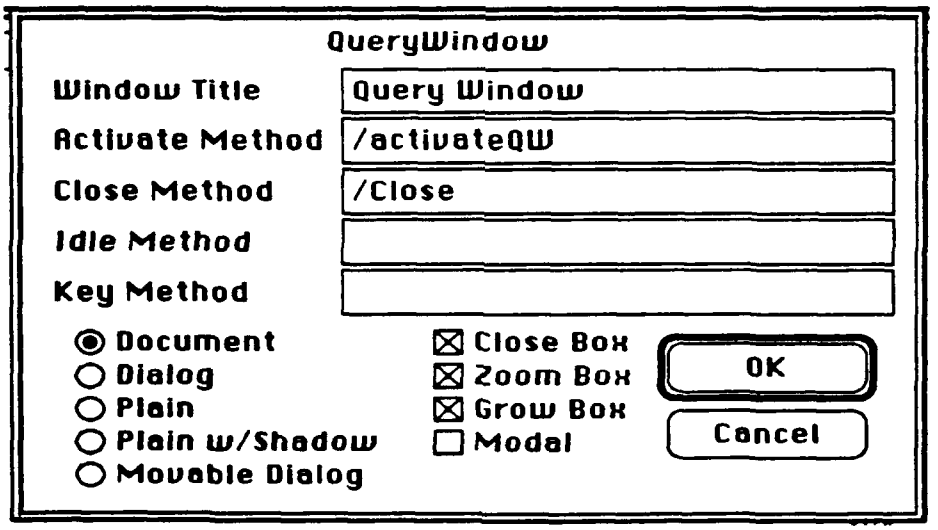



Figure IV.7 window Editor

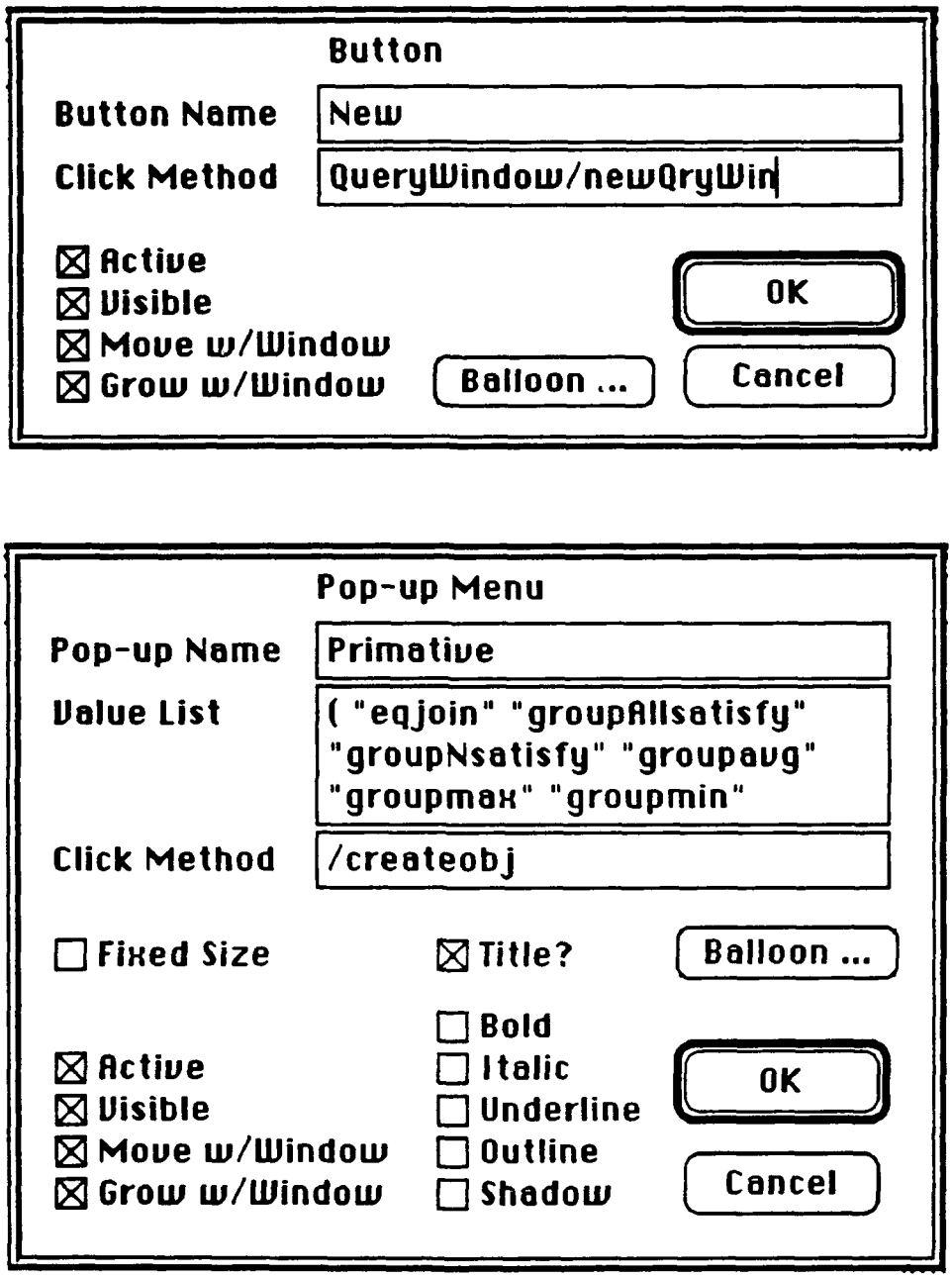



**Figure IV.8 window item Editors for Button and Pop-Up Menu**

## B. WHY OBJECT-ORIENTED PROGRAMMING

Object-oriented programming is often referred to as a new programming paradigm (Budd, 1991). In Clark's DFQL interpreter, object-oriented approach was attempted. But after a careful examination of Clark's program, we found that some major object-oriented programming features, such as responsibility driven, information hiding, and modularity do not play their role. And the program is also tightly interconnected by heavily using the *explicit-class method reference*.These drawbacks restrict the reusability of his program, so in order to use some methods in his program we spent some significant effort to modify and re-modularize his original code. Because of this experience we decide to approach our new development from object-oriented view so that our code may be more reusable, portable, and maintainable.

During the examination and modification of Clark's program, we also found many attractive features of OOP can be adopted to model our new DFQL interface. In our view, DFQL interface consists of objects like Query Window, drawing area (a canvas in which we draw our query) and operators, etc. These objects are manipulated in response to events, such as the clicking or dragging on the body of a DFQL object cause the object being selected or dragged. This awareness also encouraged us to do experiment of object-oriented programming.

## C. Evaluation Of Object-Oriented Programming

In this section, we are going to discuss some features of object-oriented programming we applied to our new development of DFQL interface and some benefits we have gotten through the whole design process.

## 1. Benefits of Responsibilities-Driven, Class, and Inheritance

To organize or discover classes, we need to take the responsibilities of each potential class into consideration. Clear assignments of responsibilities increase the degree of independence of classes thus increase the degree of information hiding and reusability. When we make an object responsible for a specific action, we can expect a certain behavior the object is to behave as we passing the request to it. The higher the degree of behaviors of objects we can expect, the easier we can debug our program when errors occur and the easier to expend our program by assigning more appropriate responsibilities to appropriate classes without causing intensive modification to the program has been existing or importing unexpected interferences between new code and old code.

Figure IV.9 shows the classes hierarchy of our program. The white-colored classes are system-provided while the black-colored classes are our created ones. Each class representing one category of objects encapsulates informations and functionalities within its attributes and methods. This classes hierarchy is organized after the following analysis: Class **Query Window** represents the main window (also named **Query Window**) with the responsibilities of detecting and handling the mouse click events on its item such as button close box, etc., creating new Query Windows, setting attributes of new created Query Windows, loading query files onto Query Windows saving query files to disks and closing Query Windows, etc., Class **Query Object** represents all query objects. Classes **Text Object** and **Operator** represent text objects and operators respectively. Because they are both query objects so we make them the descendants of class **Query Object** to inherit the attributes and methods we defined in class **Query Object**. Since operators can also

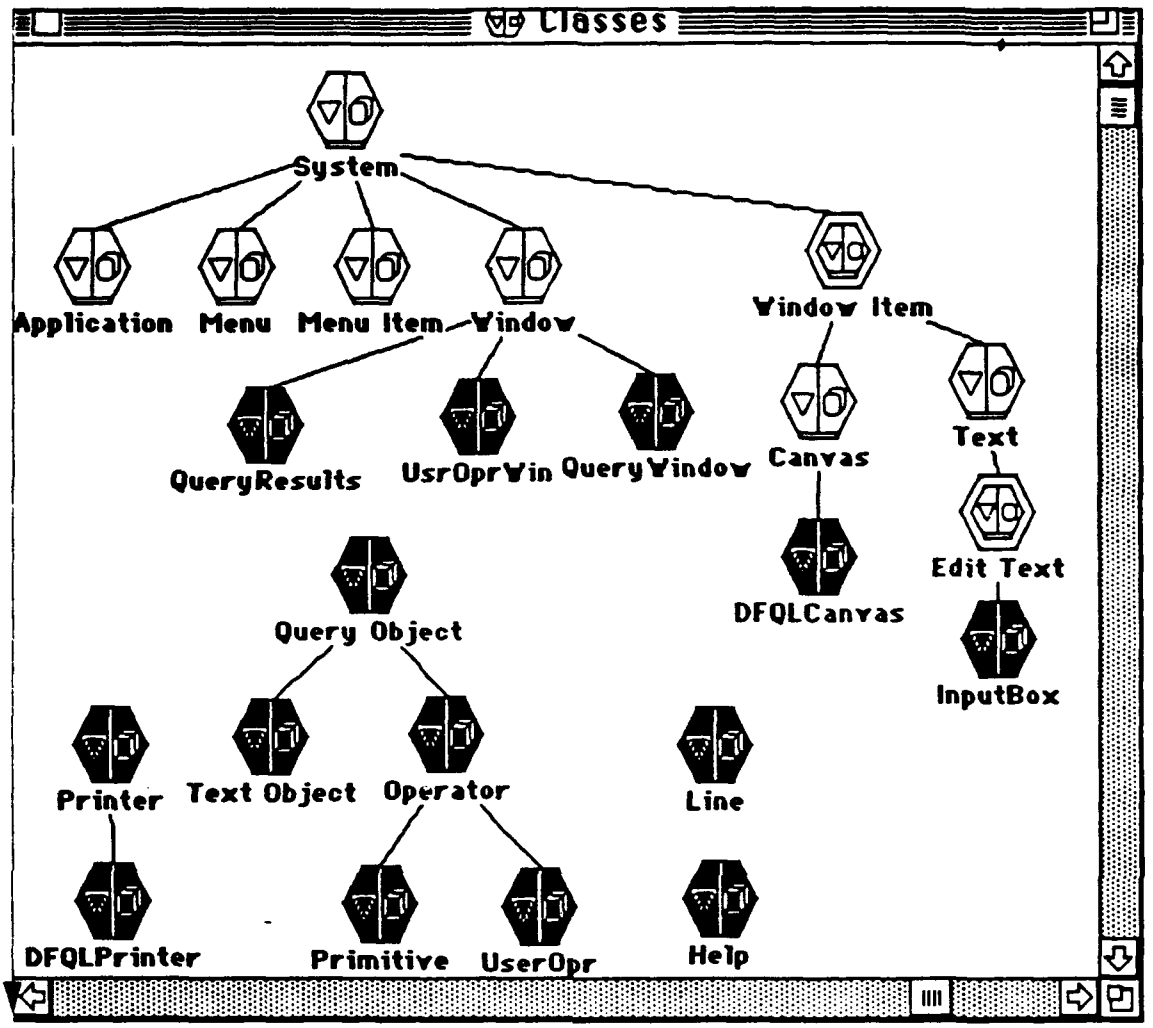



**Figure IV.9 Classes Hierarchy**

be grouped into primitive operators and user-defined operators, so we make classes **Primitive** and **UserOpr** the descendants of class **Operator** representing primitive operators and user-defined operators respectively. Class **Query Object** and its descendant classes are responsible for maintaining and creating query objects. Class **DFQLCanvas** represents the drawing area (the drawing area just like a canvas) within which we "draw" our query. It is responsible for detecting and handling all

kind of mouse click events in the drawing area and acts accordingly, drawing objects, dragging objects, and deleting objects. Class **InputBox** represents the "input box" within which users type in the text for the later creation of query objects. It is responsible for displaying the "inputbox" at where users click the mouse and reading text that users type in from the keyboard. Classes **QueryResults**, **UsrOprWin**, **Printer** and its descendant **DFQLPrinter**, **Line**, and **Help** are created by following the similar analysis.

Assuming the user wants to create an operator by selecting one from either pop-up menus **Primitive** or **UsrOpr**, then when the event of clicking mouse in a pop-up menu and selecting one operator from the menu is detected by **Query Window**, **Query Window** passes message requesting the creation of an operator with the name specified to class **DFQLCanvas**. Upon receiving the request, **DFQLCanvas** does some prelimilary actions and then passes the same request received from **Query Window** to class **Primitive**, **Primitive** then check its primitive operator list whether there is one with the same name passed in or not. If there is one, **Primitive** returns the matched operator back to **DFQLCanvas.** If there is none matched, the same message is delivered to class **UserOpr** to create an user-defined operator. After the operator is returned, **DFQLCanvas** draws it in the drawing area. The ellipses in Figure IV.10 highlight the process of message passing.

We make each class an encapsulation of abstractions having two faces. From the outside, a user of that abstract encapsulation s[illegible] a collection of methods which define the behavior of the encapsulation [illegible] actions. On the internal side, attributes (some people use the term a[illegible] *iables*, but for consistency, we will use *attributes* through this thesis) ar[illegible] to maintain the

internal state of the object. In the above example, by using responsibility-driven, we do not allow classes to interfere or modify attribute values of other classes because we view changing attribute values as the internal affair and all of them must be maintained internally by the class instance itself under any circumstances. This measure makes a program less error-prone and ease the maintenance of a program. Also, the information hiding rule is observed by the careful responsibility assignments. For example, **DFQLCanvas** issues requests of query object creation to classes **Primitive** without knowing how they are created.

The clear division of classes helps us design a well structured and modularized program. The hierarchical organization also helps us to remember the whole structure of our program and the relationship between classes easier and better. Since many methods are inherited by the descendant classes from their ancestor classes, we do not need to replicate codes over and over again. This feature shortens the development time and supports information hiding also. *Figure IV.10* shows an example of inheritance. In Figure IV.10, when either class **Primitive** or class **UserOpr** receives the message requesting the creation of an object from class **DFQLCanvas**, since there is no method capable of creating objects defined in both classes **Primitive** and **UserOpr**, the method **create** defined in their parent class **Operator** will be referenced as if it is defined in their own classes. This benefits is more significant in the inheritances of our created classes from system provided classes. For example, by making classes **Query Window**, **QueryResults**, and **UsrOprWin** as descendants of system provided class **Window**, we can simply use the methods **Create**, **Activate**, and **Idle** defined in class **Window** without knowing how they are implemented.

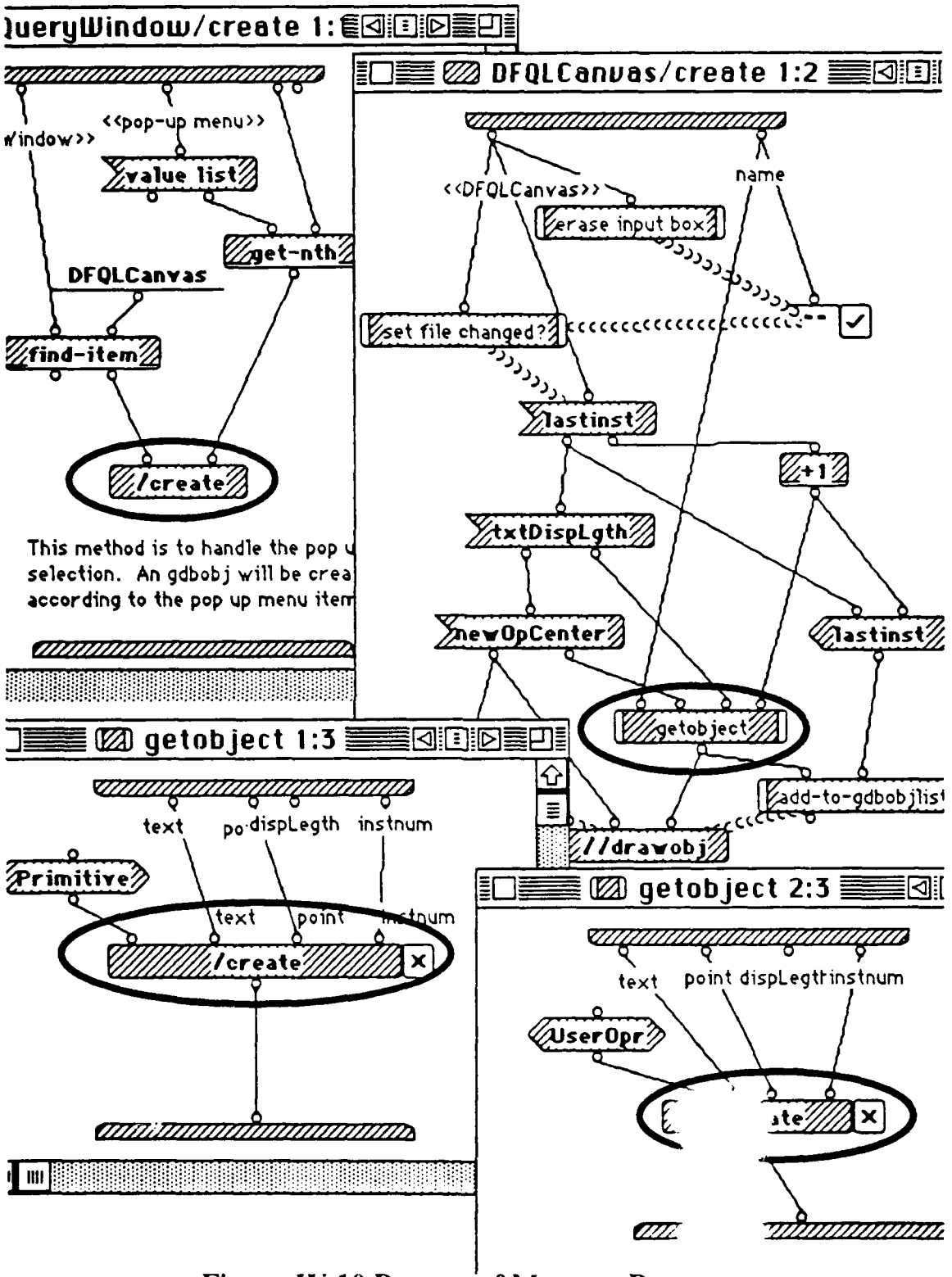
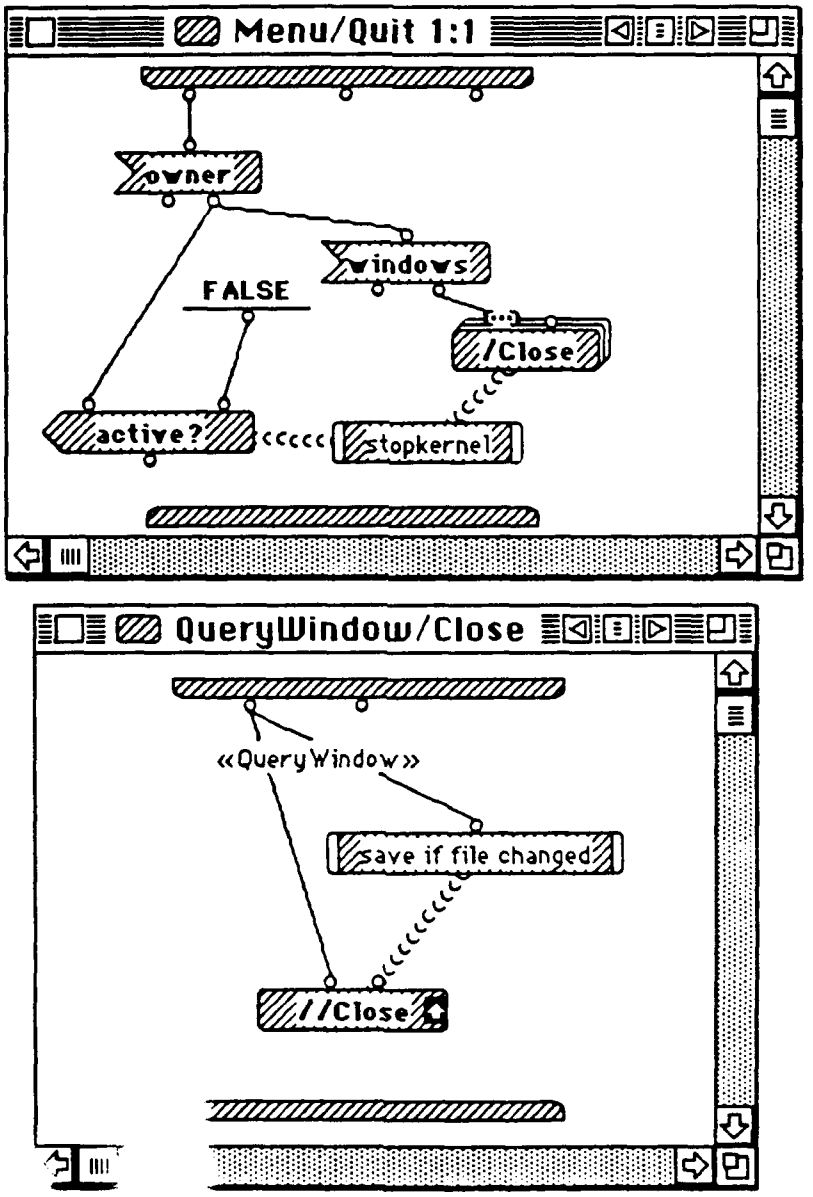



Figure IV.10 Process of Message Passing

## 2. Benefits Polymorphism and Late Binding

Polymorphism is an feature to send the same message to instances of different classes. It enhances the readability of software and leads to an easier extension of code. Figure IV.11shows an example of Polymorphism. When we




Fi IV.11 Example of Polymorphism

choose menu item **Quit** from menu **File**, method **Quit** defined in class **Menu** is called. When this method executes, a list of windows opened flows from the right output terminal of operation windows to method **Close** which is of the *Data-determined method reference* form. As we mentioned in last section, all windows are viewed as instances of different classes, so when the message *Close* is sent to them, they will behave according to the **Close** method defined in their class or ancestor class. Because all windows are subclasses of system class **Window**, so except **Query Window** which need to save the query file before closed if the content of the query file has been changed), all window will close by calling method **Close** defined in class **Window**. The method **Close** defined in class **Query Window** as shown in Figure IV.11 overrides the method **Close** in the parent class **Window** by having the same name and different behavior. It first save the query file which has been changed if the user desired and then calls the method **Close** defined in class **Window** (the upward arrow means method **Close** in the supper or parent class is to be called). In this example, we see the same message *Close* sending to instances of different classes causes them to behave differently. We also see that *late binding* utilized in this example. The proper methods **Close** to be referenced is unknown until the instances arrive (i.e., until run-tome). This makes the high-level software design more flexible.

The overriding we saw in last example is just in form of polymorphism. Another form is *overloading* which makes the flow of software execution easier to be followed by users. Figure IV.10 shows how the methods defined in classes **Query Window**, **DFQLCanvas**, and **Operator** are refere e overload the

method name **create** so the way an operator is created easier to be understood and the code is also easier to be read.

# V. CONCLUSION

## A. LESSON LEARNED

some lessons we learned in our development of this new DFQL interface can be stress in two aspects: user interface and object-oriented programming.

### 1. User Interface Aspect

As an application designer, we need to originate all the ideas from the users' point of view. So human factors need to be taken into consideration. Our experience is that some established principles like those we listed in Chapter I need to be followed all the time. We also evaluate some human factors such as

- *How long does it take a regular user to learn this new application.* We need to provide users a consistent, well-explained, and easy-to-use, and easy-to learn tool so they will not get confused or frustrated by the complexities of the tool.
- *What kind of errors may be made by users.* Taking this factor into consideration leads to a design of careless-proof or even fool-proof application.
- *What degree do users maintain their knowledge about the application after a period of time.* To increase the degree, the consistency must be enforced and on-line help messages must always be available.

### 2. Object-Oriented Programming Aspect

The following is a list of lessons we learned from the object-oriented programming aspect:

- *Each class is responsible for one task or s* *tuitively similar tasks and allow attribute values to be modifi* *intained only in its own class.* This reduces the degree of *nection between*

classes and makes each class cohesive and modular so later modification and extension is easier.

- *Avoid writing long methods.* Try dividing a task into several subtasks which can be implemented as clearly defined methods. This also makes later modification and extension easier and increases the reusability.

- *Properly use polymorphism to make program easy-to-read.* This makes programs more symmetric and easier to be understood.

- *Avoid using explicit-class method reference.* Explicit-class method reference makes methods tightly interconnected so when a class name is changed, we have to change all references in the whole program. This is error prone and time consuming.

- *Properly use abstract superclass to support information hiding and inheritance.* We discussed many benefits of information hiding and inheritance. Proper use of superclass allows the programer to concentrates on the further development without paying attention to the low-level details of the referenced methods.

## B. SUMMARY

This thesis provides an improved user interface of DFQL originally introduced by Gard J. Clark and C. Thomas Wu in 1990. In this thesis, we eliminated the shortcomings of user interface pointed out in their paper and added some new features to it. Now, this new DFQL user interface allows users to create DFQL objects simply by clicking the mouse and then entering the text right on the computer screen. Some tedious operations in the previous version of DFQL such as deleting an object, selecting an user-defined operator have been simplified in our new implementation. We also make on-line help messages easy to get and allow information display window such as **Help** window, **Query Results** window and **User-Defined Operator** window to co-exist at the same time when users are constructing their queries in **Query Window**. This makes more reference

information available at the same time. We also allow users to open **Query Windows** as many as they want and define user-defined operators consecutively. The content of help messages of user-defined operators entered by users can also be modified easily.

In addition to the new features we added to the DFQL user interface, we also did an object-oriented programming experiment on this new implementation. The main techniques of object-oriented programming such as message passing, class, responsibility-driven, inheritance, and polymorphism are used. The benefits of these techniques are also evaluated.

# LIST OF REFERENCES


Angelaccio, M., Catarci, T., and Santucci, G., *QBD*: A graphical Query Language with Recursion*, IEEE Transactions on Software Engineering, v. 16, p. 1150-1163, October 1990.

Apple Computer, Inc., *Inside Macintosh*, v. 1, Addison-Wesley, 1985.

Budd, T., *An Introduction to Object-Oriented Programming, Addison-Wesley, 1991.*

Clark, G. J., and Wu, C. T., *DFQL: Dataflow Query Language*, Submitted for publication, 1991.

Codd, E. F., *Fatal Flaws in SQL: Part I*, Datamation, v. 34, pp. 45-48, 15 August 1988.

Codd, E. F., *Fatal Flaws in SQL: Part II*, Datamation, v. 34, pp. 71-74, 1 September 1988.

Codd, E. F., *The Relational Model for Database Management: Version 2*, Addison-Wesley, 1990.

Miyao, J., and others, *Design of a High Level Query Language for End Users*, paper presented at the 1986 IEEE Workshop on Language for Automation, National University of Singapore, Kent Ridge, Singapore, 27-29 August 1986.

Sockut, G. H., et. al., *GRAQULA: A Graphical Query Language for Entity-Relationship or Relational Database*, IBM Research Report RC16877 (#73833), 14 March 1991.

TGSS (The Gunakara Sun System Limited), *PROGRAPH: Tutorial*, second printing, 1990.

TGSS (The Gunakara Sun System Limited), *PROGRAPH: Reference*, second printing, 1990.

TGSS (The Gunakara Sun System Limited), *PROGRAPH: 2.5 Updates*, first printing, 1991.

Wong, H. K. T., and Kuo *'IDE: Graphical User Interface for Database Exploration,* Proceeding of the Eighth ıational Conference on Very Large Database, pp. 22-32, September 1982.

Wu, C. T., *A new grapi .er interface for accessing a database,* Proceedings of Computer Graphics, pp. 2( 9, Tokyo, 1986.

Wu, C. T., *Development of a Visual Database Interface: An Object-Oriented Approach,* Application of Object-Oriented Programming, Pinson, L. J. and Wiener R. S., Addison-Wesley, 1990.

Wu, C. T., *GLAD: Graphics LAnguage for Database*, Proceeding of the 11th International Computer Software and Application Conference, Tokyo, Japan, October, 1987, 164-170.

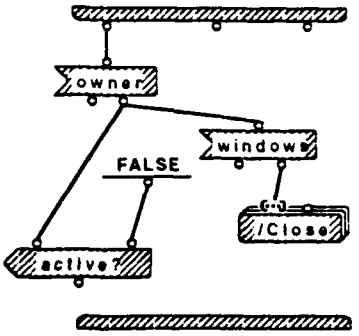
Wu, C. T., *Lecture Note for CS3320 Introduction to Database System,* Naval Postgraduate School, 1991.

Zloof, M. M., *Query-By-Example: A data Base Language,* IBM Systems Journal, v. 16, pp. 324-343, 1977.

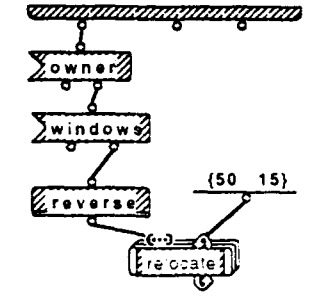
# APPENDIX

Only the source codes we developed are presented in this thesis. The original codes of system provided classes are not printed.

## Menu/Quit 1:1

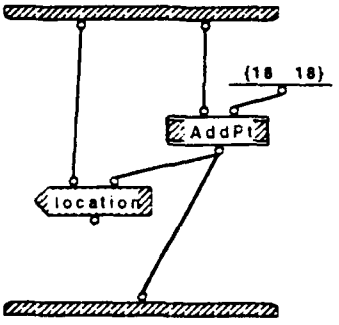



## Menu/cascade win 1:1

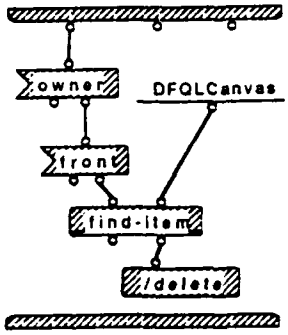



Cascade all windows.

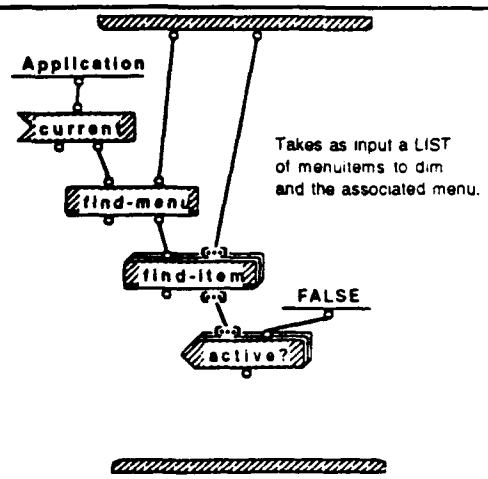
## Menu/cascade win 1:1relocate 1:1



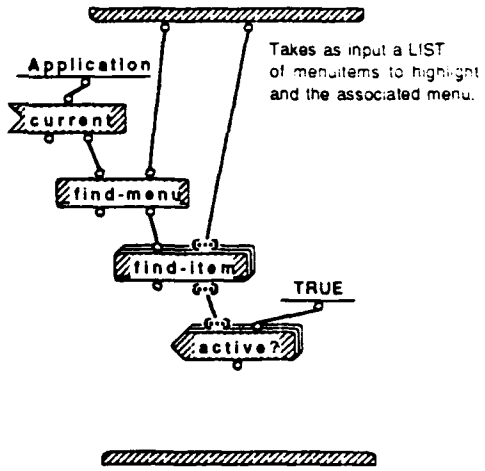
## Menu/delete 1:1



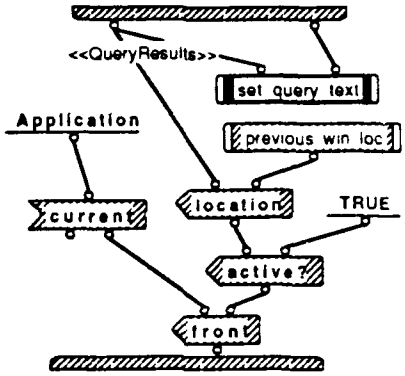
## Menu Item/dim 1:1



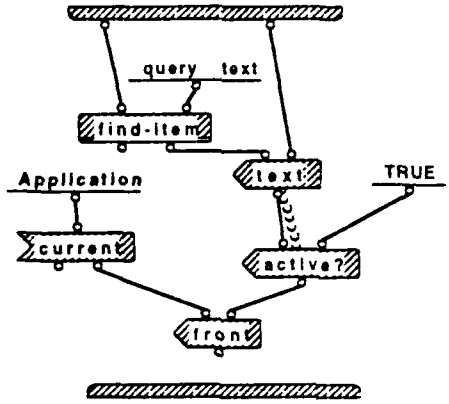

## Menu Item/highlight 1:1



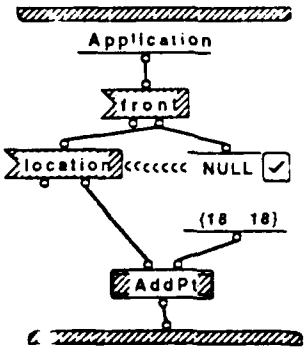
## QueryResults/showQueryResults 1:2



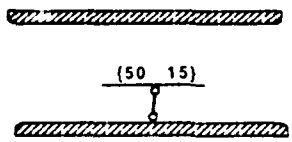

## QueryResults/showQueryResults 2:2

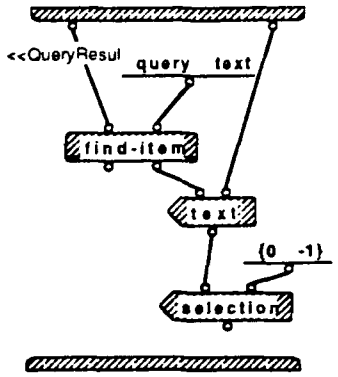



## QueryResults/showQueryResults 1:2previous win loc 1:2

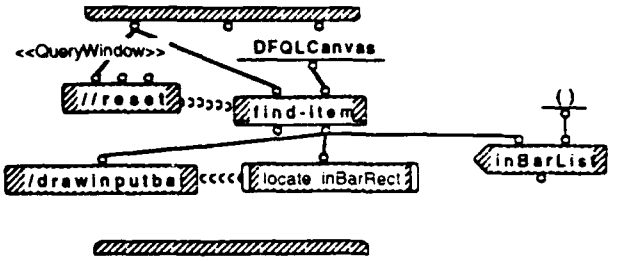



## QueryResults/showQueryResults 1:2previous win loc 2:2

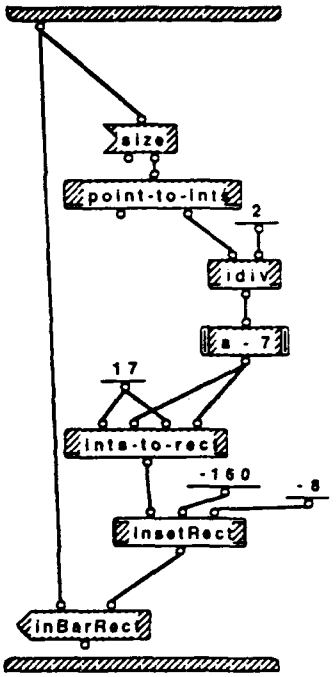



<<QueryResul
query
text
find-item
text
{0 -1}
selection

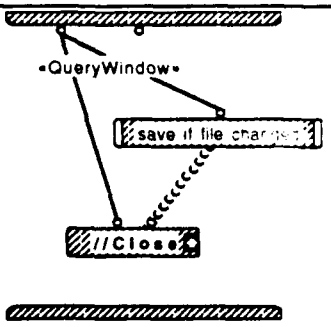
## QueryWindow/clear 1:1



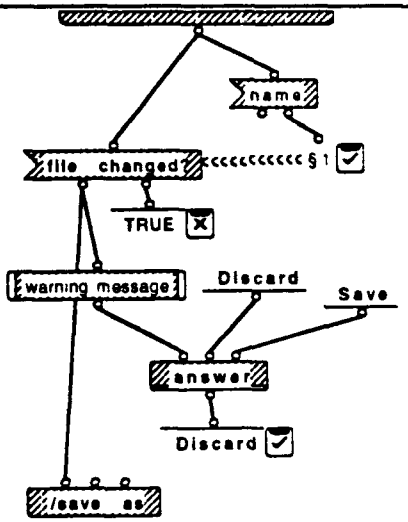

## QueryWindow/clear 1:1locate inBarRect 1:1




## QueryWindow/Close 1:1



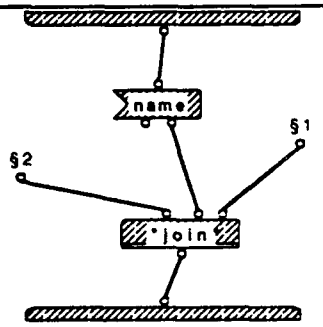
## QueryWindow/Close 1:1 save if file changed 1:1




*If the window to be closed is of type "QueryWindow" and the contents of the canvas in that window has been changed(i.e.,file changed) prompt the user to save it before closing the window.*

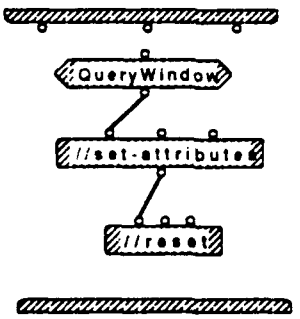
§1. Operator Definition Window

## QueryWindow/Close 1:1 save if file changed 1:1 warning message 1:1

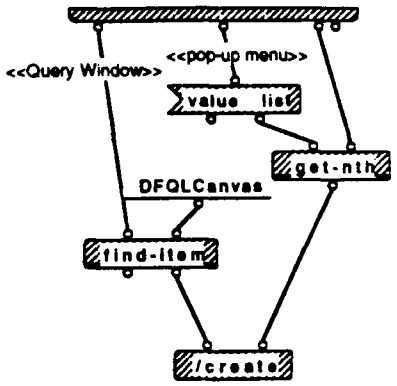



§1. has been changed. Save it?
§2. The content in

## QueryWindow/newQryWin 1:1

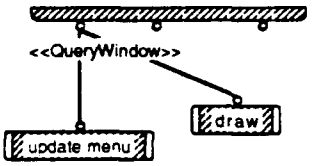


## QueryWindow/create 1:1



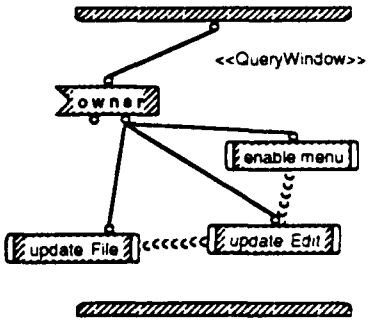

This method is to handle the pop up menu selection. An object will be created according to the pop up menu item selected.

## QueryWindow/activateQW 1:1

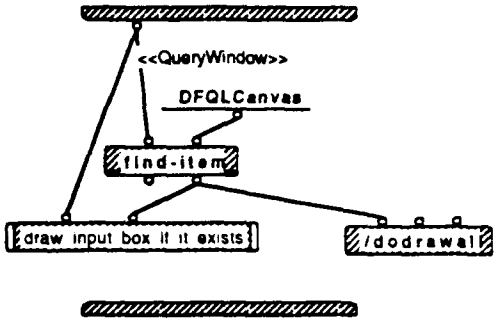



It's possible to activate a Query Window with other types of window being frontmost. So it's necessary to update menu.

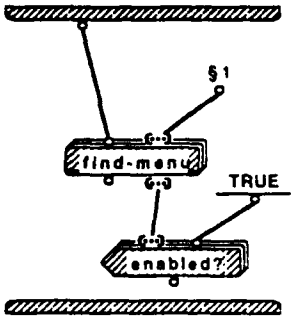
## QueryWindow/activateQW 1:1update menu 1:1



## QueryWindow/activateQW 1:1draw 1:1

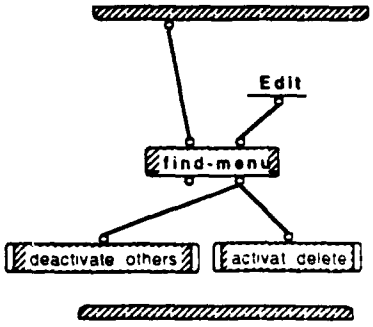



## QueryWindow/activateQW 1:1update menu 1:1enable menu 1:1

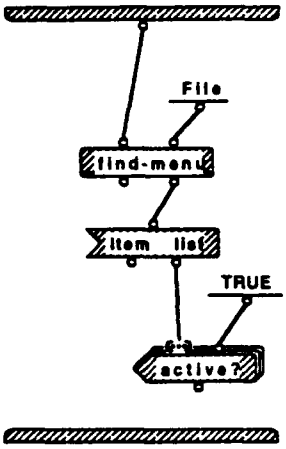



§1. (1 2 3 4 5 6)

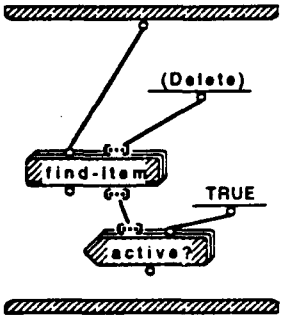
## QueryWindow/activateQW 1:1update menu 1:1update Edit 1:1



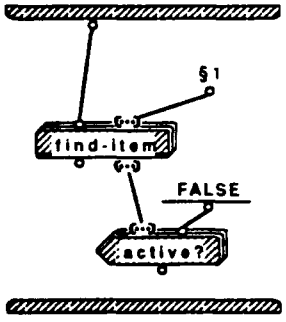
## QueryWindow/activateQW 1:1update menu 1:1update File 1:1




## QueryWindow/activateQW 1:1update menu 1:1update Edit 1:1activat delete 1:1

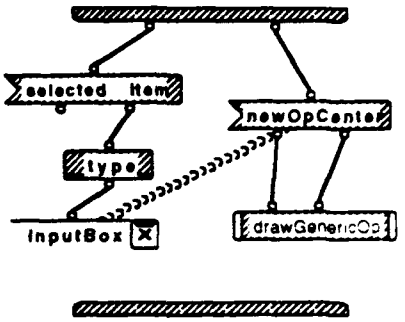



## QueryWindow/activateQW 1:1update menu 1:1update Edit 1:1deactivate others 1:1

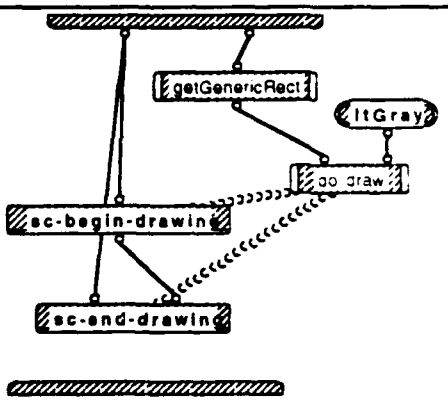



§1. ("Undo (all)" "Cut" "Copy" "Clear" "Paste")

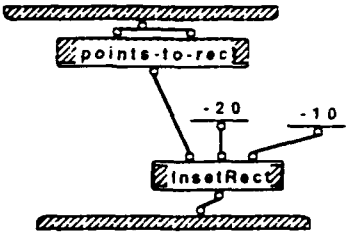
## QueryWindow/activateQW 1:1 draw 1:1 draw input box if it exists 1:1



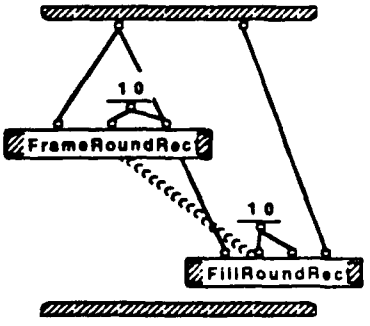

## QueryWindow/activateQW 1:1 draw 1:1 draw input box if it exists 1:1 drawGenericOp 1:1



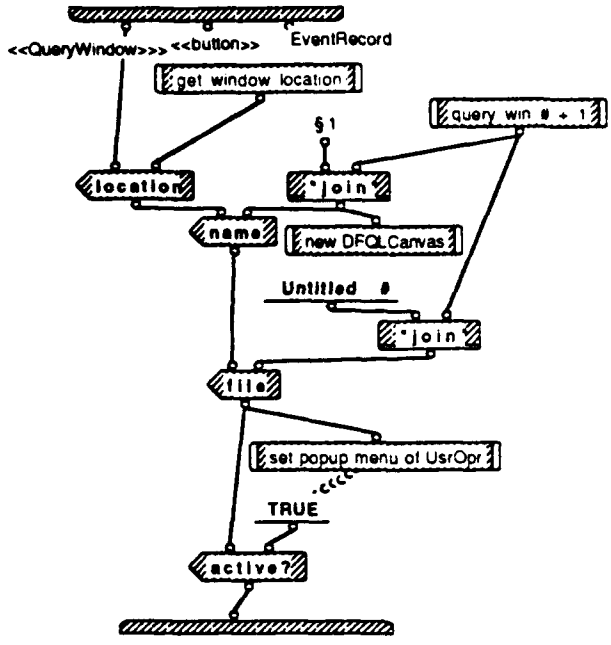

## QueryWindow/activateQW 1:1 draw 1:1 draw input box if it exists 1:1 drawGenericOp 1:1 getGenericRect 1:1




## QueryWindow/activateQW 1:1 draw 1:1 draw input box if it exists 1:1 drawGenericOp 1:1 do draw 1:1

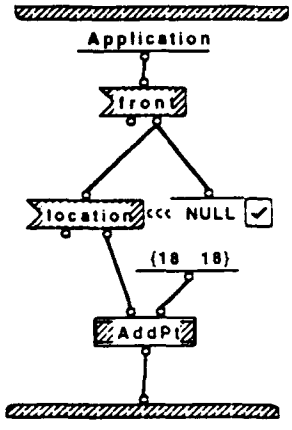


## QueryWindow/set-attributes 1:1

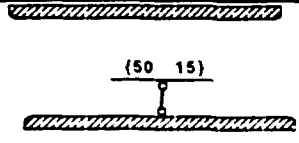



§1. Query Window

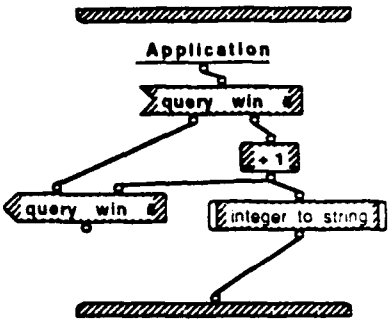
## QueryWindow/set-attributes 1:1get window location 1:2



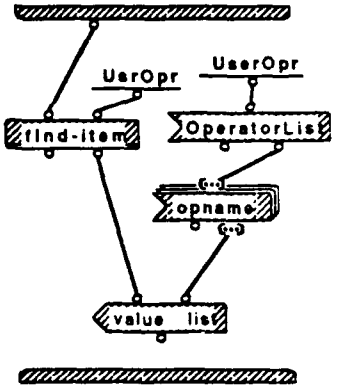

## QueryWindow/set-attributes 1:1get window location 2:2



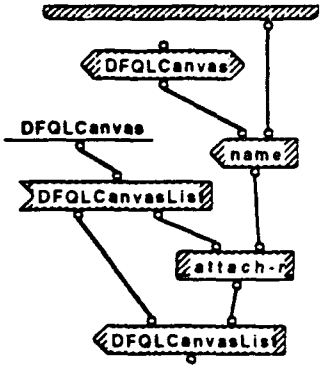
## QueryWindow/set-attributes 1:1query win # + 1 1:1




## QueryWindow/set-attributes 1:1set popup menu of UsrOpr 1:1

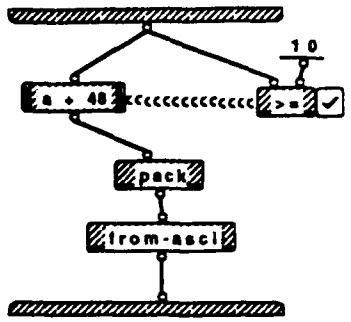



## QueryWindow/set-attributes 1:1new DFQLCanvas 1:1

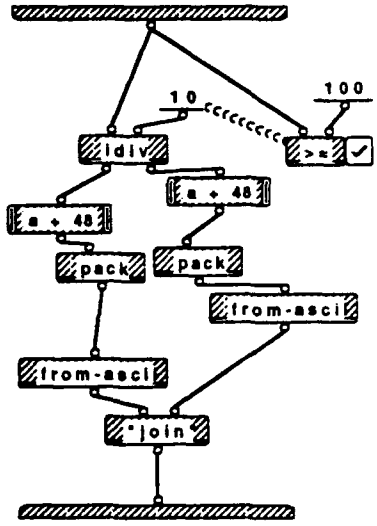



Create a new DFQLCanvas for each new QueryWindow.

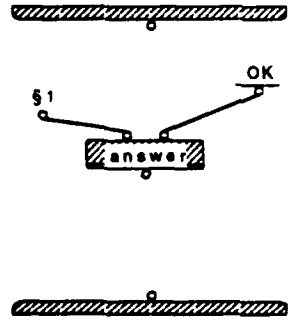
## QueryWindow/set-attributes 1:1 query win # + 1 1:1 integer to string 1:3




## QueryWindow/set-attributes 1:1 query win # + 1 1:1 integer to string 2:3

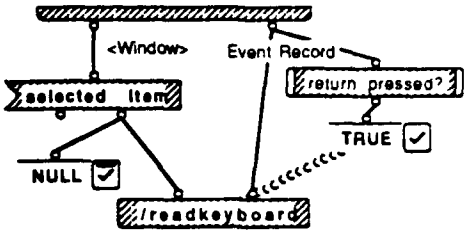



## QueryWindow/set-attributes 1:1 query win # + 1 1:1 integer to string 3:3



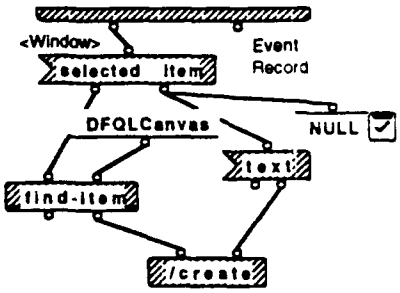

§1. You can't open so many windows.

## QueryWindow/Key 1:2



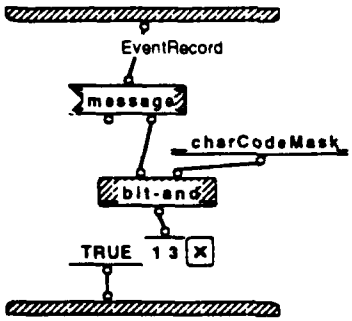

User should press "return" to finish typing.
So keep reading inputs from keyboard *until*
*"return" is pressed.*

## QueryWindow/Key 2:2

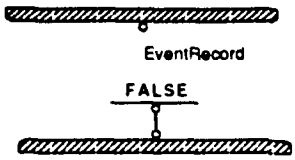



When "return" is pressed, create an gdbobj
according to the text the user typed in.

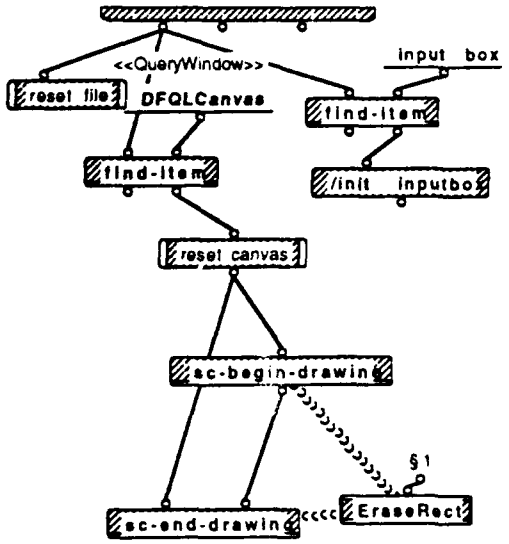
## QueryWindow/Key 1:2return pressed? 1:2




## QueryWindow/Key 1:2return pressed? 2:2



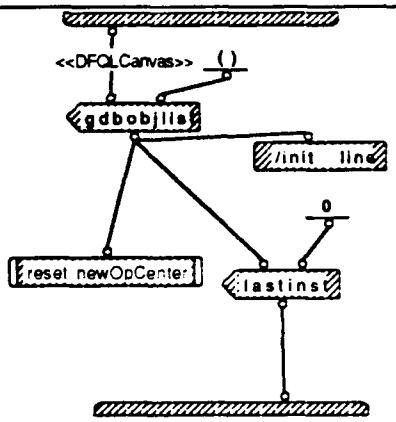
## QueryWindow/reset 1:1



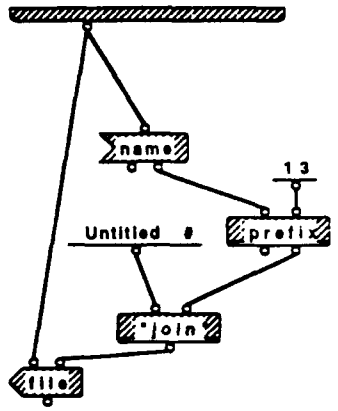

Clear drawing canvas in Query Window, empty gdbobjlist.

§1. (0 0 32000 32000)

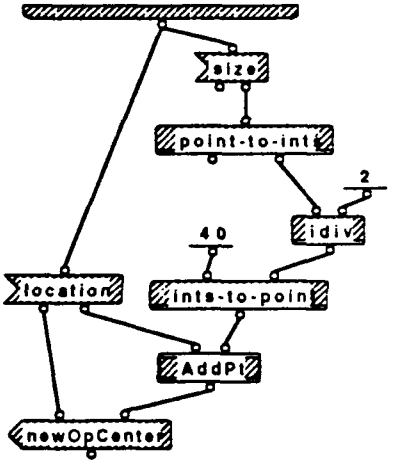
## QueryWindow/reset 1:1reset canvas 1:1



## QueryWindow/reset 1:1reset file 1:1

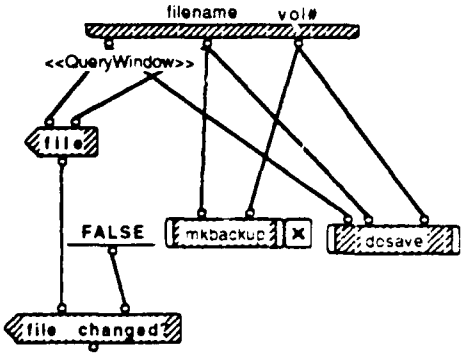



## QueryWindow/reset 1:1reset canvas 1:1reset newOpCenter 1:1




Locate the newOpCenter at the top of canvas.

## QueryWindow/saveit 1:2




First backup the file if possible, and then save the data from areclist into filename on vol#.

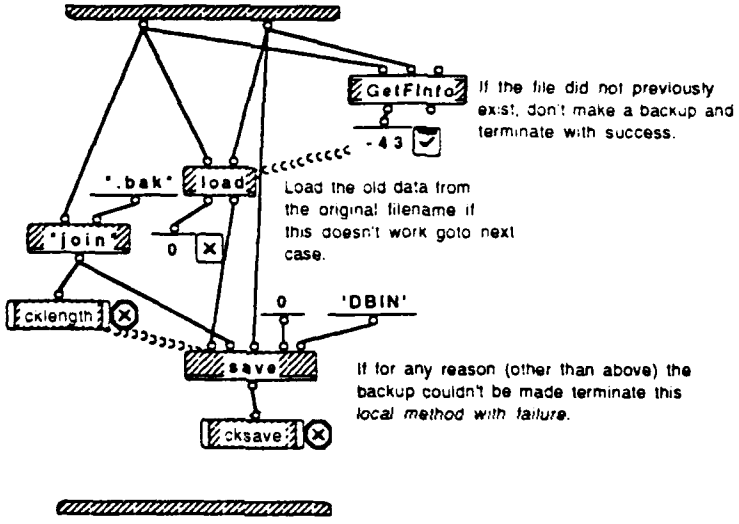
## QueryWindow/saveit 2:2

filename vol#

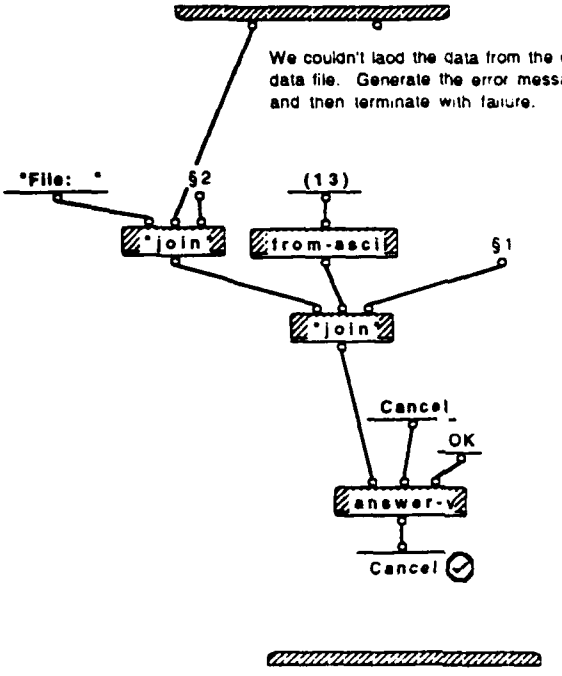
If the backup operation failed, then don't save the new file because it could write over our previous data without having that data backed up.

The error messages for failed backup are *all contained in the mkbackup local method.*

## QueryWindow/saveit 1:2mkbackup 1:2

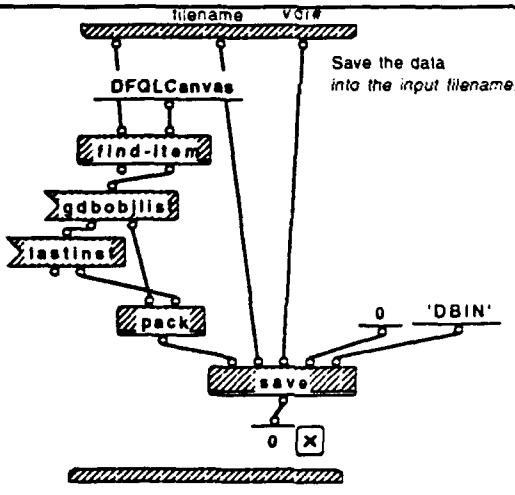


## QueryWindow/saveit 1:2mkbackup 2:2

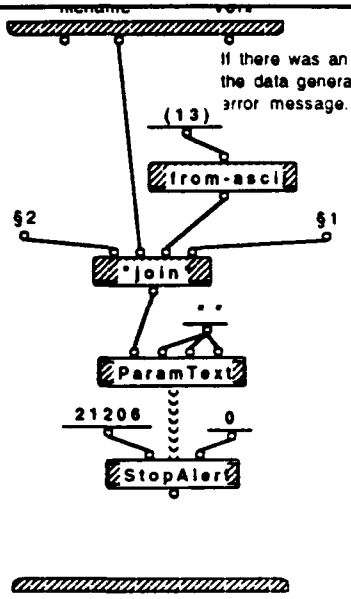



§1. "Continuing will cause this file to be replaced!"
§2. " is not an appropriate dbinterface file."

## QueryWindow/saveit 1:2dosave 1:2

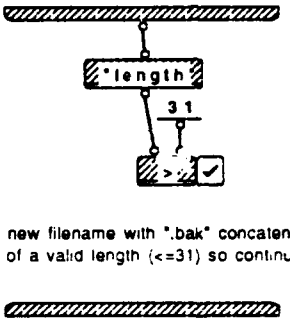


## QueryWindow/saveit 1:2dosave 2:2

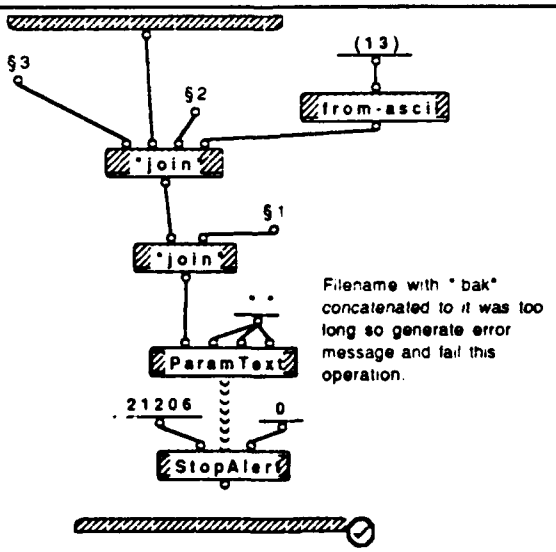



§1. Your data will not be saved!
§2. "Error in attempting to write to "

## QueryWindow/saveit 1:2mkbackup 1:2cklength 1:2



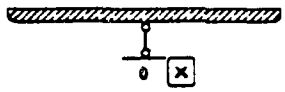

## QueryWindow/saveit 1:2mkbackup 1:2cklength 2:2



## QueryWindow/saveit 1:2mkbackup 1:2cklength 2:2

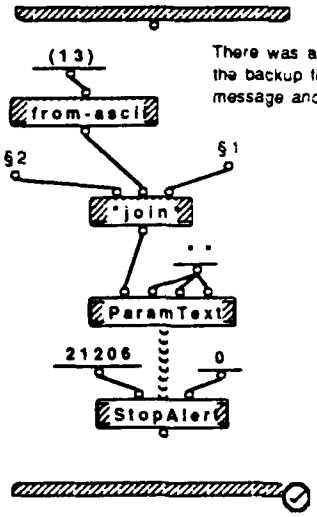
Your file will not be saved!"
is too long!"
Sorry, Backup file name: "

## QueryWindow/saveit 1:2mkbackup 1:2cksave 1:2




Saving the backup file was successful.

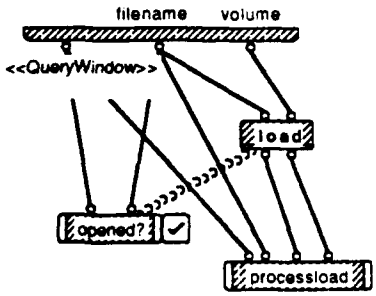
## QueryWindow/saveit 1:2mkbackup 1:2cksave 2:2




There was an error on trying to save the backup file so generate an error message and fail this local method.

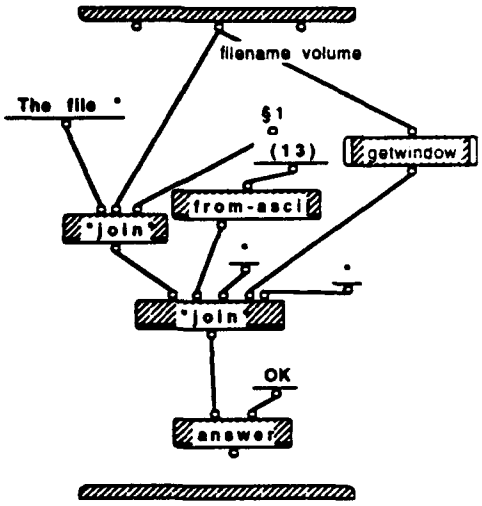
§1. Your data will not be saved!
§2. Backup file could not be made.

## QueryWindow/loadit 1:2



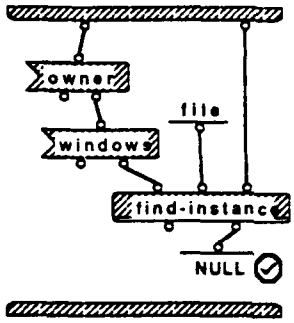

If the file has not been opened load the data from the specified file. Processload updates the gdbobjlist and window or reports on any errors that were generated by the load.

## QueryWindow/loadit 2:2

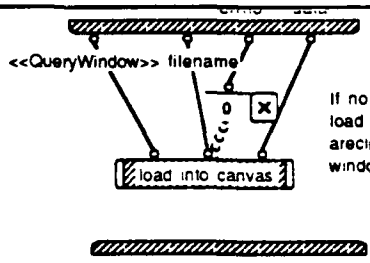



§1. * has been opened in

## QueryWindow/loadit 1:2opened? 1:1

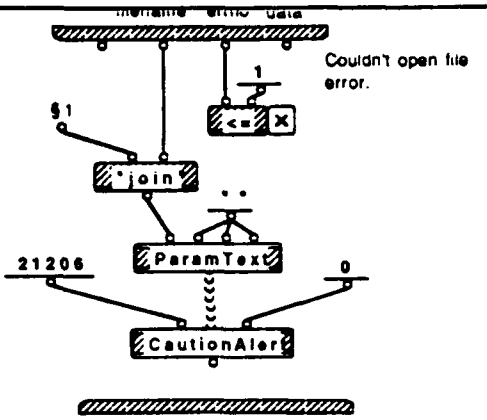



## QueryWindow/loadit 1:2processload 1:3



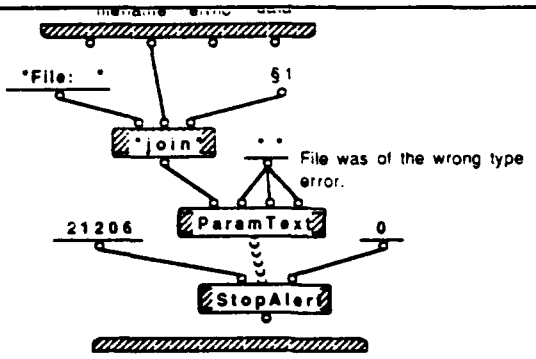

If no error was generated then load the returned data into the arecist and then update the window with the new data.

## QueryWindow/loadit 1:2processload 2:3



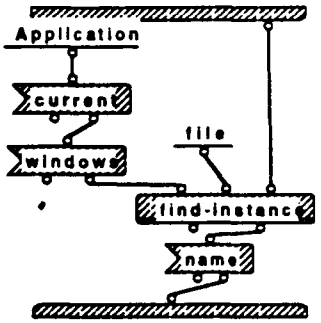

§1. "Sorry, I could not open the file: "

## QueryWindow/loadit 1:2processload 3:3

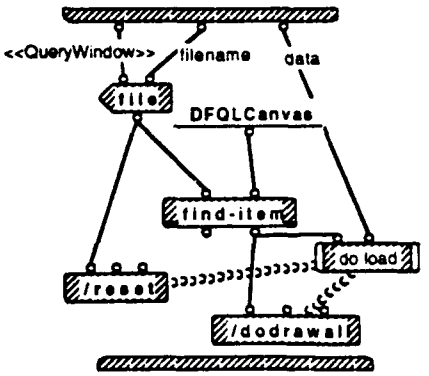



§1. " is of the wrong type!"

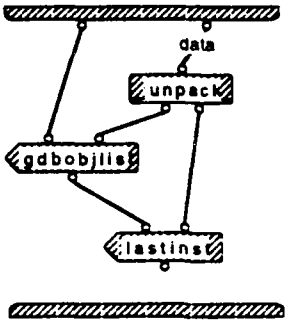
## QueryWindow/loadit 2:2getwindow 1:1



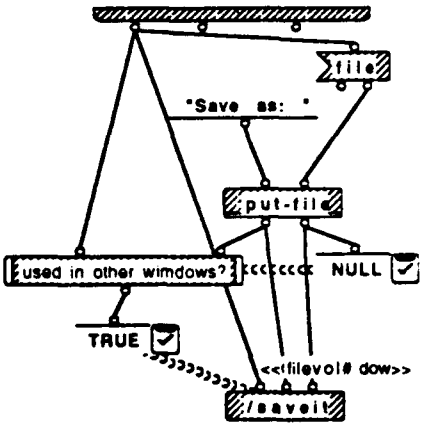
## QueryWindow/loadit 1:2processload 1:3load into canvas 1:1




## QueryWindow/loadit 1:2processload 1:3load into canvas 1:1do load 1:1

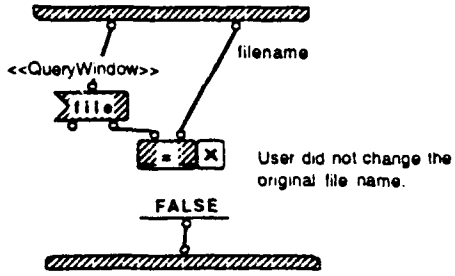



## QueryWindow/save as 1:1

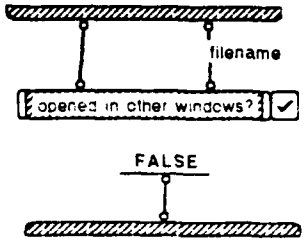



Save the data into
the file entered by the user through
the put-file dialog.

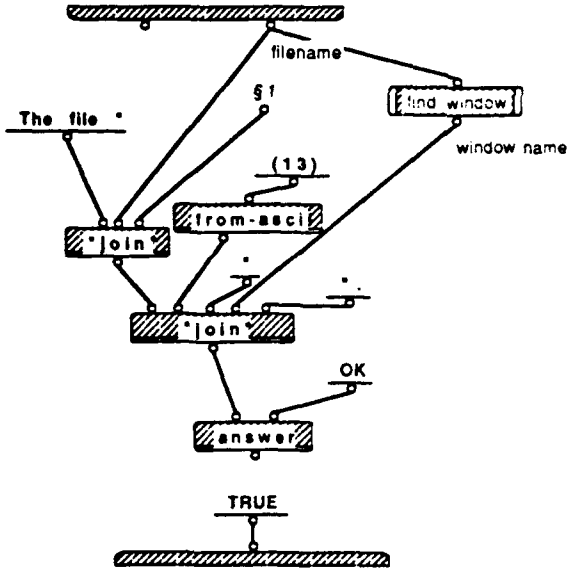
## QueryWindow/save as 1:1used in other wimdows? 1:3




## QueryWindow/save as 1:1used in other wimdows? 2:3

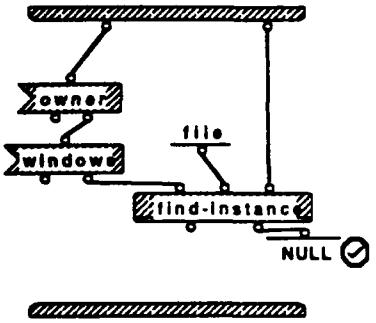



## QueryWindow/save as 1:1used in other wimdows? 3:3

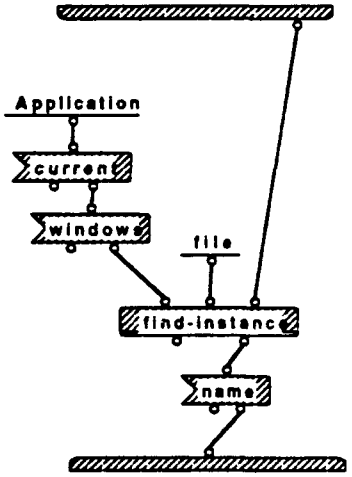



§1. * has been opened in

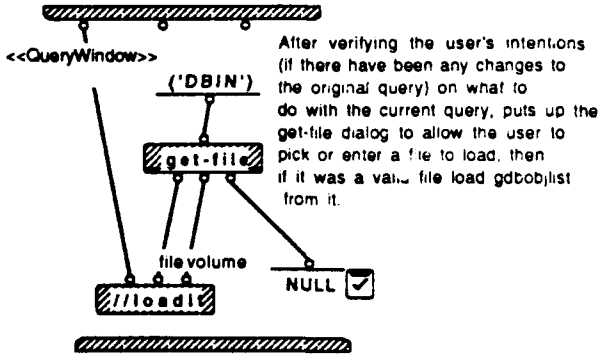
## QueryWindow/save as 1:1used in other wimdows? 2:3opened in other windows? 1:1




## QueryWindow/save as 1:1used in other wimdows? 3:3find window 1:1

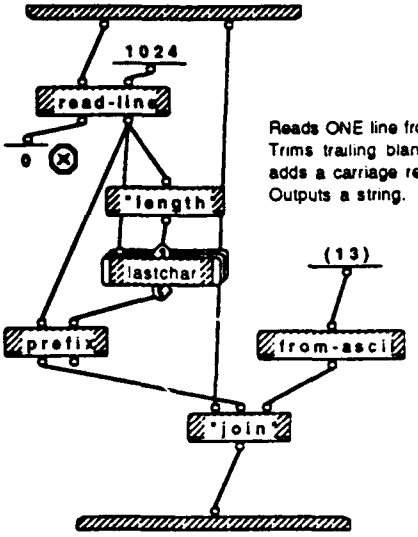



## QueryWindow/open 1:1



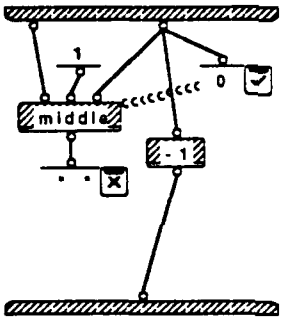

After verifying the user's intentions (if there have been any changes to the original query) on what to do with the current query, puts up the get-file dialog to allow the user to pick or enter a file to load, then if it was a valid file load gdbobjlist from it.

## QueryWindow/readtext 1:1

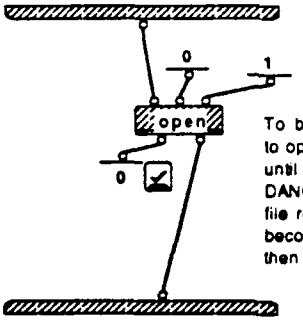



Reads ONE line from a file. Trims trailing blanks and adds a carriage return. Outputs a string.

## QueryWindow/readtext 1:1lastchar 1:1

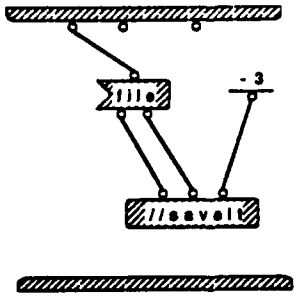



## QueryWindow/openloop 1:1

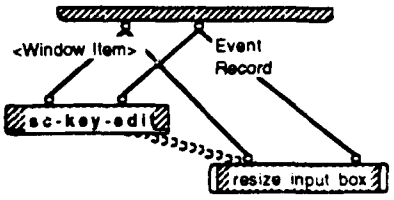



To be called repetitively to open a file. Will loop until the file is opened. DANGEROUS to use if the file requested may not become available. This will then loop forever.

## QueryWindow/save 1:1

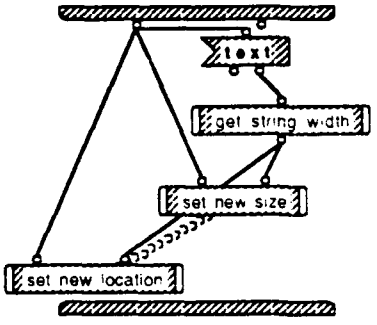


## InputBox/readkeyboard 1:1

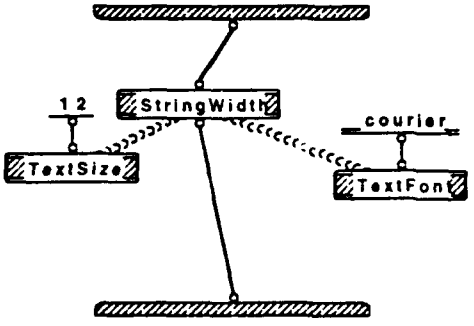



Read keyboard and enlarge the input box according to the length of the text.

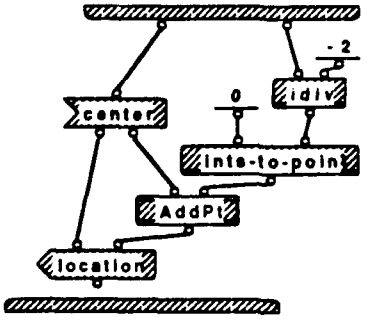
## InputBox/readkeyboard 1:1resize input box 1:1



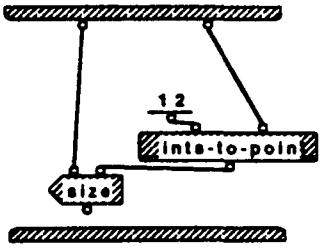

## InputBox/readkeyboard 1:1resize input box 1:1get string width 1:1



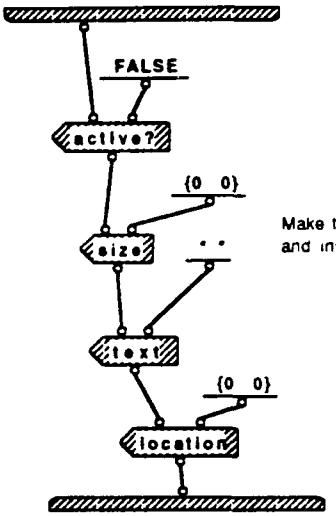

## InputBox/readkeyboard 1:1resize input box 1:1set new location 1:1



## InputBox/readkeyboard 1:1resize input box 1:1set new size 1:1

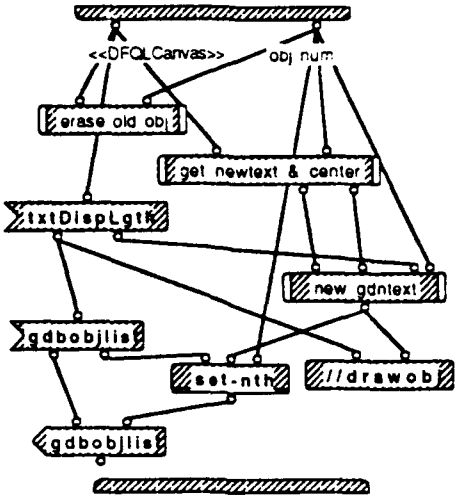



## InputBox/init inputbox 1:1

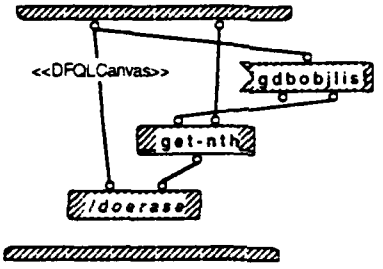



Make the input box unselected and invisible.

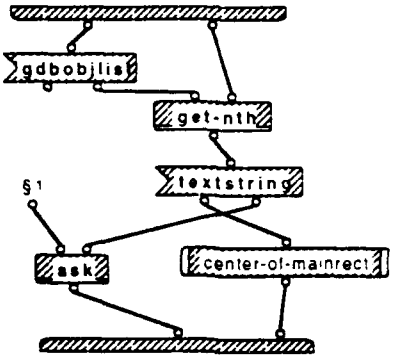
## DFQLCanvas/updategdbtext 1:1




## DFQLCanvas/updategdbtext 1:1erase old obj 1:1

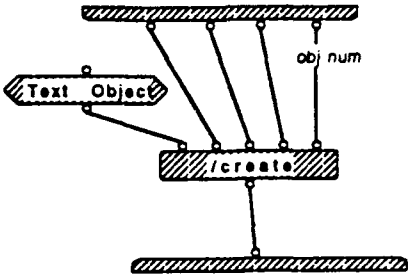



## DFQLCanvas/updategdbtext 1:1get newtext & center 1:1

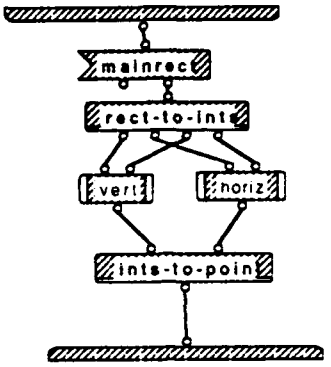



§1. You may edit the string below.

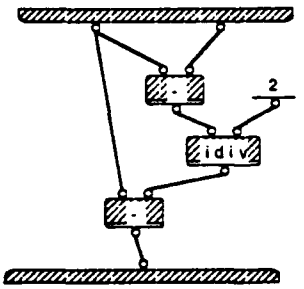
## DFQLCanvas/updategdbtext 1:1new gdntext 1:1



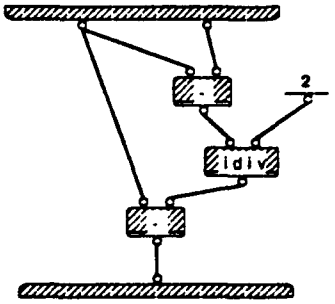

## DFQLCanvas/updategdbtext 1:1get newtext & center 1:1center-of-mainrect 1:1



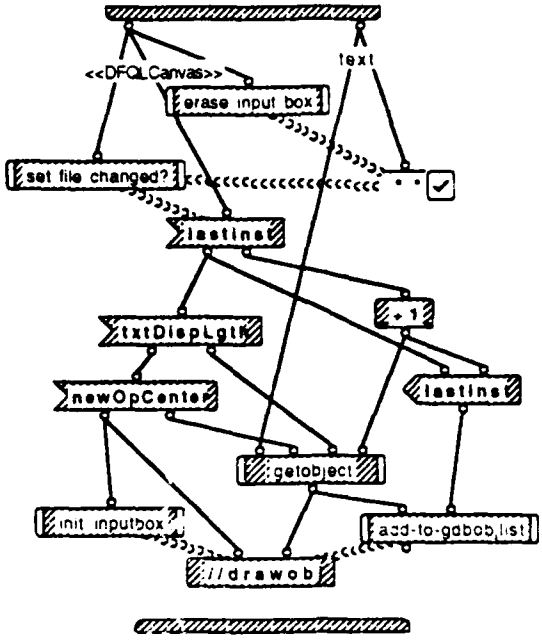

## DFQLCanvas/updategdbtext 1:1get newtext & center 1:1center-of-mainrect 1:1horiz 1:1



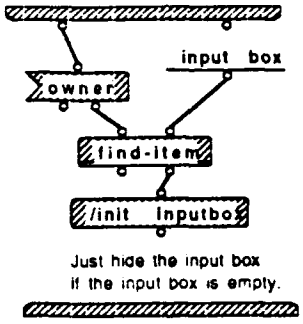

## DFQLCanvas/updategdbtext 1:1get newtext & center 1:1center-of-mainrect 1:1vert 1:1



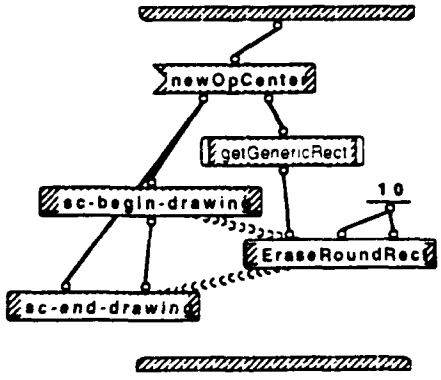
## DFQLCanvas/create 1:2



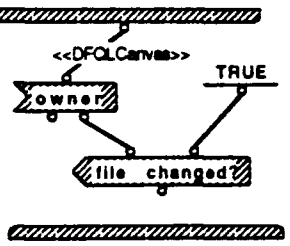

## DFQLCanvas/create 2:2



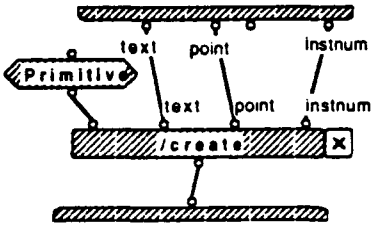

## DFQLCanvas/create 1:2erase input box 1:1



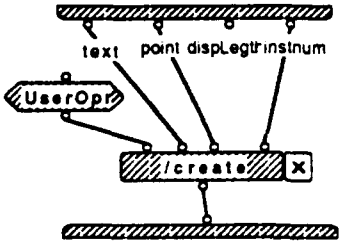
## DFQLCanvas/create 1:2set file changed? 1:1



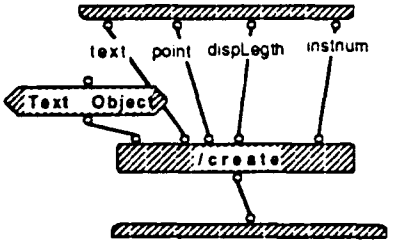

## DFQLCanvas/create 1:2getobject 1:3



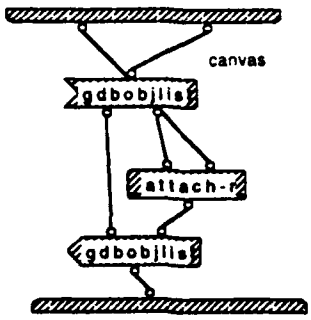

## DFQLCanvas/create 1:2getobject 2:3



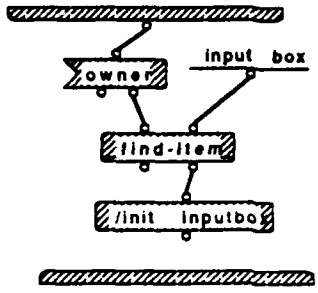

## DFQLCanvas/create 1:2getobject 3:3



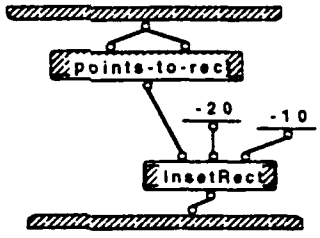
## DFQLCanvas/create 1:2add-to-gdbobjlist 1:1



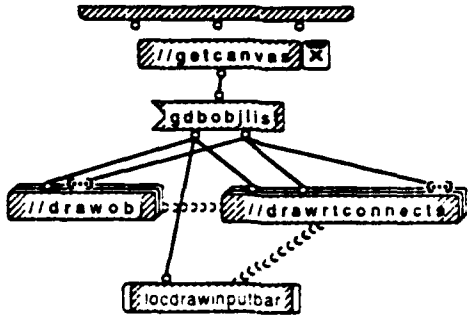

## DFQLCanvas/create 1:2init inputbox 1:1




## DFQLCanvas/create 1:2erase input box 1:1getGenericRect 1:1

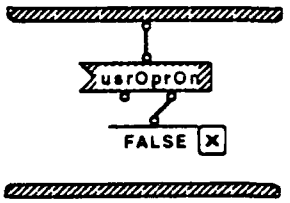



## DFQLCanvas/dodrawall 1:1

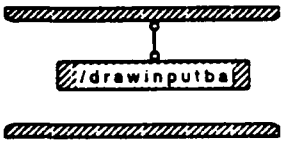



Draws all objects in th canvas.

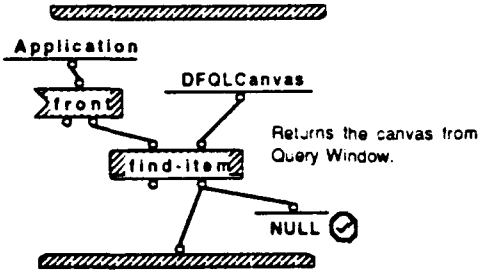
## DFQLCanvas/dodrawall 1:1locdrawinputbar 1:2



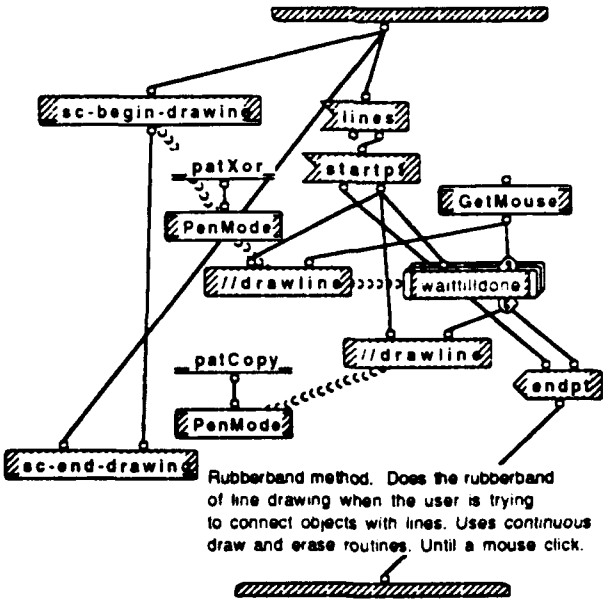

## DFQLCanvas/dodrawall 1:1locdrawinputbar 2:2



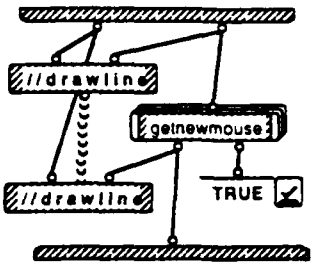

## DFQLCanvas/getcanvas 1:1



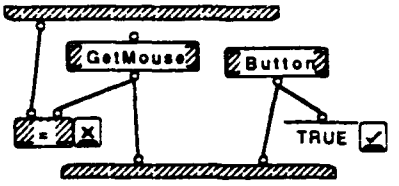

## DFQLCanvas/rubberb 1:1



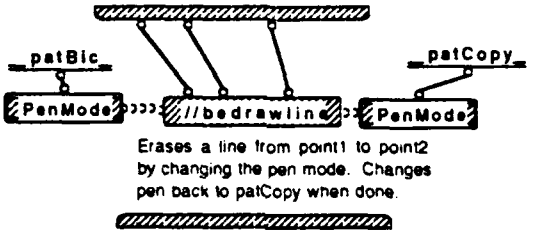
## DFQLCanvas/rubberb 1:1waittilldone 1:1



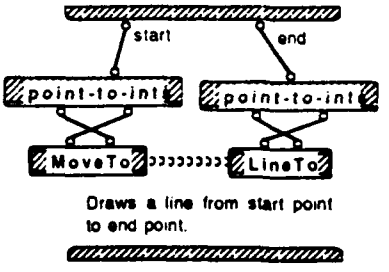

## DFQLCanvas/rubberb 1:1waittilldone 1:1getnewmouse 1:1



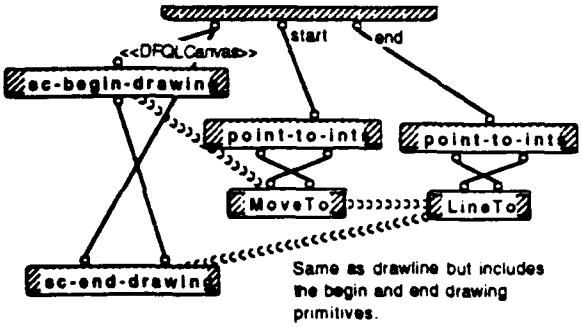

## DFQLCanvas/eraseline 1:1




## DFQLCanvas/drawline 1:1

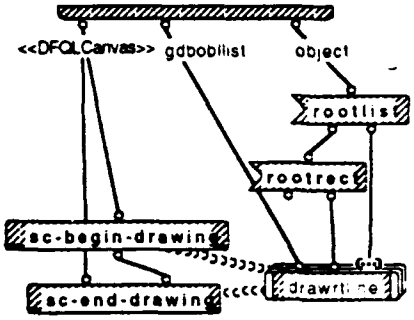


## DFQLCanvas/bedrawline 1:1



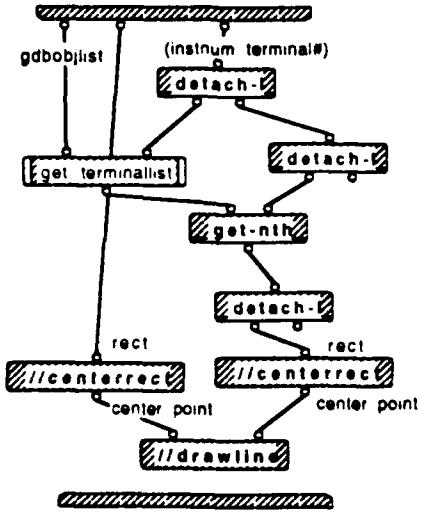

Same as drawline but includes the begin and end drawing primitives.

## DFQLCanvas/drawrtconnects 1:1

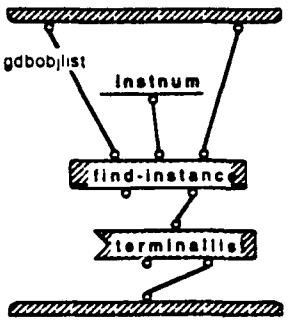



Draws all connecting lines from the root of an input object.

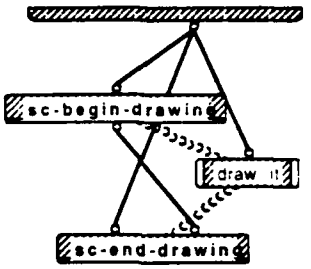
## DFQLCanvas/drawrtconnects 1:1drawrtline 1:1



## DFQLCanvas/drawrtconnects 1:1 drawrtline 1:1 get terminallist 1:1

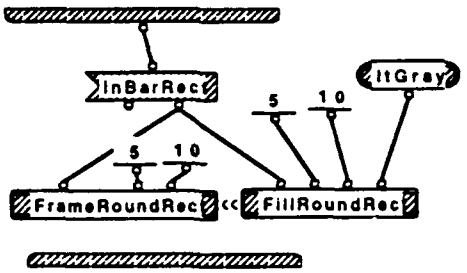



## DFQLCanvas/drawinputbar 1:1

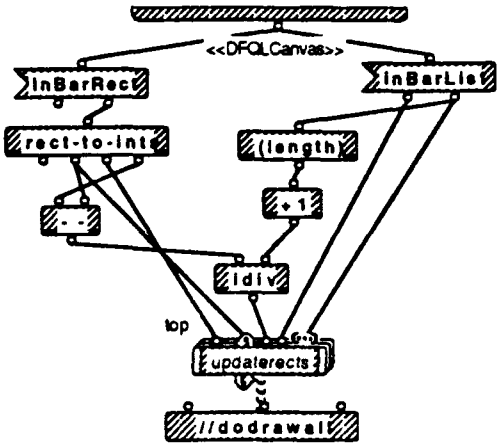



*Draws the input bar for the user-defined operators screens.*

## DFQLCanvas/drawinputbar 1:1 draw it 1:1

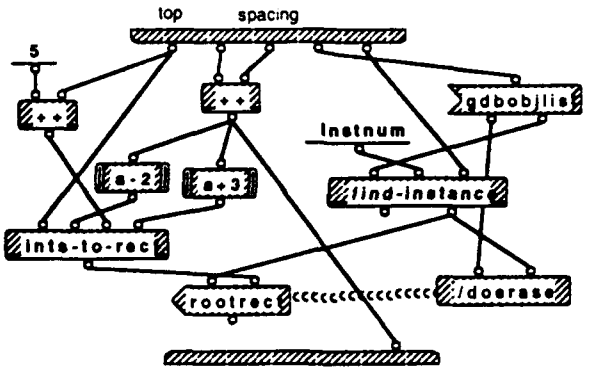


## DFQLCanvas/drawinnodes 1:1

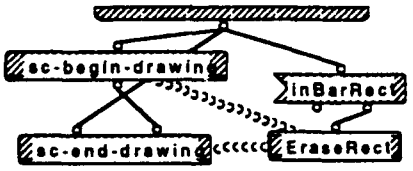



Draws nodes on the input bar.
Used from user-defined operators.

## DFQLCanvas/drawinnodes 1:1updaterects 1:1

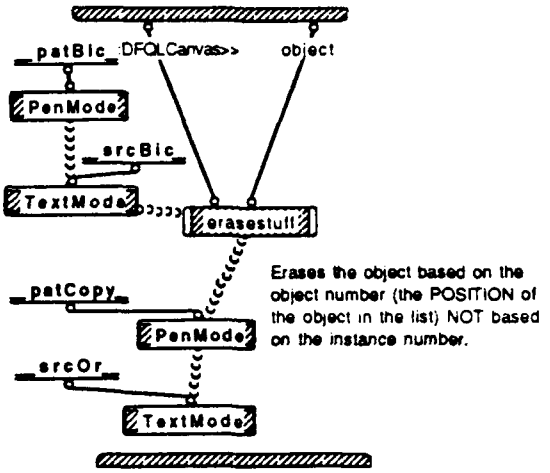



## DFQLCanvas/eraseinputbar 1:1

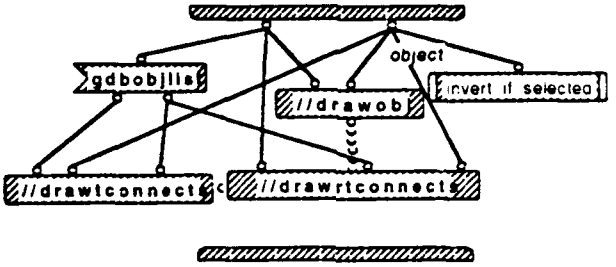



Erases the input bar from the canvas. Used at the termination of user defined ops screens.

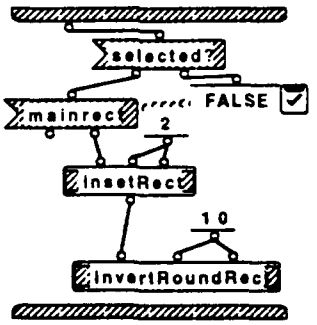
## DFQLCanvas/doerase 1:1



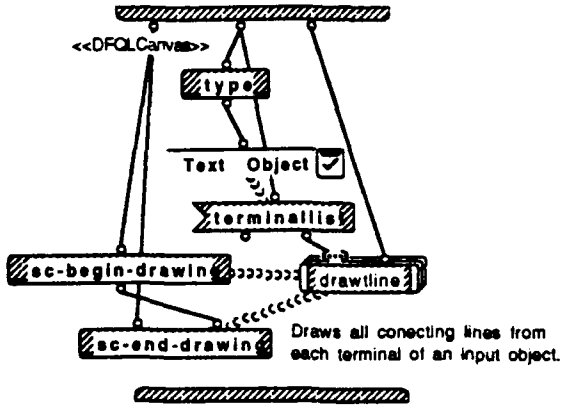

## DFQLCanvas/doerase 1:1erasestuff 1:1



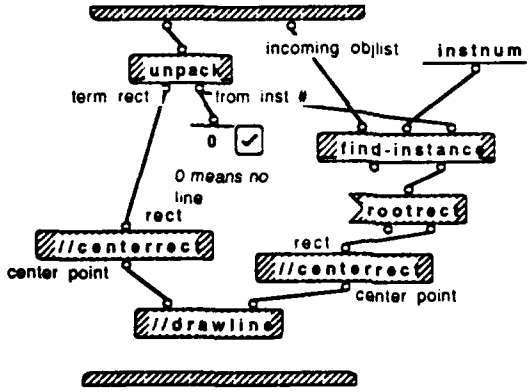

## DFQLCanvas/doerase 1:1erasestuff 1:1invert if selected 1:1



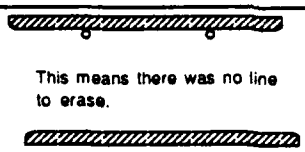
## DFQLCanvas/drawtconnects 1:1



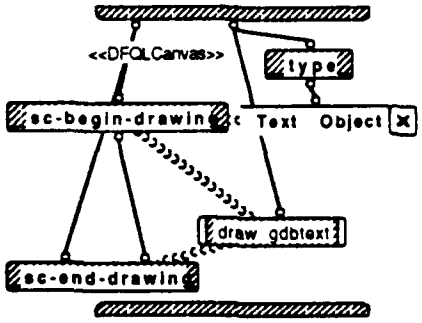

## DFQLCanvas/drawtconnects 1:1drawtline 1:2



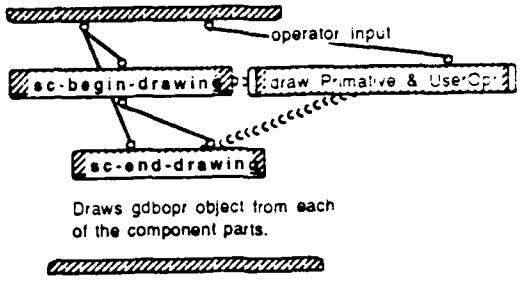

## DFQLCanvas/drawtconnects 1:1drawtline 2:2




## DFQLCanvas/drawobj 1:2

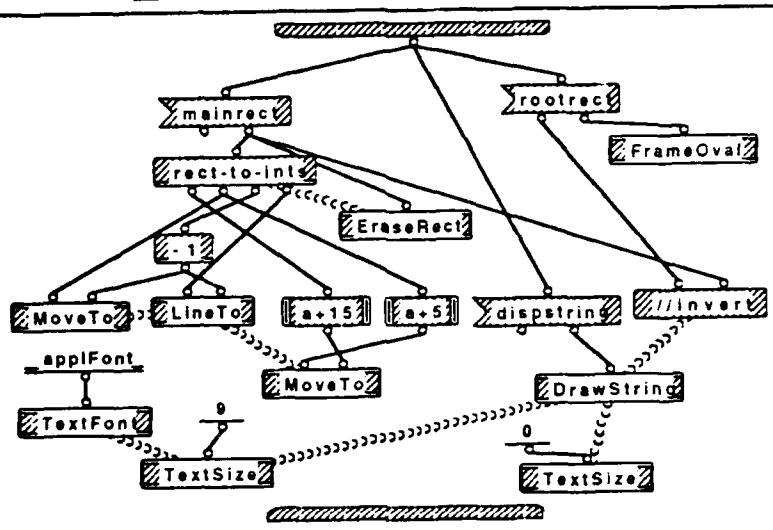


## DFQLCanvas/drawobj 2:2

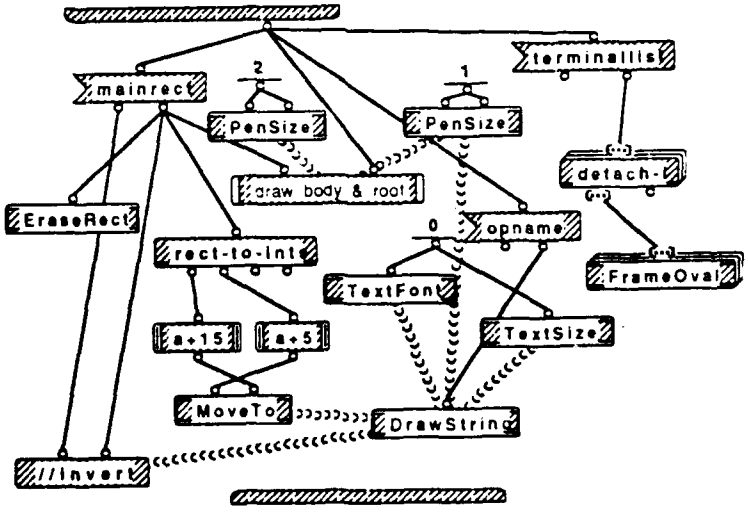



Draws gdbopr object from each of the component parts.

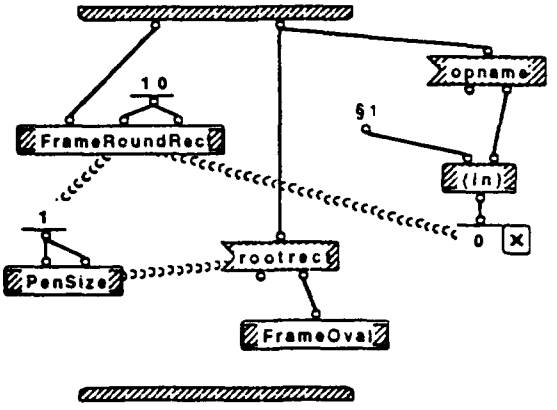
## DFQLCanvas/drawobj 1:2draw gdbtext 1:1




## DFQLCanvas/drawobj 2:2draw Primative & UserOpr 1:1

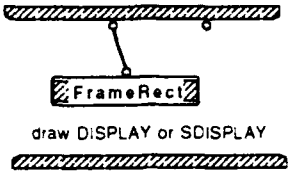


## DFQLCanvas/drawobj 2:2draw Primative & UserOpr 1:1draw body & root 1:2

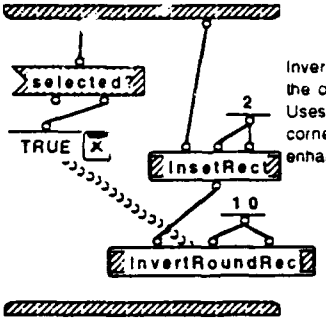



§1. (DISPLAY SDISPLAY)

## DFQLCanvas/drawobj 2:2draw Primative & UserOpr 1:1draw body & root 2:2

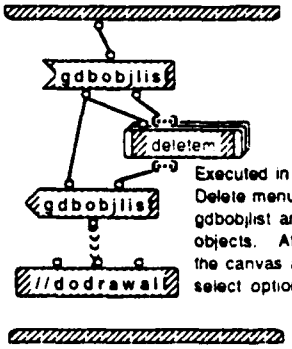



## DFQLCanvas/invert 1:1



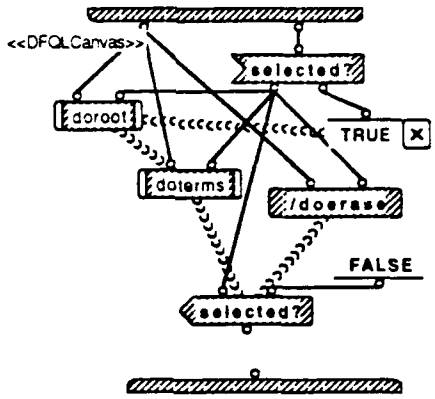

Inverts the color of an object if the object is selected. Uses an inset first so that the corners of the object remain to enhance the appearance.

## DFQLCanvas/delete 1:1

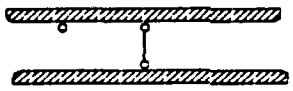



Executed in response to the Delete menu item. Goes through gdbobjlist and deletes all selected objects. After deletions redraws the canvas and then turns off the select option.

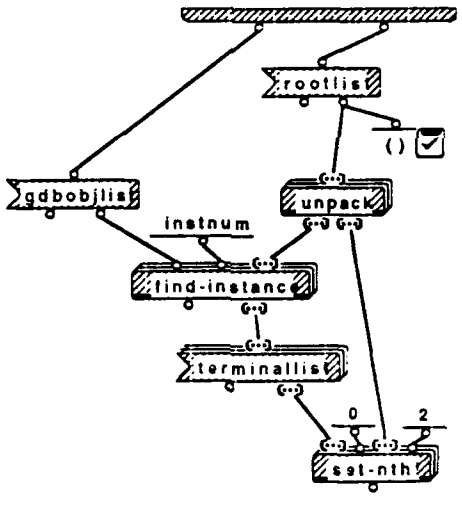
## DFQLCanvas/delete 1:1deletem 1:2



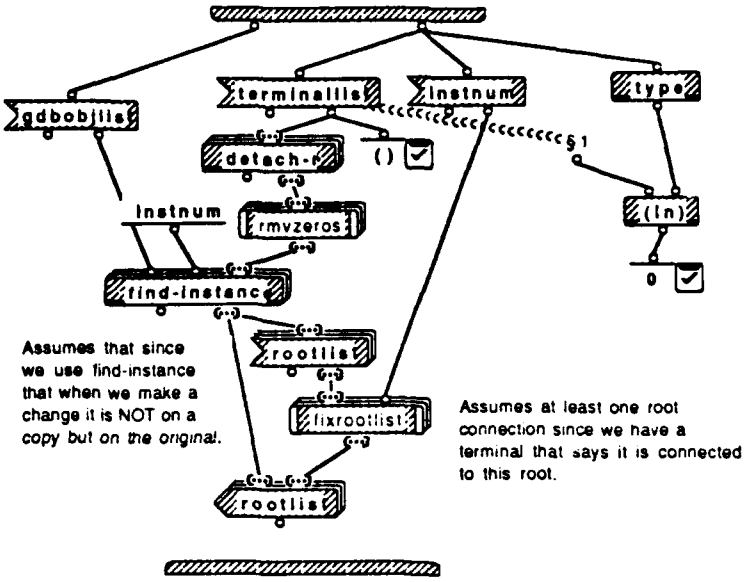

## DFQLCanvas/delete 1:1deletem 2:2



## DFQLCanvas/delete 1:1deletem 1:2doroot 1:1

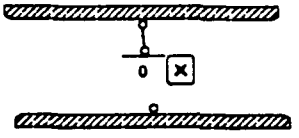


## DFQLCanvas/delete 1:1deletem 1:2doterms 1:1

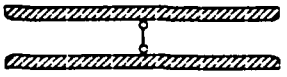



§1. (Primitive UserOpr)

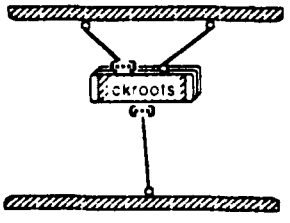
## DFQLCanvas/delete 1:1deletem 1:2doterms 1:1rmvzeros 1:2



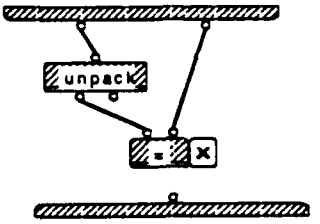

## DFQLCanvas/delete 1:1deletem 1:2doterms 1:1rmvzeros 2:2



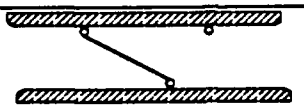
## DFQLCanvas/delete 1:1deletem 1:2doterms 1:1fixrootlist 1:1



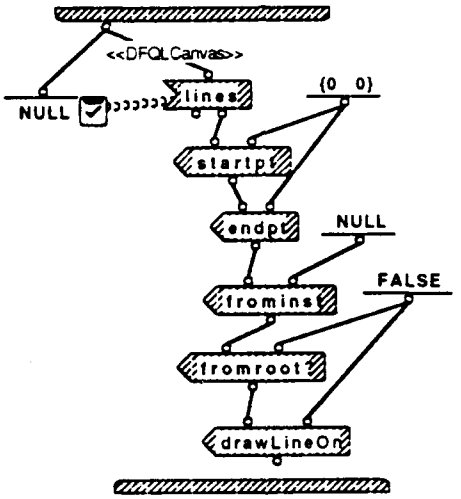
## DFQLCanvas/delete 1:1deletem 1:2doterms 1:1fixrootlist 1:1ckroots 1:2



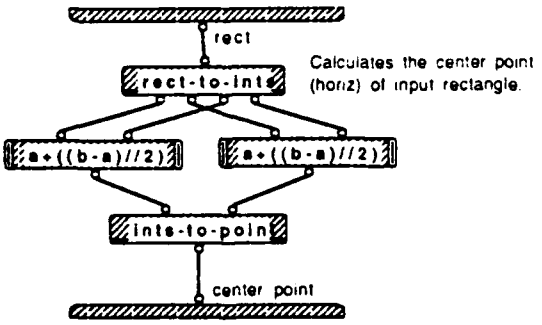

## DFQLCanvas/delete 1:1deletem 1:2doterms 1:1fixrootlist 1:1ckroots 2:2



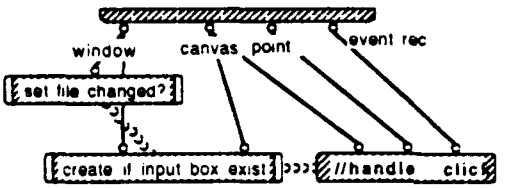
## DFQLCanvas/init line 1:1




## DFQLCanvas/centerrect 1:1

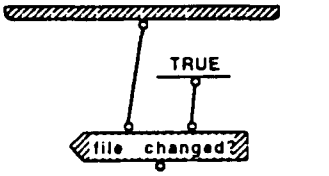


## DFQLCanvas/myclick 1:1

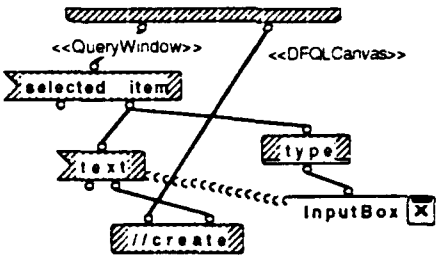



The contents of the window is changed when clicked. Upon clicking, if there is a input box exists create a gdbobj according to the text in the input box before handling the click.

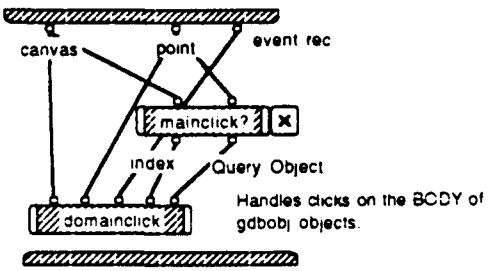
## DFQLCanvas/myclick 1:1set file changed? 1:1




## DFQLCanvas/myclick 1:1create if input box exist 1:1

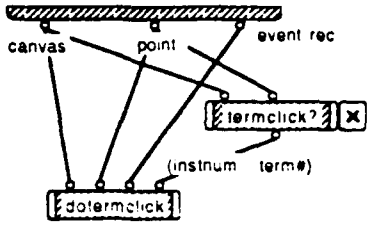



## DFQLCanvas/handle click 1:5



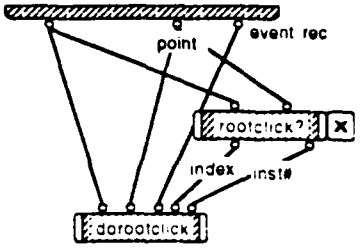

Handles clicks on the BODY of gdbobj objects.

## DFQLCanvas/handle click 2:5



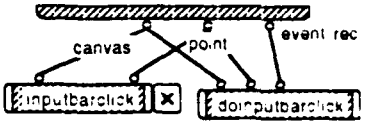

Click on a terminal (input node).

## DFQLCanvas/handle click 3:5



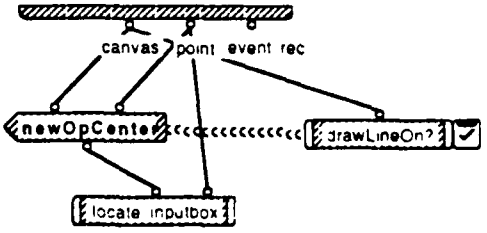

Handles click if on a root (output node)

## DFQLCanvas/handle click 4:5



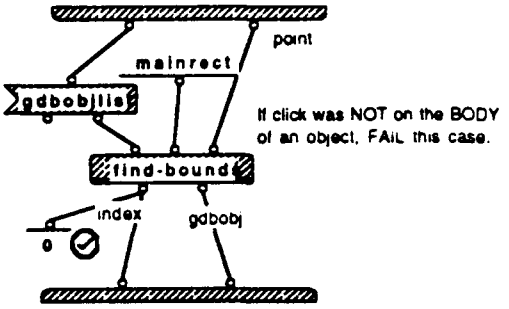

Handle click on the input bar. (From add user operator state.)

## DFQLCanvas/handle click 5:5

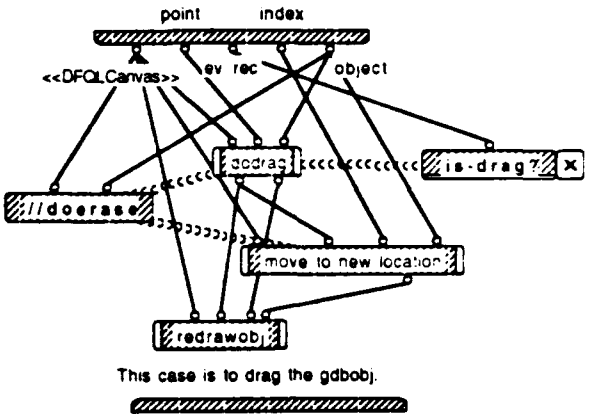



If drawLineOn is true then just terminate this method. Else, if a generic operator exists, create a gdbobj according to the text in the input box. Then create another generic operator

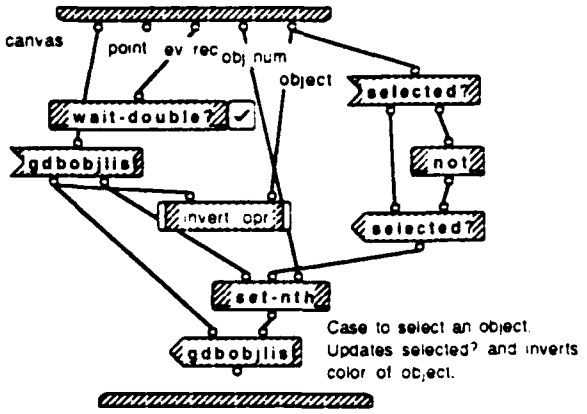
## DFQLCanvas/handle click 1:5mainclick? 1:1



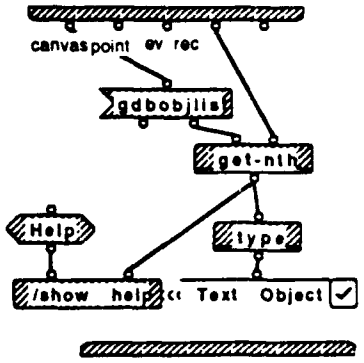

## DFQLCanvas/handle click 1:5domainclick 1:4



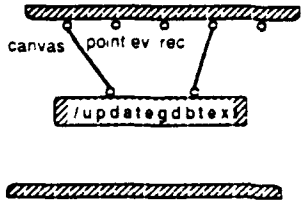

## DFQLCanvas/handle click 1:5domainclick 2:4



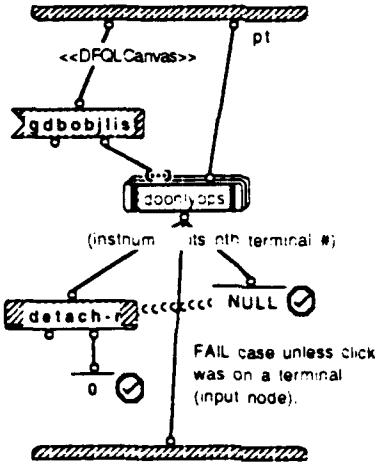
## DFQLCanvas/handle click 1:5domainclick 3:4



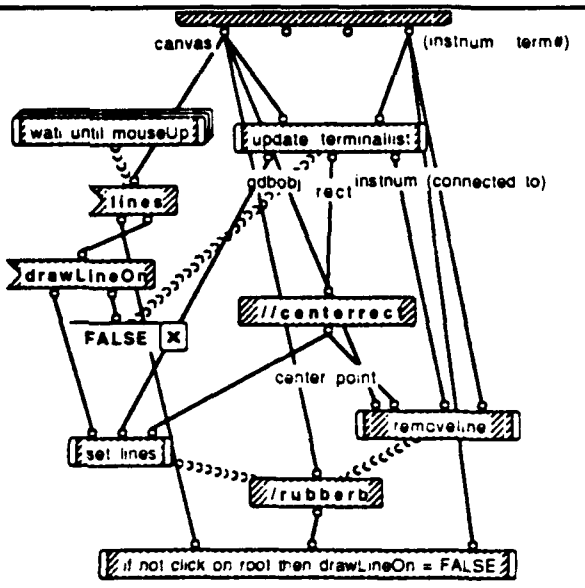

## DFQLCanvas/handle click 1:5domainclick 4:4




## DFQLCanvas/handle click 2:5termclick? 1:1

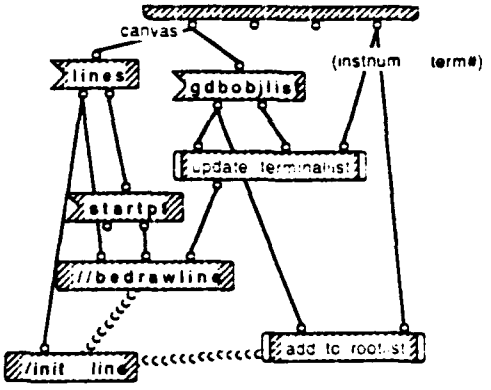


## DFQLCanvas/handle click 2:5dotermclick 1:2



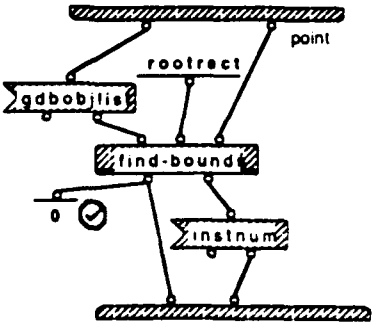

Start with drawLinOn off. Turn it on. Remove any path from terminal. Delete any line from term. Initialize to hook up a new line.

## DFQLCanvas/handle click 2:5dotermclick 2:2

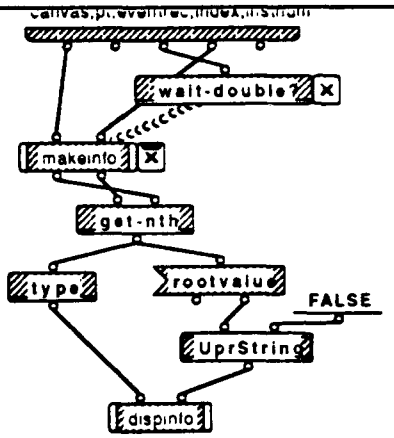



Update terminallist and redrawlines.

## DFQLCanvas/handle click 3:5rootclick? 1:1

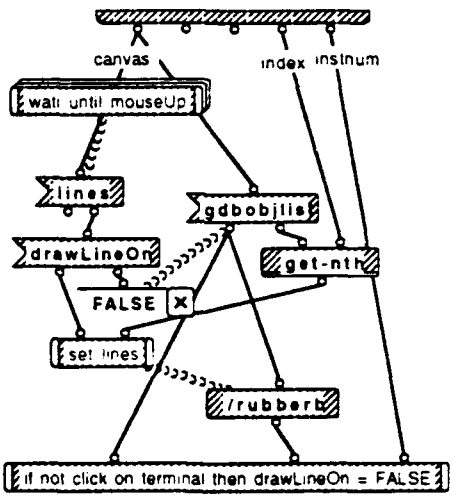


## DFQLCanvas/handle click 3:5dorootclick 1:3

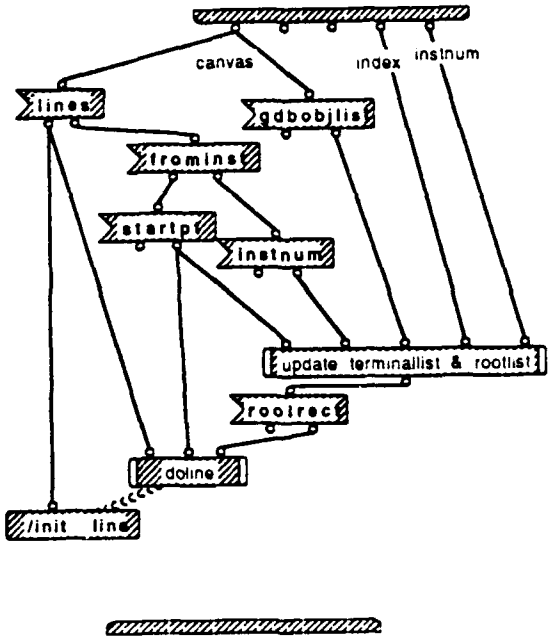



If double clicked in a root display information.

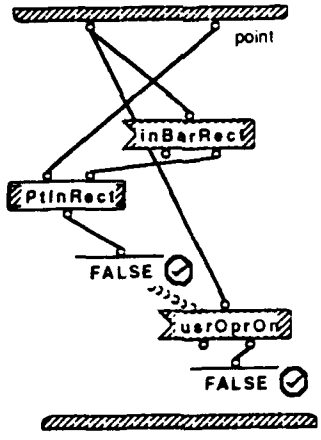
## DFQLCanvas/handle click 3:5dorootclick 2:3



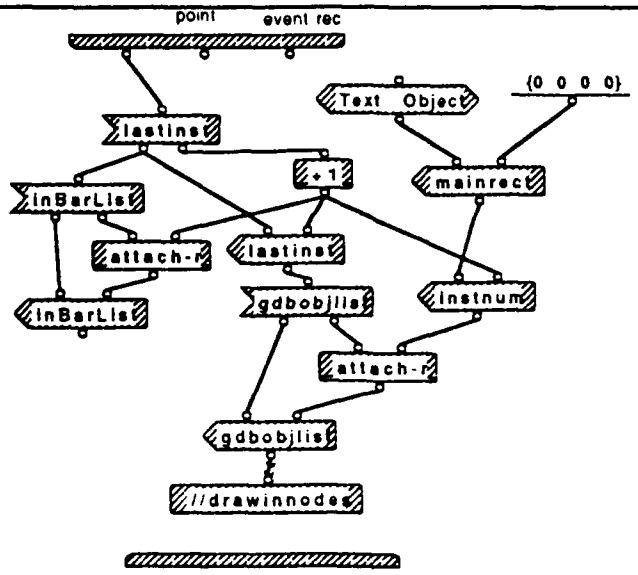
## DFQLCanvas/handle click 3:5dorootclick 3:3



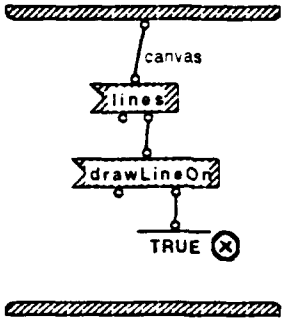

## DFQLCanvas/handle click 4:5inputbarclick 1:1



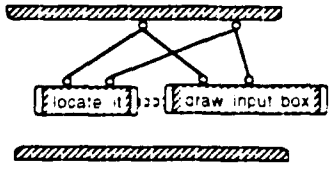
## DFQLCanvas/handle click 4:5doinputbarclick 1:1



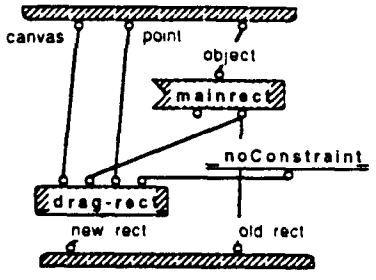

## DFQLCanvas/handle click 5:5drawLineOn? 1:1



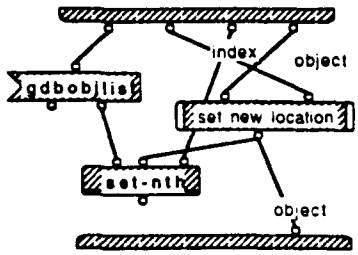

## DFQLCanvas/handle click 5:5locate inputbox 1:1



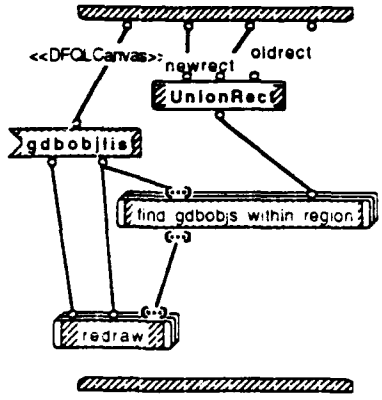

## DFQLCanvas/handle click 1:5domainclick 1:4dodrag 1:1



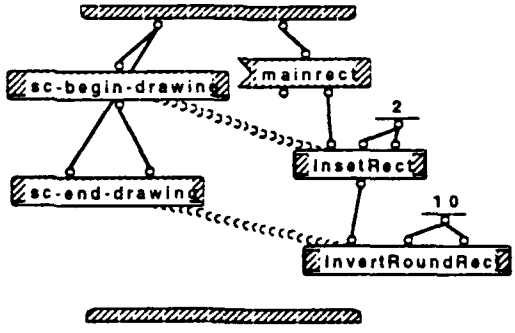
## DFQLCanvas/handle click 1:5domainclick 1:4move to new location 1:1



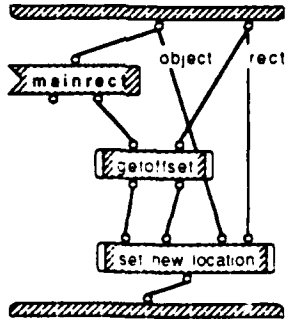

## DFQLCanvas/handle click 1:5domainclick 1:4redrawobj 1:1



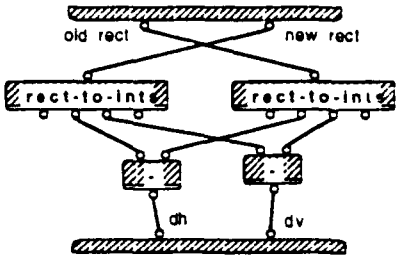

## DFQLCanvas/handle click 1:5domainclick 2:4invert opr 1:1



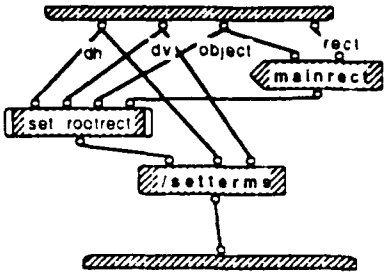

## DFQLCanvas/handle click 1:5domainclick 1:4move to new location 1:1set new location 1:1



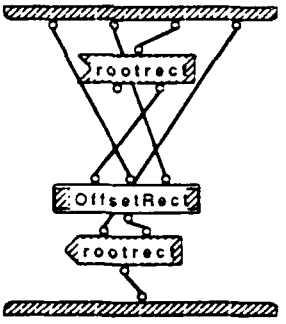
## DFQLCanvas/handle click 1:5domainclick 1:4move to new location 1:1set new location 1:1getoffset 1:1



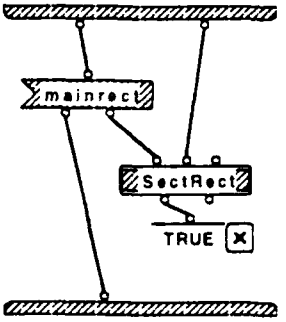

## DFQLCanvas/handle click 1:5domainclick 1:4move to new location 1:1set new location 1:1set new location 1:1



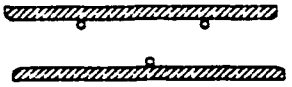

## DFQLCanvas/handle click 1:5domainclick 1:4move to new location 1:1set new location 1:1set new location 1:1set rootrect 1:1



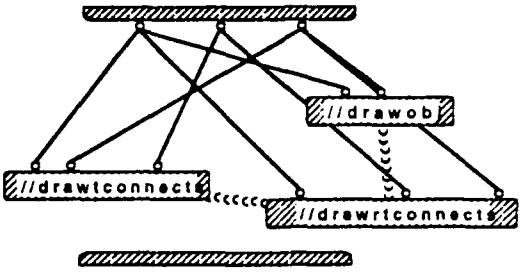

## DFQLCanvas/handle click 1:5domainclick 1:4redrawobj 1:1find gdbobjs within region 1:2



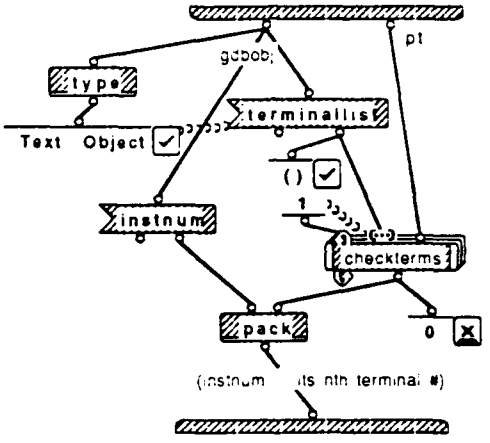
## DFQLCanvas/handle click 1:5domainclick 1:4redrawobj 1:1find gdbobjs within region 2:2



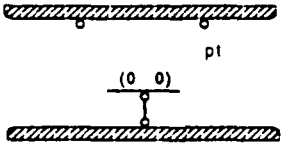
## DFQLCanvas/handle click 1:5domainclick 1:4redrawobj 1:1redraw 1:1



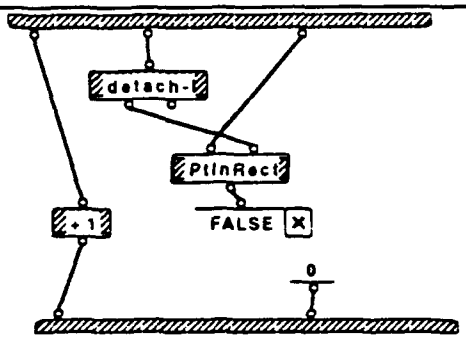

## DFQLCanvas/handle click 2:5termclick? 1:1doonlyops 1:2



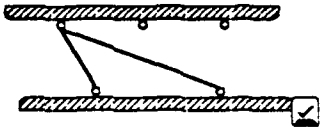

## DFQLCanvas/handle click 2:5termclick? 1:1doonlyops 2:2



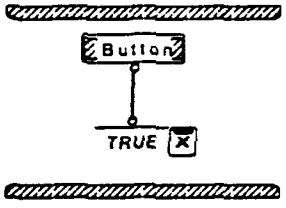
## DFQLCanvas/handle click 2:5termclick? 1:1doonlyops 1:2checkterms 1:2



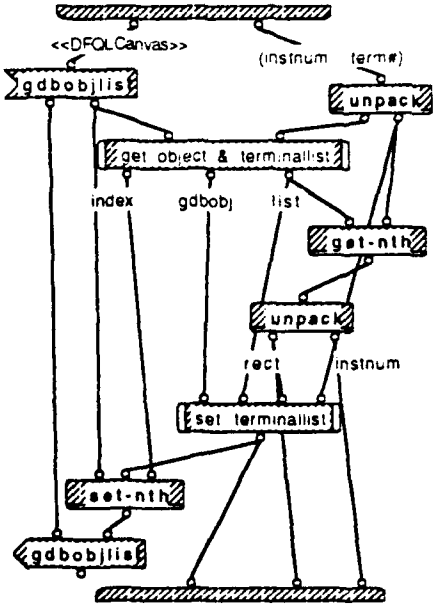

## DFQLCanvas/handle click 2:5termclick? 1:1doonlyops 1:2checkterms 2:2



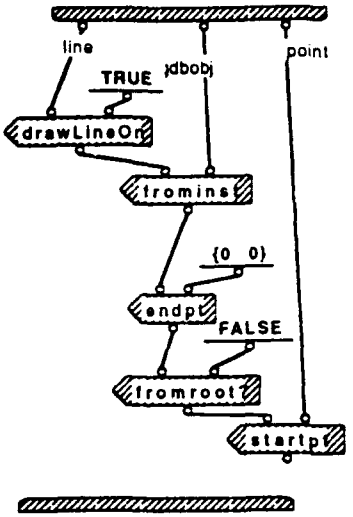
## DFQLCanvas/handle click 2:5dotermclick 1:2wati until mouseUp 1:1



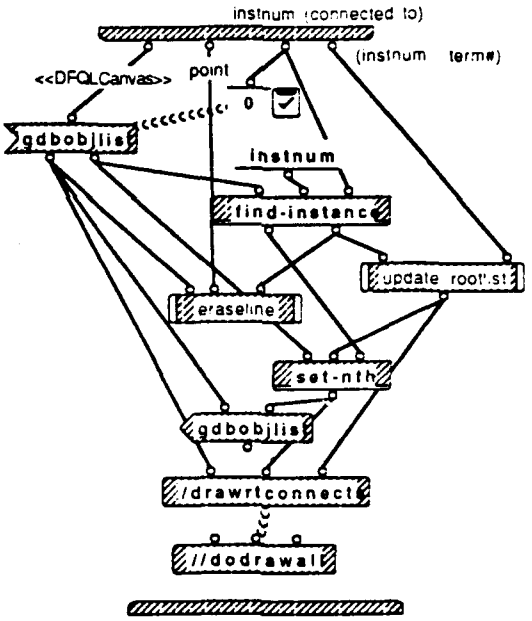

## DFQLCanvas/handle click 2:5dotermclick 1:2update terminallist 1:1



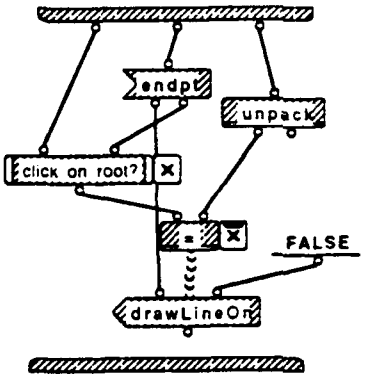
## DFQLCanvas/handle click 2:5dotermclick 1:2set lines 1:1



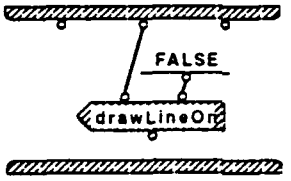

## DFQLCanvas/handle click 2:5dotermclick 1:2removeline 1:1



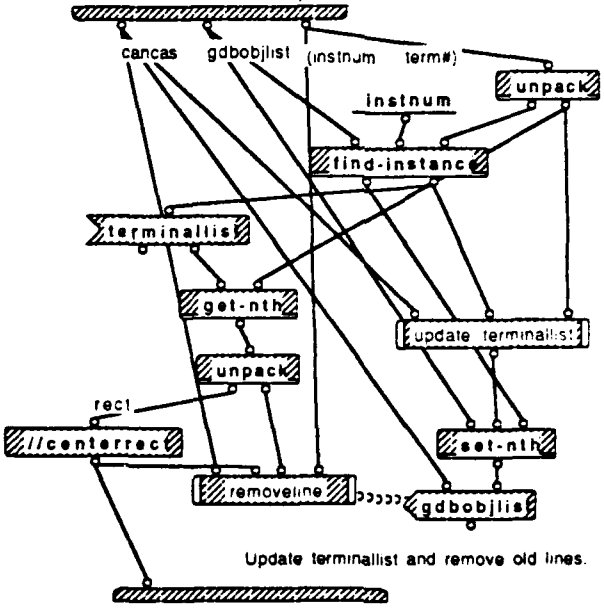
## DFQLCanvas/handle click 2:5dotermclick 1:2if not click on root then drawLineOn = FALSE 1:2



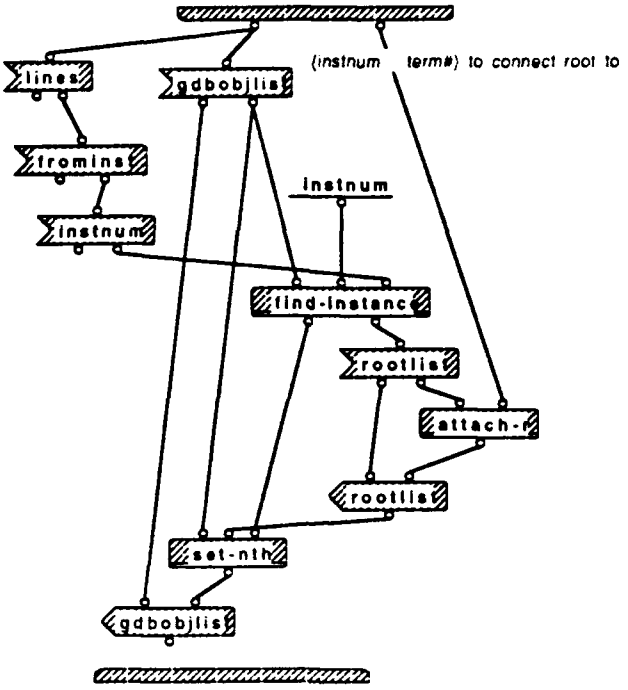

## DFQLCanvas/handle click 2:5dotermclick 1:2if not click on root then drawLineOn = FALSE 2:2



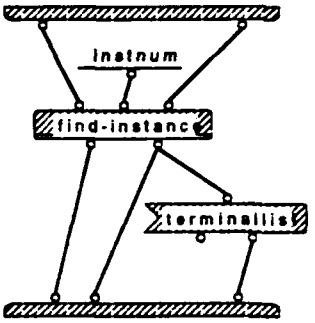

## DFQLCanvas/handle click 2:5dotermclick 2:2update terminallist 1:1



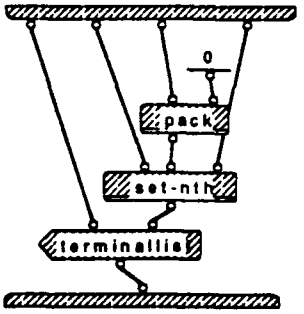
## DFQLCanvas/handle click 2:5dotermclick 2:2add to rootlist 1:1



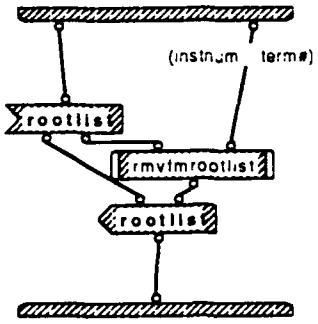

## DFQLCanvas/handle click 2:5dotermclick 1:2update terminallist 1:1get object & terminallist 1:1



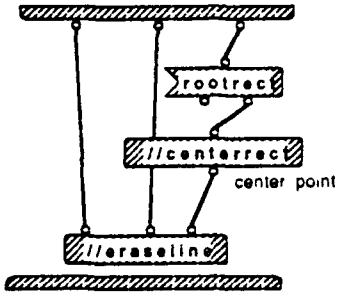

## DFQLCanvas/handle click 2:5dotermclick 1:2update terminallist 1:1set terminallist 1:1



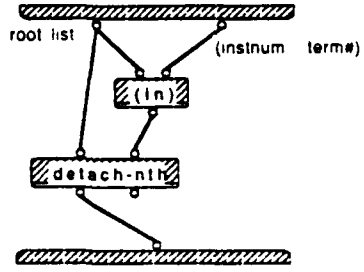
## DFQLCanvas/handle click 2:5dotermclick 1:2removeline 1:1update rootlist 1:1



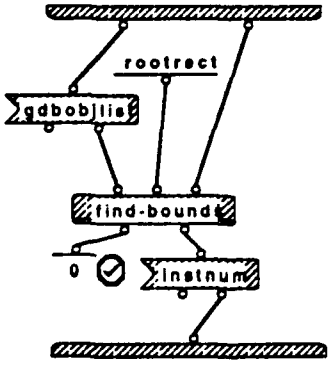

## DFQLCanvas/handle click 2:5dotermclick 1:2removeline 1:1eraseline 1:1



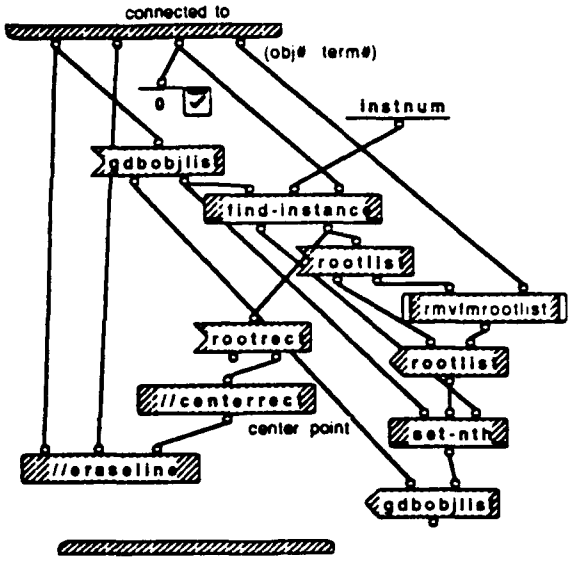

## DFQLCanvas/handle click 2:5dotermclick 1:2removeline 1:1update rootlist 1:1rmvfmrootlist 1:1



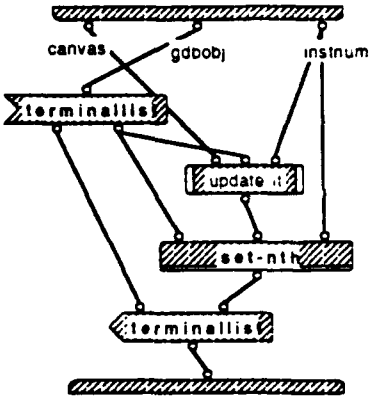

## DFQLCanvas/handle click 2:5dotermclick 1:2if not click on root then drawLineOn = FALSE 1:2click on root? 1:1



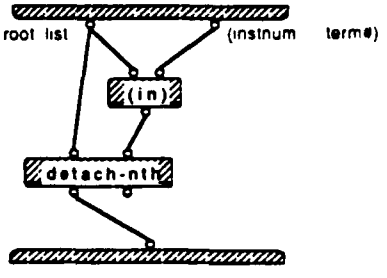
## DFQLCanvas/handle click 2:5dotermclick 2:2update terminallist 1:1removeline 1:1



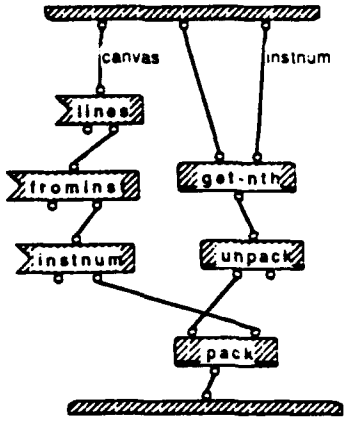

## DFQLCanvas/handle click 2:5dotermclick 2:2update terminallist 1:1update terminallist 1:1



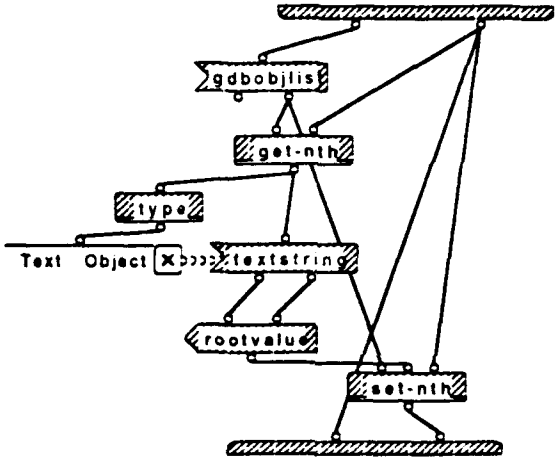

## DFQLCanvas/handle click 2:5dotermclick 2:2update terminallist 1:1removeline 1:1rmvfmrootlist 1:1



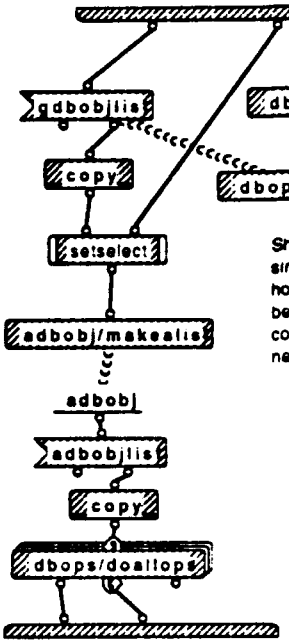
**DFQLCanvas/handle click 2:5dotermclick 2:2update terminallist 1:1update terminallist 1:1update it 1:1**




**DFQLCanvas/handle click 3:5dorootclick 1:3makeinfo 1:3**

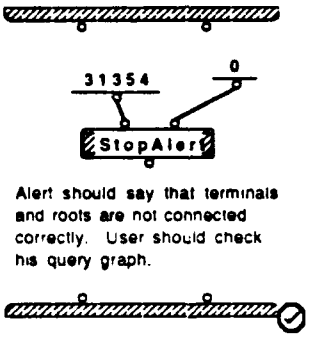


## DFQLCanvas/handle click 3:5dorootclick 1:3makeinfo 2:3



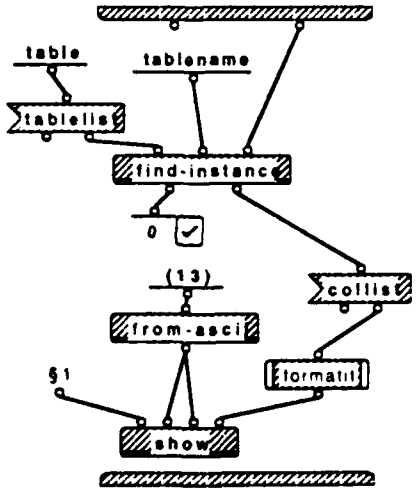

Shouldn't really need a copy here since we remake the list every execution; however this keeps adbobjlist from EVER being corrupted (otherwise it would be corrupt after one execution and before the next).

## DFQLCanvas/handle click 3:5dorootclick 1:3makeinfo 3:3




Alert should say that terminals and roots are not connected correctly. User should check his query graph.

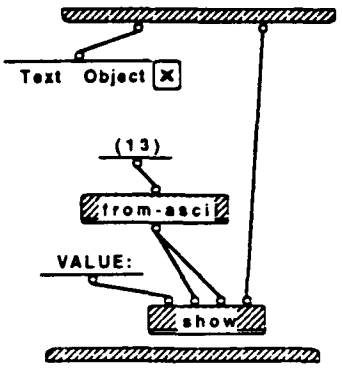
## DFQLCanvas/handle click 3:5dorootclick 1:3dispinfo 1:3



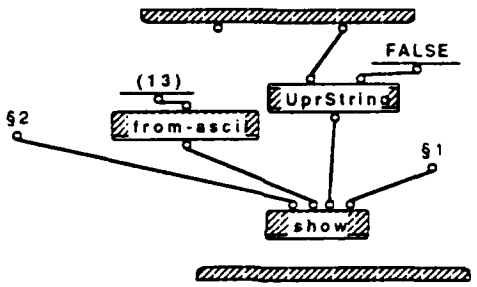
## DFQLCanvas/handle click 3:5dorootclick 1:3dispinfo 1:3

OLUMN NAMES:

## DFQLCanvas/handle click 3:5dorootclick 1:3dispinfo 2:3

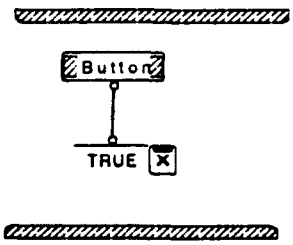



## DFQLCanvas/handle click 3:5dorootclick 1:3dispinfo 3:3

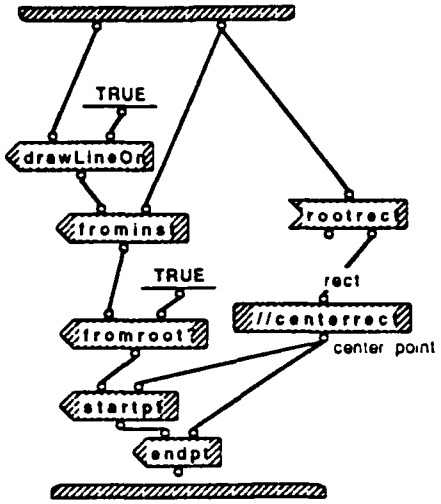



§1. " is not in the table list."
§2. "Error on table lookup! "

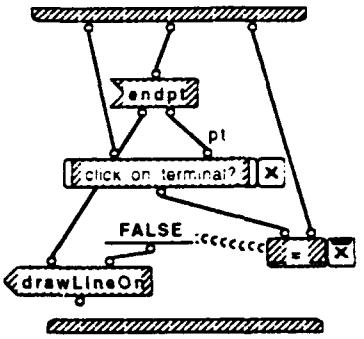
## DFQLCanvas/handle click 3:5dorootclick 2:3wati until mouseUp 1:1



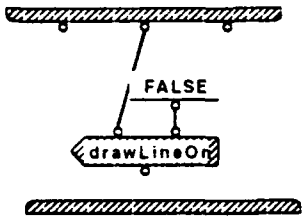
## DFQLCanvas/handle click 3:5dorootclick 2:3set lines 1:1



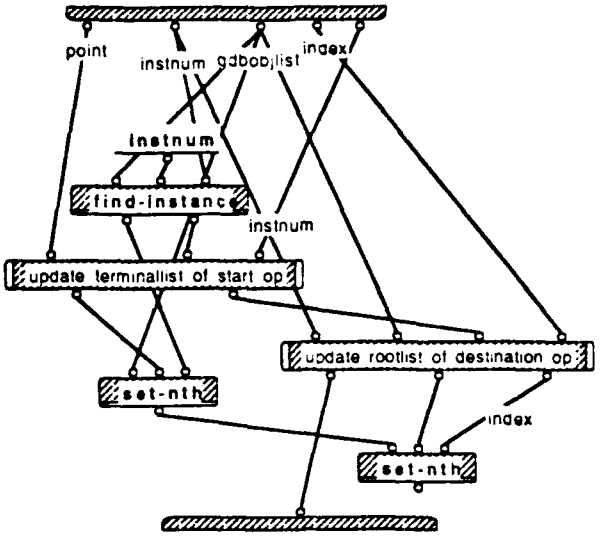

## DFQLCanvas/handle click 3:5dorootclick 2:3if not click on terminal then drawLineOn = FALSE 1:2



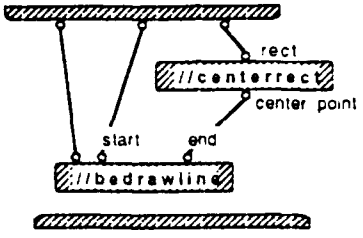

## DFQLCanvas/handle click 3:5dorootclick 2:3if not click on terminal then drawLineOn = FALSE 2:2



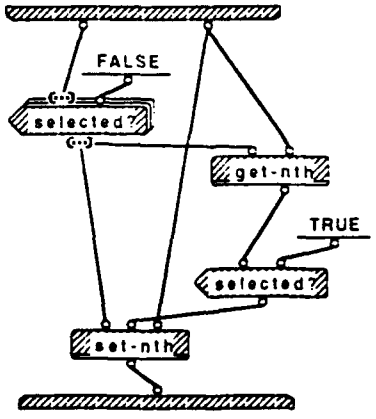
## DFQLCanvas/handle click 3:5dorootclick 3:3update terminallist & rootlist 1:1



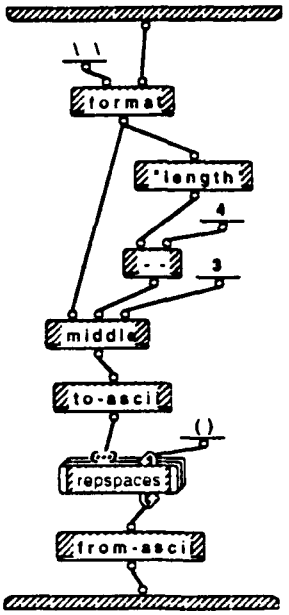

## DFQLCanvas/handle click 3:5dorootclick 3:3doline 1:1



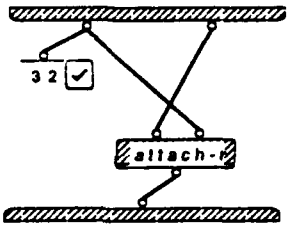

## DFQLCanvas/handle click 3:5dorootclick 1:3makeinfo 2:3setselect 1:1



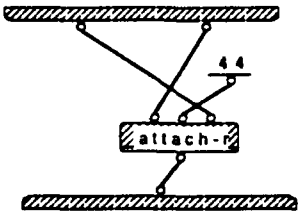
## DFQLCanvas/handle click 3:5dorootclick 1:3dispinfo 1:3formatit 1:1



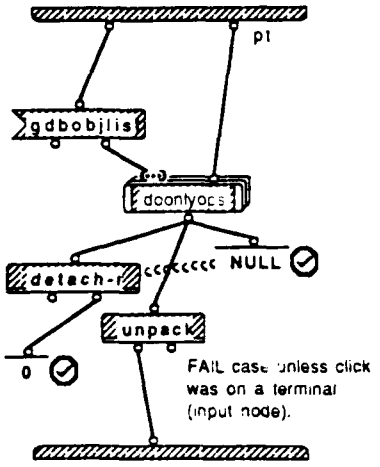

## DFQLCanvas/handle click 3:5dorootclick 1:3dispinfo 1:3formatit 1:1repspaces 1:2



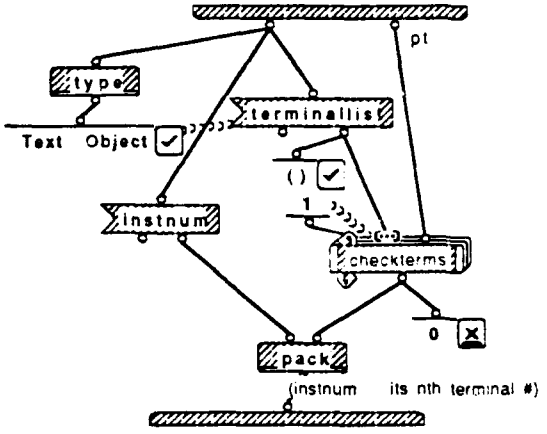

## DFQLCanvas/handle click 3:5dorootclick 1:3dispinfo 1:3formatit 1:1repspaces 2:2



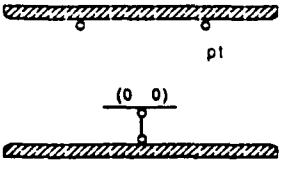
## DFQLCanvas/handle click 3:5dorootclick 2:3if not click on terminal then drawLineOn = FALSE 1:2click on terminal? 1:1



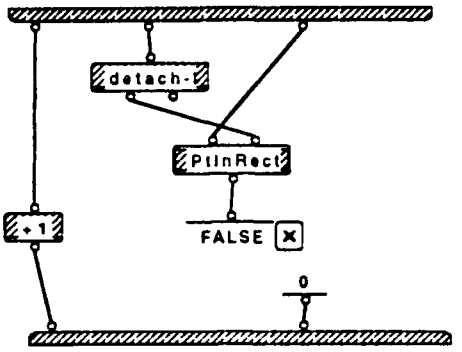

DFQLCanvas/handle click 3:5dorootclick 2:3if not click on terminal then drawLineOn = FALSE 1:2click on terminal? 1:1doonlyops 1:2



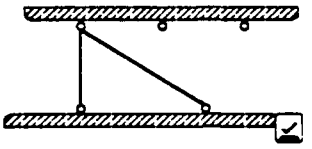

DFQLCanvas/handle click 3:5dorootclick 2:3if not click on termina then drawLineOn = FALSE 1:2click on terminal? 1:1doonlyops 2:2



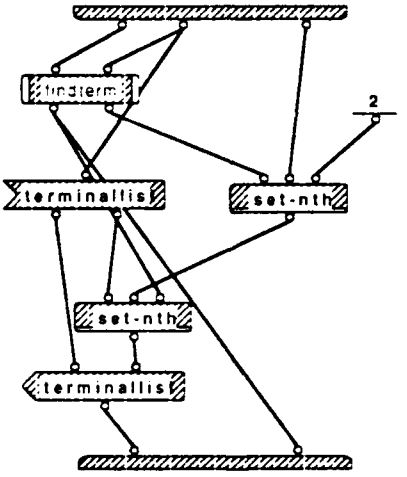
DFQLCanvas/handle click 3:5dorootclick 2:3if not click on terminal then drawLineOn = FALSE 1:2click on terminal? 1:1doonlyops 1:2checkterms 1:2



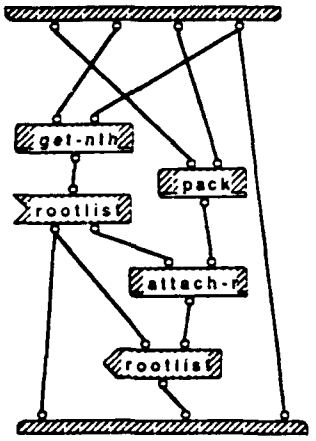

DFQLCanvas/handle click 3:5dorootclick 2:3if not click on terminal then drawLineOn = FALSE 1:2click on terminal? 1:1doonlyops 1:2checkterms 2:2



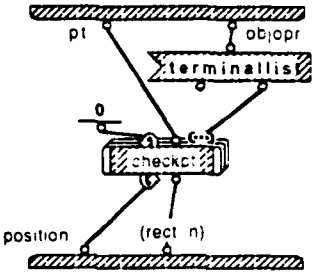
**DFQLCanvas/handle click 3:5dorootclick 3:3update terminallist & rootlist 1:1update terminallist of start op 1:1**



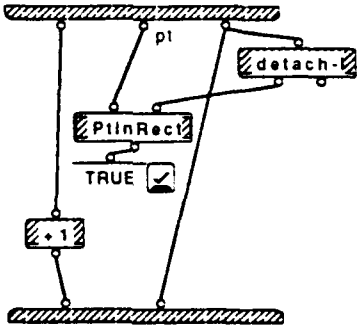
## DFQLCanvas/handle click 3:5dorootclick 3:3update terminallist & rootlist 1:1update rootlist of destination op 1:1



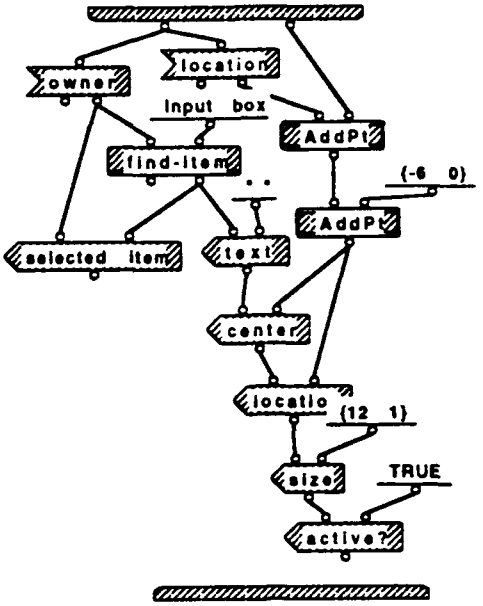

## DFQLCanvas/handle click 3:5dorootclick 3:3update terminallist & rootlist 1:1update terminallist of start op 1:1findterm 1:1



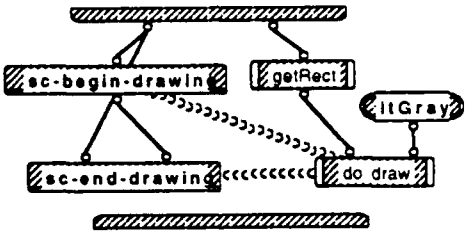

## DFQLCanvas/handle click 3:5dorootclick 3:3update terminallist & rootlist 1:1update terminallist of start op 1:1findterm 1:1checkpt 1:1



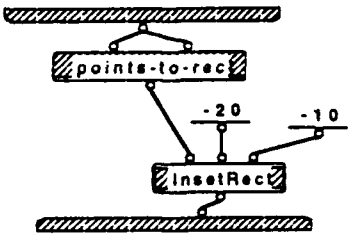
## DFQLCanvas/handle click 5:5locate inputbox 1:1locate it 1:1



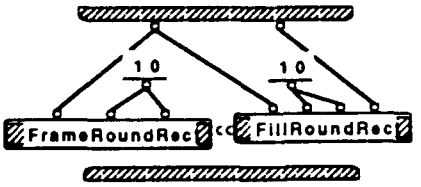

## DFQLCanvas/handle click 5:5locate inputbox 1:1draw input box 1:1



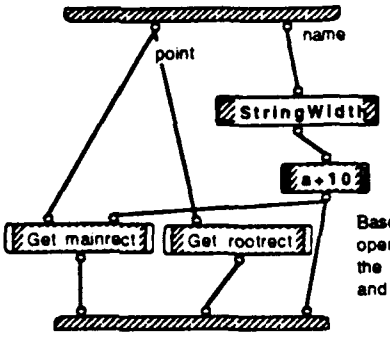

## DFQLCanvas/handle click 5:5locate inputbox 1:1draw input box 1:1getRect 1:1




## DFQLCanvas/handle click 5:5locate inputbox 1:1draw input box 1:1do draw 1:1

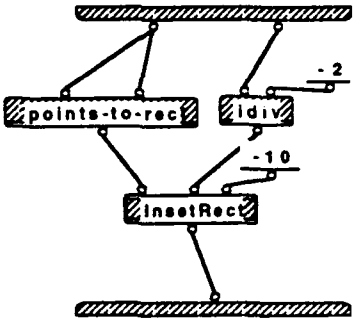


## Query Object/calcrects 1:1

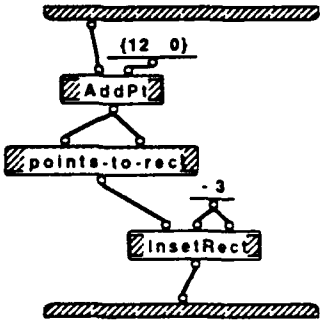



Based on the length of the operator name determines the coords of the main rect and root rect (output node).

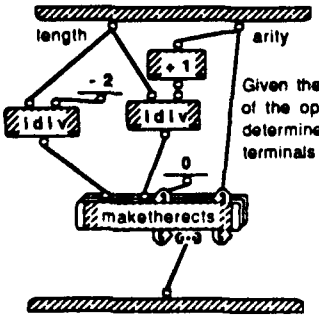
## Query Object/calcrects 1:1Get mainrect 1:1



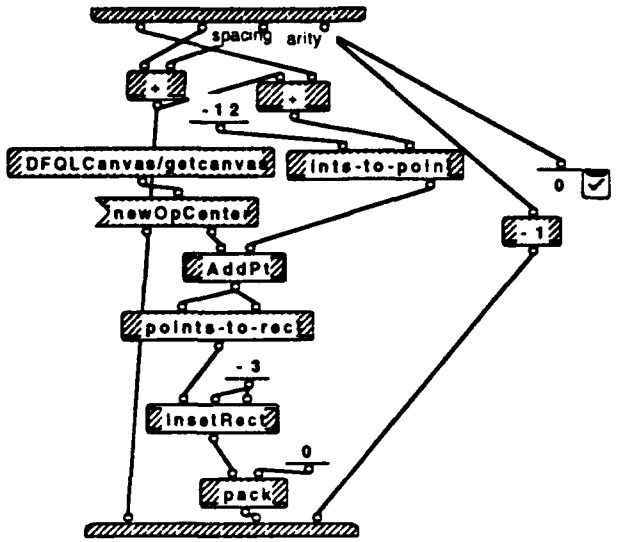

## Query Object/calcrects 1:1Get rootrect 1:1



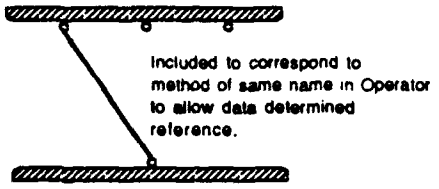

## Query Object/mktrmlst 1:1



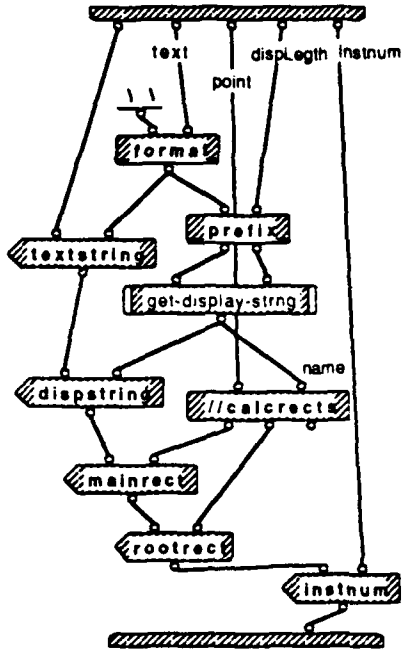

Given the arity and the length of the operator's body rectangle determines where to draw the terminals (input nodes).


spacing
arity
+
+
-12
DFQLCanvas/getcanvas
ints-to-point
0
newOpCenter
-1
AddPt
points-to-rec
-3
insetRect
0
pack

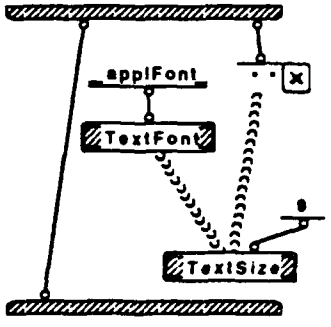
## Text Object/setterms 1:1



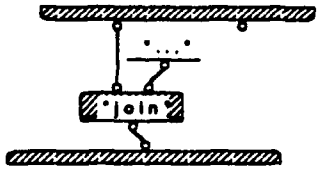

## Text Object/create 1:1



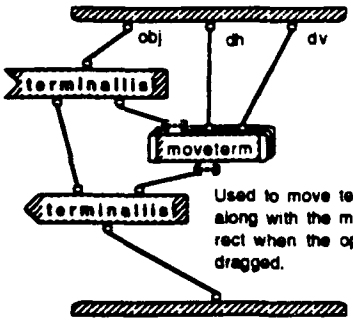

## Text Object/create 1:1get-display-strng 1:2




## Text Object/create 1:1get-display-strng 2:2

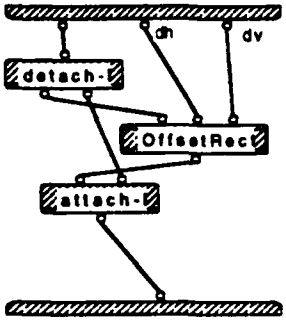


## Operator/setterms 1:1

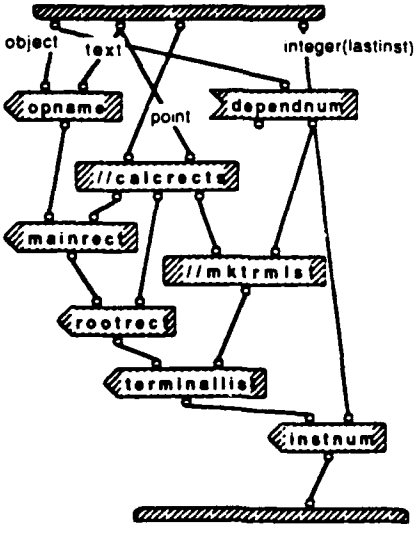



Used to move terminals along with the main (body) rect when the operator is dragged.

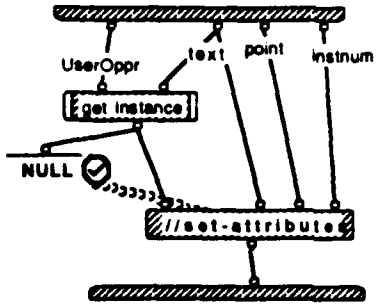
## Operator/setterms 1:1moveterm 1:1



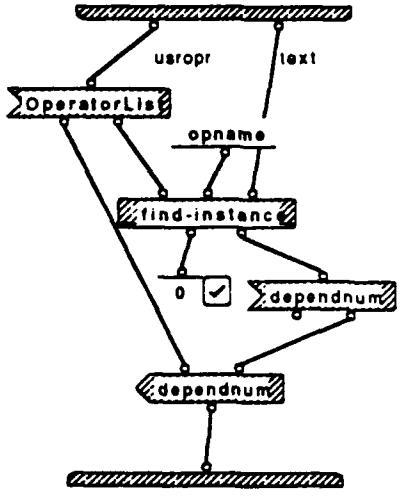

## Operator/set-attributes 1:1



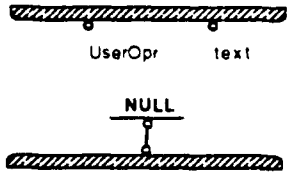
## Operator/create 1:1



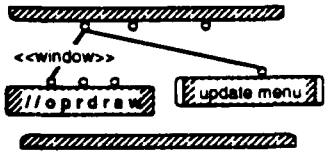

## Operator/create 1:1get instance 1:2



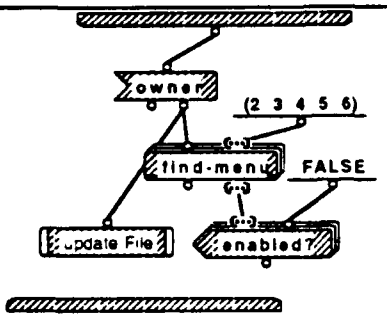

## Operator/create 1:1get instance 2:2



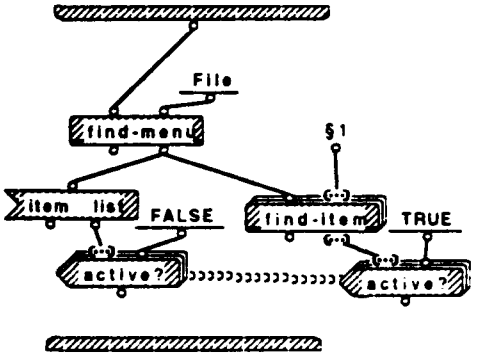
## UserOpr/activeview 1:1




## UserOpr/activeview 1:1update menu 1:1

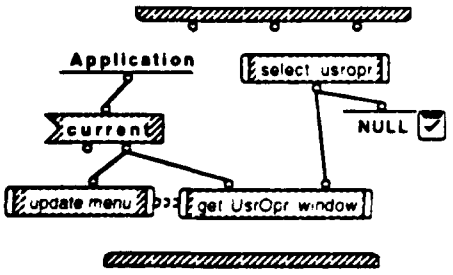



## UserOpr/activeview 1:1update menu 1:1update File 1:1



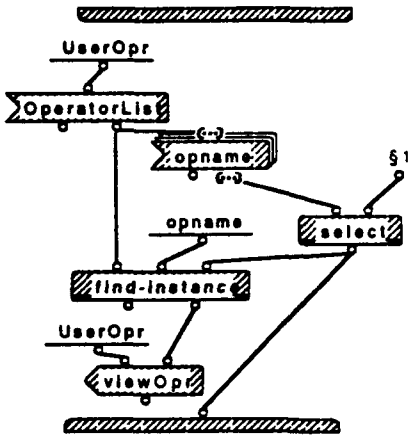

§1. ("Print..." "Page Setup...")

## UserOpr/viewop 1:1



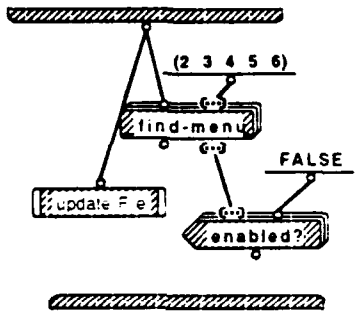

Uses select dialog to display available user operators. Takes the one that was selected and displays it in the View User Operator -- window. Concatenates the name of the user operator being displayed to the window name so it gets displayed in the title bar.

## UserOpr/viewop 1:1select usropr 1:1

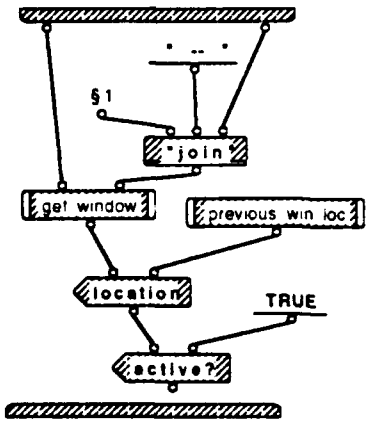



§1. CHOOSE OPERATOR

## UserOpr/viewop 1:1update menu 1:1

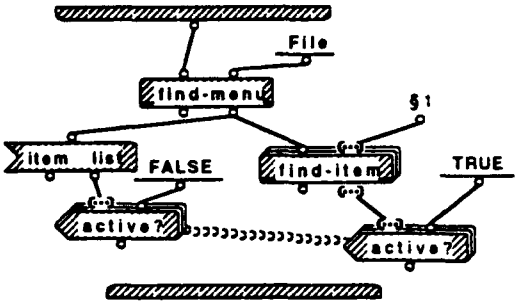



## UserOpr/viewop 1:1get UsrOpr window 1:1



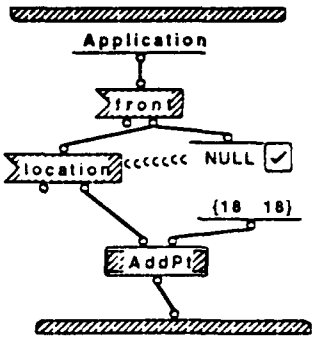

§1. User-Defined Operator

## UserOpr/viewop 1:1update menu 1:1update File 1:1

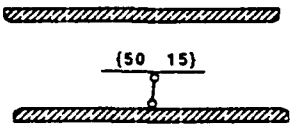



§1. ("Print..." "Page Setup...")

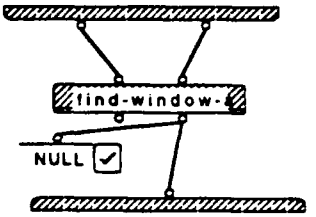
## UserOpr/viewop 1:1get UsrOpr window 1:1previous win loc 1:2



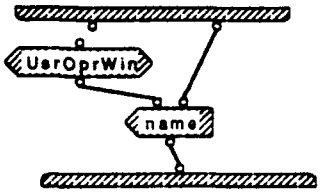

## UserOpr/viewop 1:1get UsrOpr window 1:1previous win loc 2:2



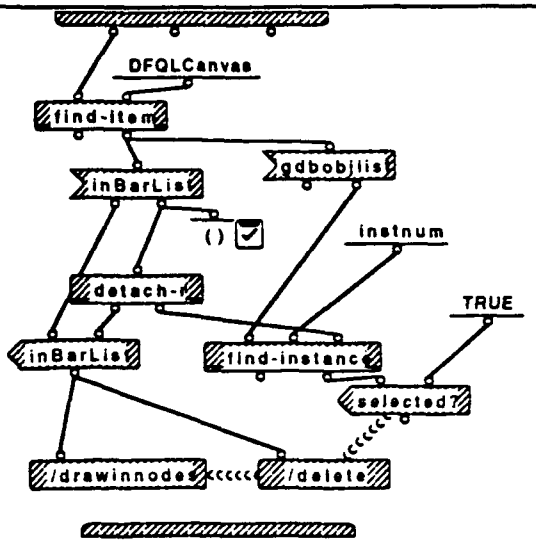

## UserOpr/viewop 1:1get UsrOpr window 1:1get window 1:2



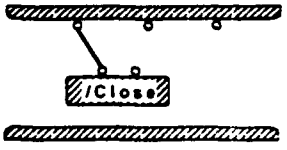

## UserOpr/viewop 1:1get UsrOpr window 1:1get window 2:2



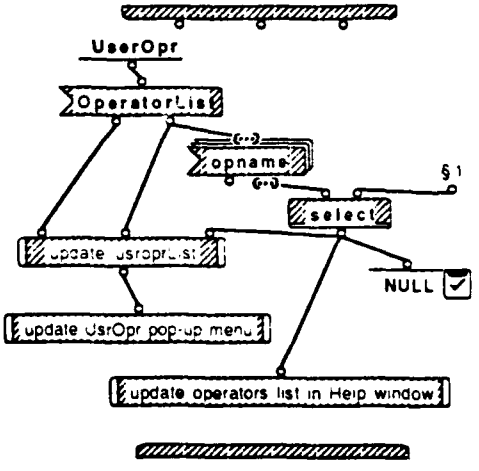
## UserOpr/delete input 1:1




## UserOpr/cancel 1:1

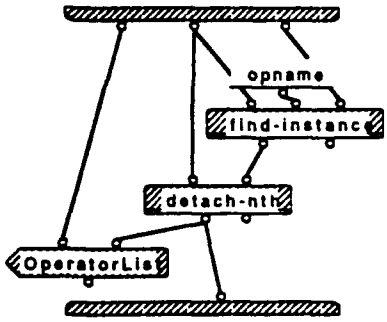



## UserOpr/delop 1:1

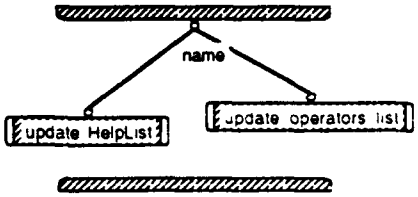



§1. DELETE OPERATOR

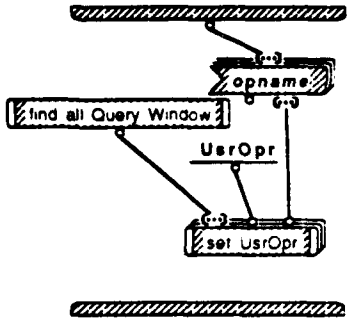
## UserOpr/delop 1:1update usroprList 1:1



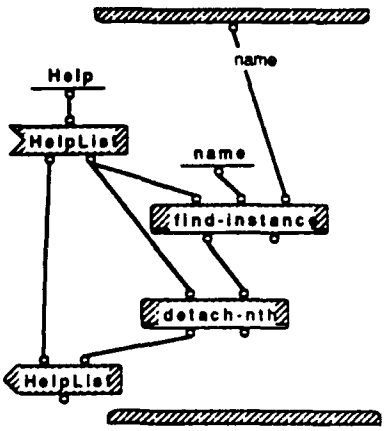

## UserOpr/delop 1:1update operators list in Help window 1:1



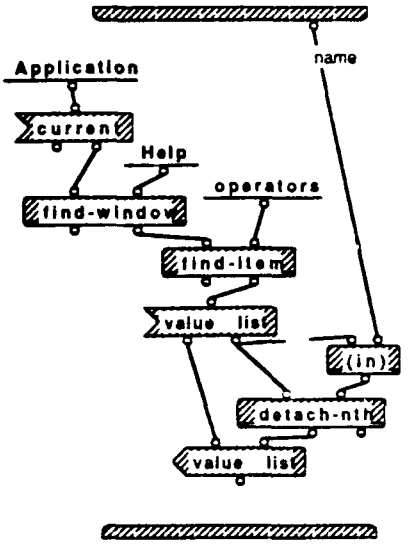

## UserOpr/delop 1:1update UsrOpr pop-up menu 1:1



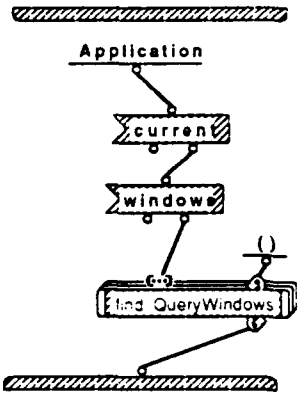

## UserOpr/delop 1:1update operators list in Help window 1:1update HelpList 1:1



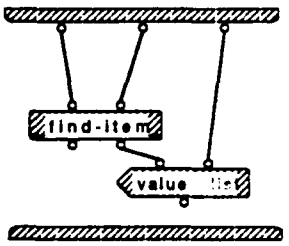
## UserOpr/delop 1:1update operators list in Help window 1:1update operators list 1:1



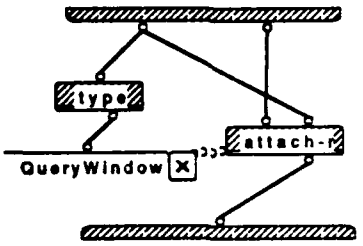

## UserOpr/delop 1:1update UsrOpr pop-up menu 1:1find all Query Window 1:1



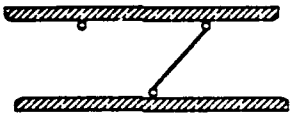

## UserOpr/delop 1:1update UsrOpr pop-up menu 1:1set UsrOpr 1:1



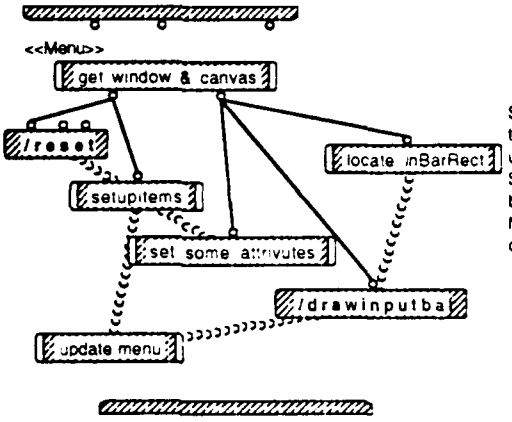
## UserOpr/delop 1:1 update UsrOpr pop-up menu 1:1 find all Query Window 1:1 find QueryWindows 1:2




## UserOpr/delop 1:1 update UsrOpr pop-up menu 1:1 find all Query Window 1:1 find QueryWindows 2:2

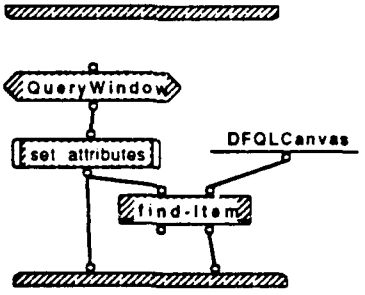


## UserOpr/newusrop 1:1

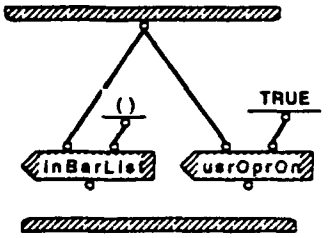



Sets up Query Window screen to accept definition of a new user defined operator. Setupitems turns off all buttons and menu selections NOT associated with user op definition.

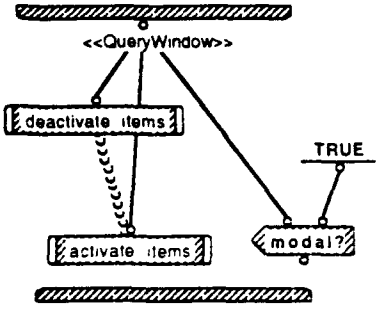
## UserOpr/newusrop 1:1 get window & canvas 1:1



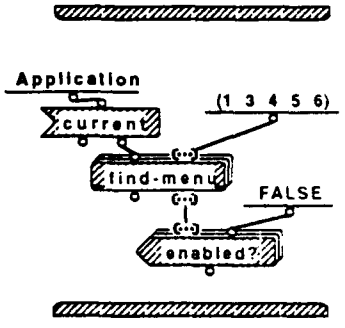
## UserOpr/newusrop 1:1set some attriuutes 1:1



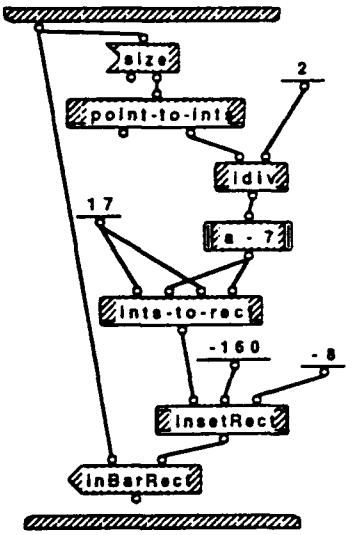

## UserOpr/newusrop 1:1setupitems 1:1



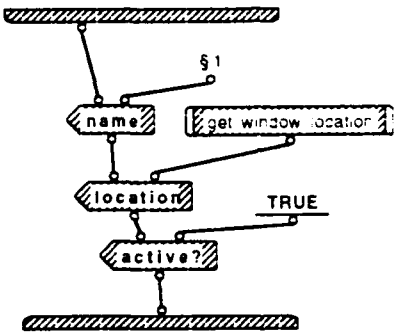

## UserOpr/newusrop 1:1update menu 1:1



## UserOpr/newusrop 1:1locate inBarRect 1:1

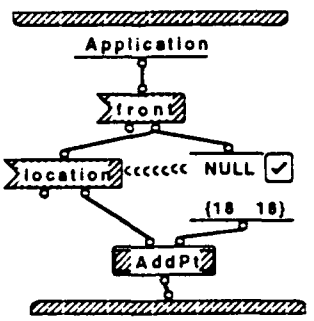



## UserOpr/newusrop 1:1get window & canvas 1:1set attributes 1:1

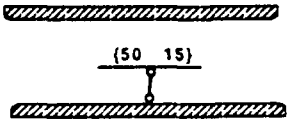



§1. Operator Definition Window

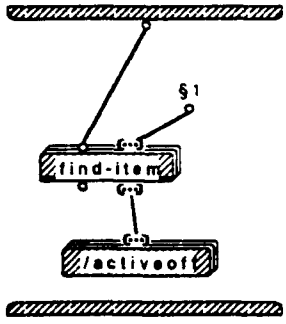
## UserOpr/newusrop 1:1get window & canvas 1:1set attributes 1:1get window location 1:2




## UserOpr/newusrop 1:1get window & canvas 1:1set attributes 1:1get window location 2:2

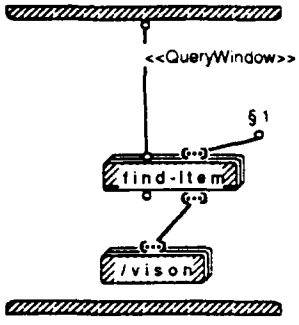


## UserOpr/newusrop 1:1setupitems 1:1deactivate items 1:1



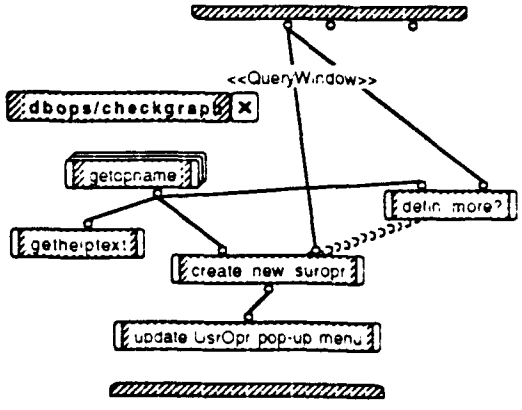

§1. (Run New Open "Save as" Save Reset)

## UserOpr/newusrop 1:1setupitems 1:1activate items 1:1



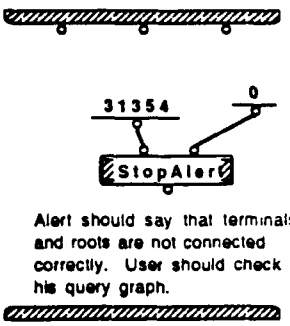

Store Cancel "Delete Input" "Clear All")

## UserOpr/storeop 1:2



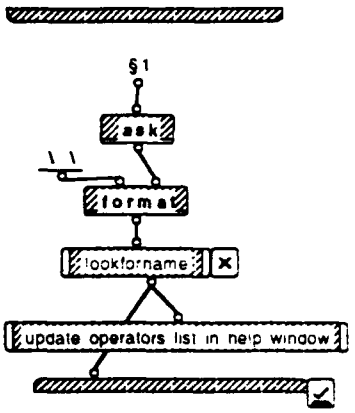

First checks the user op for correct connections. Then gets the name for the operator (getopname) and the help message for it (gethelptext) from the user. Then adds it in correct alphabetical order (determined by getopname -- when it also checks for attempted use of already used names) to the persistent list of all user defined operators.

## UserOpr/storeop 2:2



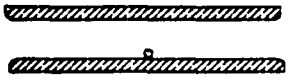

Alert should say that terminals and roots are not connected correctly. User should check his query graph.

## UserOpr/storeop 1:2getopname 1:2

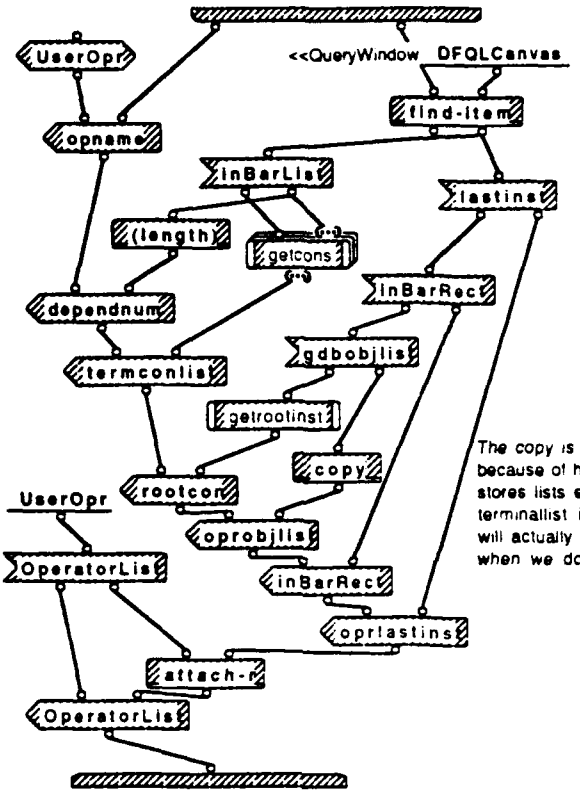



§1. Enter the desired name for this operator:

## UserOpr/storeop 1:2getopname 2:2

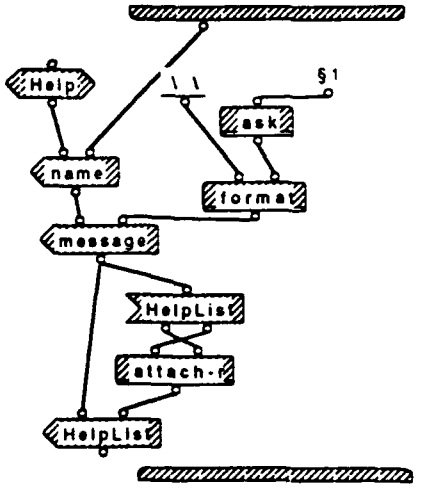


## UserOpr/storeop 1:2create new suropr 1:1



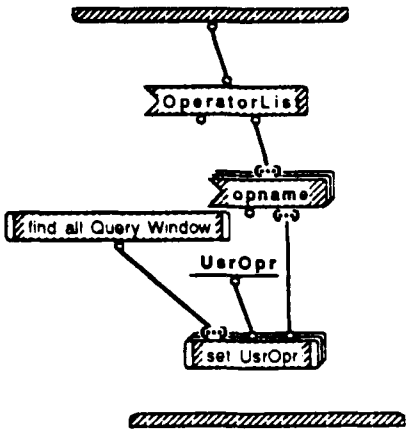

*The copy is required* because of how prograph stores lists etc. The terminallist in the persistent will actually get messed up when we do //cancel otherwise.

## UserOpr/storeop 1:2gethelptext 1:1

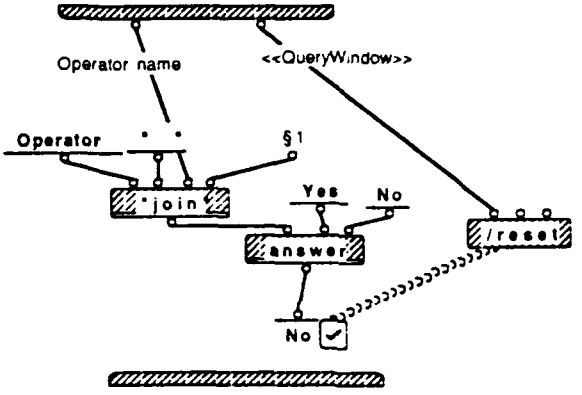



§1. Please enter help information for your new operator.
(Use option-return to start new lines.)

## UserOpr/storeop 1:2update UsrOpr pop-up menu 1:1

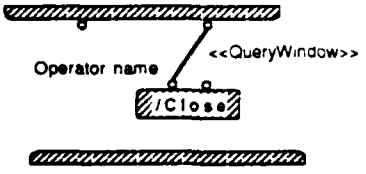



## UserOpr/storeop 1:2defin more? 1:2

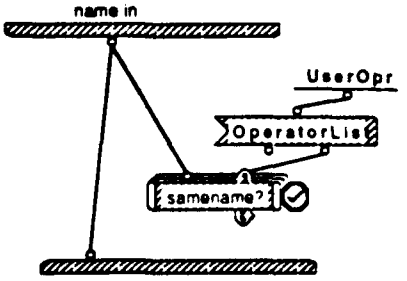



§1. saved. Define any more?

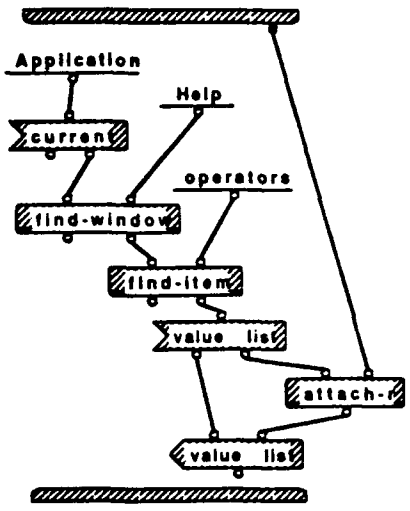
## UserOpr/storeop 1:2defin more? 2:2



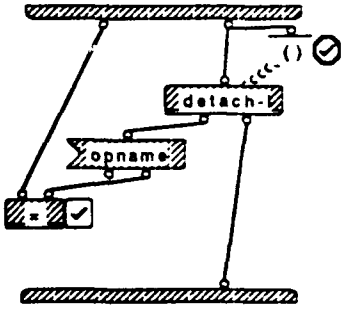

## UserOpr/storeop 1:2getopname 1:2lookforname 1:1



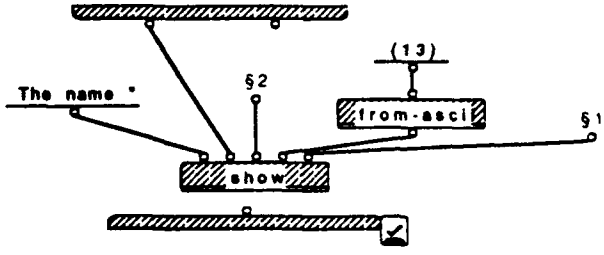
## UserOpr/storeop 1:2getopname 1:2update operators list in help window 1:1




## UserOpr/storeop 1:2getopname 1:2lookforname 1:1samename? 1:2

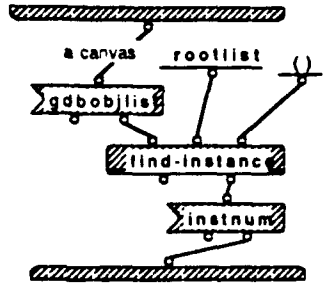



## UserOpr/storeop 1:2getopname 1:2lookforname 1:1samename? 2:2

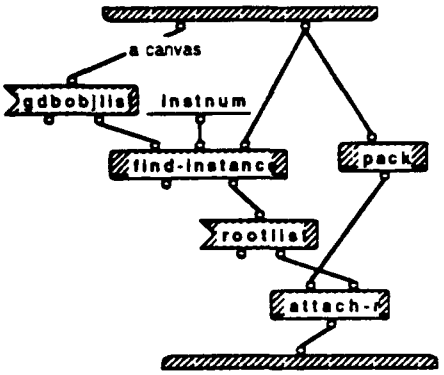



§1. Please enter a different name.
§2. " is already in use!

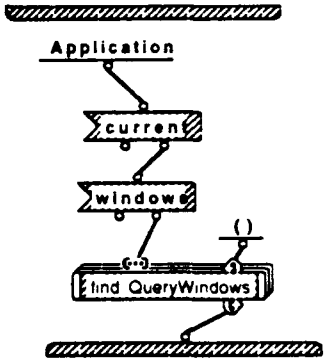
## UserOpr/storeop 1:2create new suropr 1:1getrootinst 1:1



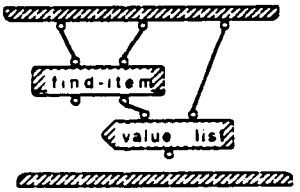
## UserOpr/storeop 1:2create new suropr 1:1getcons 1:1



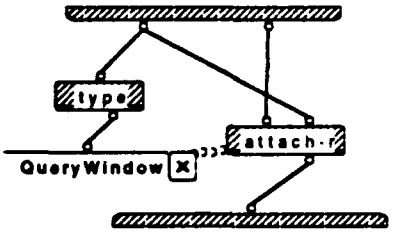

## UserOpr/storeop 1:2update UsrOpr pop-up menu 1:1find all Query Window 1:1



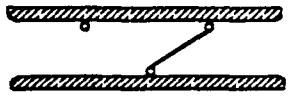

## UserOpr/storeop 1:2update UsrOpr pop-up menu 1:1set UsrOpr 1:1



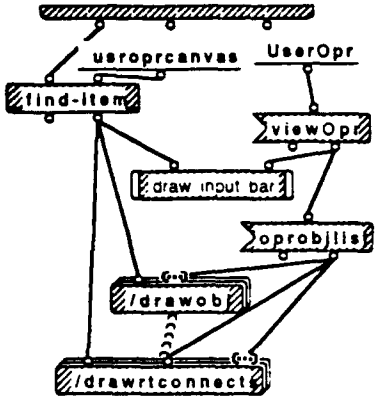

## UserOpr/storeop 1:2update UsrOpr pop-up menu 1:1find all Query Window 1:1find QueryWindows 1:2



## UserOpr/storeop 1:2update UsrOpr pop-up menu 1:1find all Query Window 1:1find QueryWindows 2:2

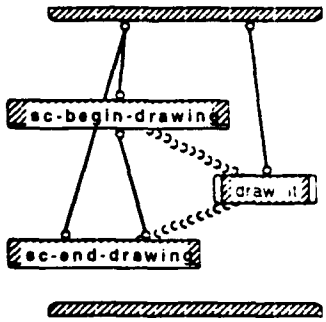


## UserOpr/oprdraw 1:1

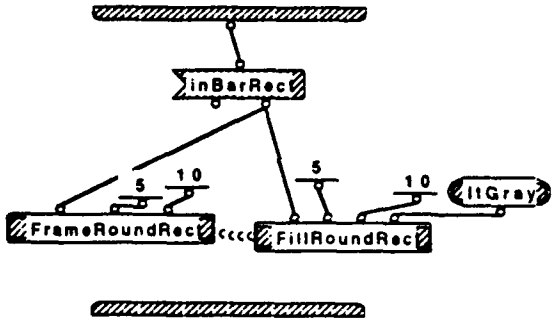



Basically the same as gdbopr draw except it must account for the input bar and any connections to it.

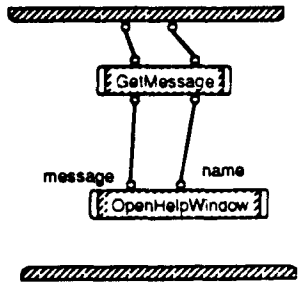
## UserOpr/oprdraw 1:1draw input bar 1:1



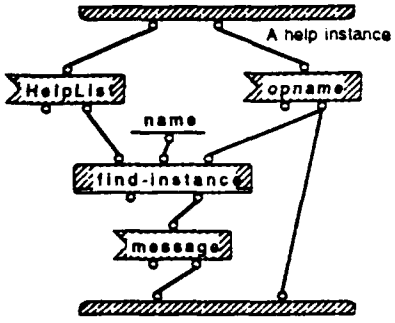

## UserOpr/oprdraw 1:1draw input bar 1:1draw it 1:1



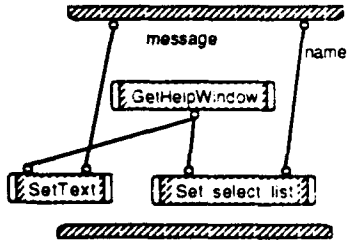
## Help/show help 1:1



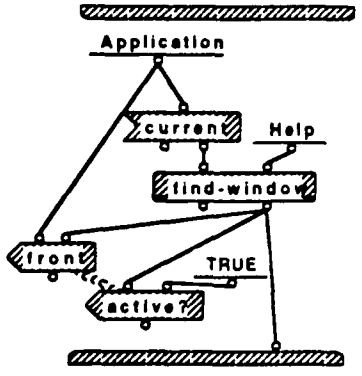

## Help/show help 1:1GetMessage 1:1



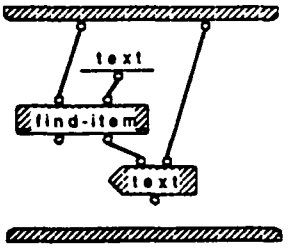

## Help/show help 1:1OpenHelpWindow 1:1



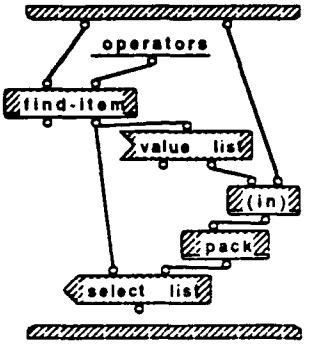

## Help/show help 1:1OpenHelpWindow 1:1GetHelpWindow 1:1



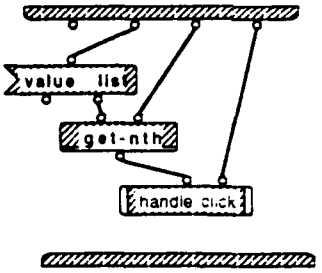
## Help/show help 1:1OpenHelpWindow 1:1SetText 1:1



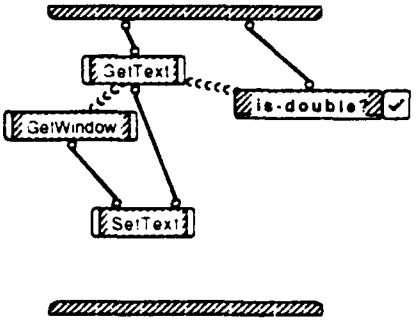

## Help/show help 1:1OpenHelpWindow 1:1Set select list 1:1



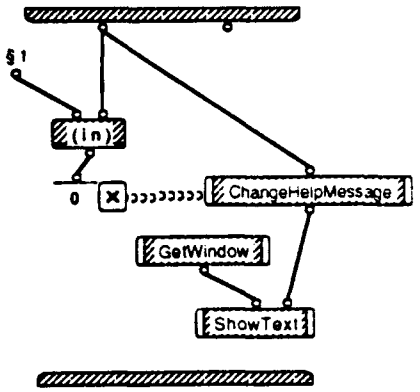

## Help/help click 1:1




## Help/help click 1:1handle click 1:3

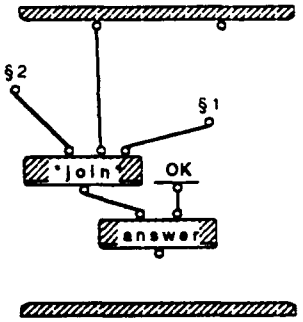


## Help/help click 1:1handle click 2:3



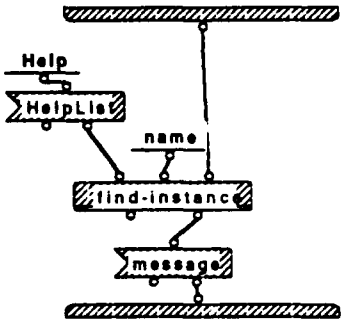

§1. (select join diff union project groupcnt eqjoin groupAllsatisfy groupNsatisfy groupavg groupmax groupmin intersect DISPLAY SDISPLAY)

## Help/help click 1:1handle click 3:3

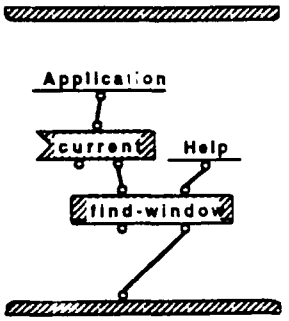



§1. ' can not be changed!
§2. The description of '

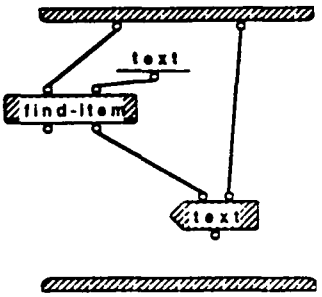
## Help/help click 1:1handle click 1:3GetText 1:1



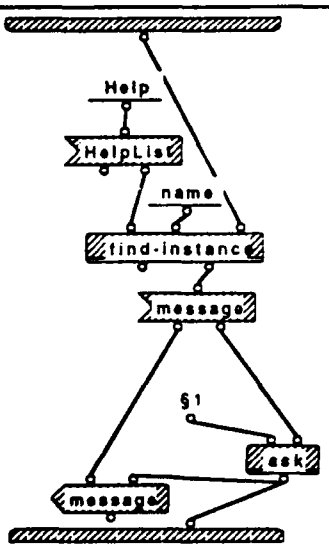
## Help/help click 1:1handle click 1:3GetWindow 1:1




## Help/help click 1:1handle click 1:3SetText 1:1

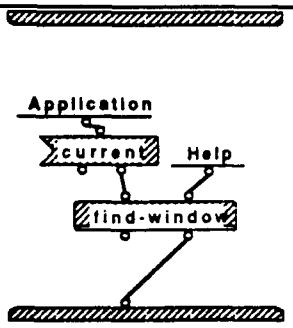



## Help/help click 1:1handle click 2:3ChangeHelpMessage 1:1

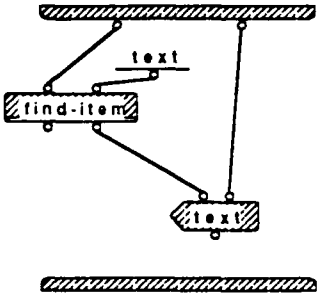



§1 You may edit the text below.(Use option-return to start new lines.)

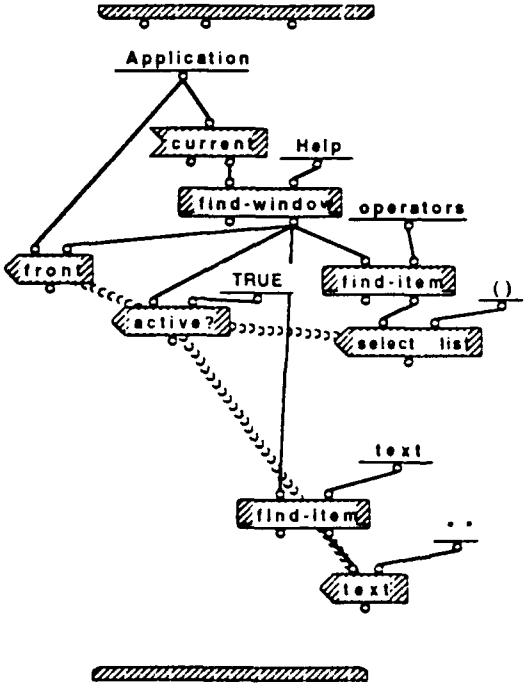
## Help/help click 1:1handle click 2:3GetWindow 1:1




## Help/help click 1:1handle click 2:3ShowText 1:1




## Help/ActivateHelp 1:1



# BIBLIOGRAPHY

Shneiderman Ben, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, 1987.

Kim, W., and Lochovsky, F. H., *Object-Oriented Concepts, and Applications*, Addison-Wesley, 1989.

Wu, C. T., *OOP + Visual Dataflow Diagram = Prograph*, Journal of Object-Oriented Programming, pp. 71-75, June 1991.

Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 1988.

# INITIAL DISTRIBUTION LIST

| | |
|---|---|
| Dudley Knox Library<br>Code 52<br>Naval Postgraduate School<br>Monterey, CA 93943 | 2 |
| Chairman, Code CS<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943 | 2 |
| Professor C. Thomas Wu<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943 | 2 |
| Li, Chang-Tsun<br>Weapon System Department<br>Chung Cheng Institute of Technology<br>Tashi, Taoyuan, Taiwan, R.O.C. 33509 | 3 |
| Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22304-6145 | 2 |