WEIGHTED PARTIAL PATTERN MATCHING

Dr. W.G. Rudd
Rukmani Tekchandani

Oregon State University
Corvallis OR 97331

1. INTRODUCTION:

Pattern matching is the process of comparing two objects to see if they are similar to each other. Objects can be physical objects, situations, facts or events. The criteria of matching can be exact or partial. Traditional pattern matching systems rely on exact matches, in which a small set of predefined pattern matching rules are applied in all-or-none fashion. Partial matching is superior to exact matching because it can be used to identify the best from a set of options without insisting on a perfect match with what is required.

Partial matching is important in retrieval from database and knowledge base systems. We need to retrieve data from databases that are "like" data we are looking for. For example, a particular book can be selected from a collection of books in a library by partial matching a description of the subject we are interested in against a set of descriptions from a database that describes the library holdings.

Calculating how similar objects are to each other is the fundamental problem of partial matching. There have been two basic approaches to similarity:

1. Geometric Approach:

Each object is represented by a point in some coordinate space so that the distance between points in that space is a measure of similarities between the objects. The space is assumed to be Euclidian and the purpose of the similarity relations is to embed the objects in a space of minimum dimensionality on the basis of the observed similarities.

2.Feature-theoretical approach:

Each object a is characterized by a set of features, denoted A, and the observed similarity of a to b denoted $s(a,b)$, is expressed as a function of their common and distinctive features[E. Rosch]. This view is shared in other literature as partial match of a and b.

PM(a,b) = (a*b, a - a*b, b - a*b)   [F. Hayes-Roth]

where, a*b denotes the common properties of a and b and a - a*b and b - a*b denote the properties of a and b not included in a*b. Similarity in this approach is a feature matching function that measures the degree to which two sets of features match each other.

Our approach to similarity resembles feature-theoretical approach in that we compute the similarity between objects depending on similarity of their individual features. Some of the important points of our partial pattern matching technique are:

1. Similarity functions for features can be tailored to the particular application in hand.

2. Features can be assigned weight by the user, thus relative importance of feature can be changed according to need.

3. Values of all attributes may not be known:

In section 2 we briefly describe few of the many applications of partial matching. Section 3 contains the theory behind the technique and in section 4 we describe its implementation with database management facilities. In the last section we describe two of the experiments carried out and analyse the technique in the light of results.

2. APPLICATIONS:

1. Information Retrieval:

The traditional way of retrieving a piece of information from some collection, for example, a word from a database, is to search the collection for the information we are looking for. If the information we seek is not present exactly as specified we look for the most similar or best-matching information. This can be achieved by a partial match technique. For example, in agricultural studies, experiments are done and the results are stored in the database. Given a description of experiment, we would like to find the experiment that resembles it most closely.

2. Template-Matching:

Templates also known as frames are usually hierarchically organized descriptions of observable phenomena. Once the best-matching templates are found, data can be interpreted by imposing the frame structure upon them. For example, in a speech understanding system, templates are phrases of language and data to be matched are arrays of words.

Our Weighted partial pattern matching algorithm has been used for template matching in Investment Portfolio Expert Systems[M. Fry]. The investor and economic conditions are matched with previously used portfolio templates.

3. Classification:

Classifying an object is relating a description of an object to a pre-enumerated set of classes. Thus if set A contains all the different kinds of computers like mini, micro, mainframes, workstations etc., partial match allows to classify a given computer into one of the classes.

3. THEORY:

We use feature-vectors that consist of list of attribute-value pairs to describe objects. In comparing a pair of objects, we compute a similarity index for each of the attribute. We compute V, a weighted sum of the similarity indices, where the weights are assigned by the user. To account for the fact that attribute-values may not be known for some attributes, we compute two other measures of similarity. V_{min} is the value of V we could get, if all the missing values would have ended in contributing zero as similarity index, and V_{max}, which is the value of V we would have obtained, had all the missing values had the similarity index of 1.

Given two descriptions, let

N be the set of indices of all the attributes,

M be the set of all indices of attributes for which values are known in both the descriptions,

w_i be the weight for attribute i,

s_i be the similarity index of attributte i

Then,

we can formally define $SM(A,B)$ to be similarity measure of two objects A and B as follows:

$$SM(A,B) = (V, V_{\min}, V_{\max}) \tag{1}$$

where

$$V = \frac{\sum_{i \epsilon M} w_i\, s_i}{\sum_{i \epsilon M} w_i} \tag{2}$$

V is a weighted, normalized sum of similiarity indices for attributes with corresponding values in both objects.

$$V_{min} = \frac{\sum_{i \epsilon M} w_i\, s_i + \sum_{i \epsilon N-M} w_i .0}{\sum_{i \epsilon N} w_i} \tag{3}$$

Vmin is a weighted, normalized sum of similiarity indices for matching attributes and attributes whose values are missing in either one or both of objects. In calculating *Vmin* we assume that similarity index for attribute whose value is missing is zero.

$$V_{max} = \frac{\sum_{i \epsilon M} w_i \, s_i \ \sum_{i \epsilon N-M} w_i \, .1}{\sum_{i \epsilon N} w_i} \tag{4}$$

V max is a weighted, normalized sum of similiarity measures for matching attributes and attributes whose values are missing in either one or both of objects. In calculating *Vmax* we assume that similarity index for attribute whose value is missing is 1.

4. IMPLEMENTATION:

This algorithm has been implemented in a pattern matching system which helps in maintaining database of alternative descriptions of object and given any current description finds the best-matches from the database. We have written this system in Franz lisp running on 4404 AI workstations. Uniflex operating system was used.

Following information is needed by the system:

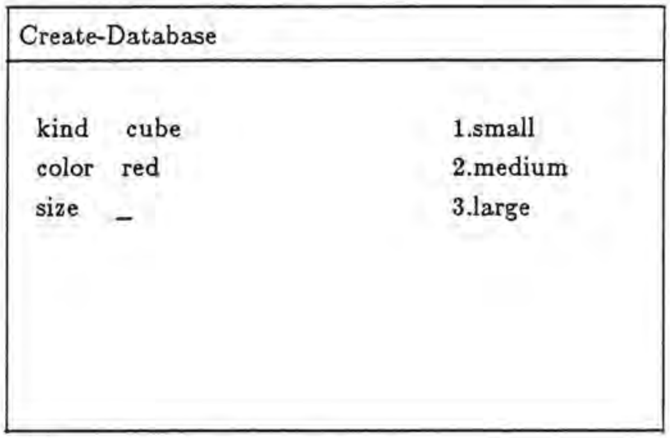
1. Descriptions of objects: The objects which are used to match against the given descriptions are referred subsequently as stored descriptions and their collection is referred to as the database.

2. Weight of the attribute: Each attribute has a weight assigned by the user and is a property of the attribute.

3. Similarity index: Similarity indices are calculated by invoking similarity functions. The functions are either built-in for some general types of attributes or are supplied by the user.

Database:

The database is a collection of object descriptions. Database management facilities are available in the system. Thus object descriptions can be entered, edited and any description can be viewed directly if its number is known. Descriptions are represented as lists of attribute-value pairs. This representation is transparent to user. The attribute-names are displayed on screen for which user puts in the values. As these values are used by similarity-functions to calculate similarity-indeces, any values entered which are not known to these functions will cause an error. For the same reasons, the system can't handle any spelling mistakes. When a user enters the value of attribute, valid values for that attribute are displayed. The user is asked to select the corresponding number of the value. That number is translated into the value and entered in the representation.



Create-Database

kind	cube	1.small
color	red	2.medium
size	_	3.large

--> ((kind cube) (color red)...)

Fig1: Screen-view of entering
object description

Internal representation
of description

Directory

The directory contains one entry for each attribute. The directory is created by the expert who is setting up the system for a particular application. Similarity functions are not written in directory but are loaded from an external file. Each entry contains an attribute-name, its weight, its possible values and name of the function used to compute its its similarity index.

Suppose, that we have the following information for an object

Attribute name	Valid values	Weight	Function name
a1	v1,c1	1.0	a1-distance
a2	v2,c2	0.4	a2-distance
a3	nil	0.6	a3-distance
a4	v4	0.9	a4-distance

Then directory contents look like this:

```
(a1 1.0 a1-distance (v1 c1))
(a2 0.4 a2-distance (v2 c2))
(a3 0.6 a3-distance (nil))
(a4 0.9 a4-distance (v4))
```

Similarity Definitions

Following similarity-functions have been implemented.

• *Number-distance:*

This function calculates similiarity index for number. The valid numbers are integers between 0 and 5. This function can be modified to change upperlimit of 5 to some other integer.

Let T be the total numbers possible, and let 'a' and 'b' be the values of attributes. Then,

Number-distance(a,b) = 1 - (abs(a - b) / T)

Example

T = 5

a = 1

b = 4

Number-distance(a,b) = 0.6

• *Shade-distance:*

This function calculates similiarity index for shades of gray. The valid values are verylightgray, lightgray, gray, darkgray and black. These values can be arranged on a distance scale as follows:

1. Verylightgray
2. Lightgray
3. Gray
4. Darkgray
5. Black

Let val1 and val2 be the two shades to be compared. And let get-number be the function which returns the number associated with the shades. Then

Shade-distance(val1, val2) = 1 - (abs(get-number(val1) - get-number(val2)) / 5

Example

val1 = gray

val2 = darkgray

Shade-distance(val1, val2) = 0.8

• *Color-distance:*

This function calculates similiarity index for colors found in spectrum. The valid values are violet, indigo, blue, green, yellow, orange and red. These values can be arranged on a distance scale as follows:

1. Violet
2. Indigo
3. Blue

4. Green
5. Yellow
6. Orange
7. Red

Let val1 and val2 be the two colors to be compared. And let get-number be the function which returns the number associated with the colors. Then

Color-distance(val1, val2) = 1 - (abs(get-number(val1) - get-number(val2)) / 7

Example

val1 = Red
val2 = Green

Shade-distance(val1, val2) = 0.57 (Rounded)

•*Shape-distance:*

This function calculates similiarity index for block shapes. The valid values are sphere, cone, pyramid, cube and cylinder. These values can be arranged on a distance scale as follows:

1. sphere
2. Cone
3. Cylinder
4. Pyramid
5. Cube

Let val1 and val2 be the two shapes to be compared. And let get-number be the function which returns the number associated with the shapes. Then

Shape-distance(val1, val2) = 1 - (abs(get-number(val1) - get-number(val2)) / 5

Example

val1 = sphere
val2 = Cube

Shade-distance(val1, val2) = 0.2

•*String-distance:*

[K.Abe]

This function calculates similiarity index for given strings of symbols. For given two strings A = a1, a2,... am and B = b1, b2... bn calculate the following values of d(i,j) iteratively:

d(1,1) = h(1,1)
d(i,j) = min { d(i-1, j-1), d(i, j-1), d(i-1, j)} + h(i,j)

String-distance(A,B) = 1 - (d(m,n) / m+n)

5. EXPERIMENTS AND RESULTS:

For the purpose of verifying the algorithm and analysing the similarity measures two simple experiments have been carried out.

Experiment#1:

Objects are square regions with geometrical patterns. The only patterns possible are vertical bars in a square region, with their number varying from 1 to 5 and the color varying in shades of gray. The objects chosen are very simple, so that they can be displayed on the screen. For the internal representation, these regions can be described by two attributes:

a) shade (of bars).

b) number (of bars).

As 5 different shades of gray are possible(again so chosen because Franz lisp Graphics package supports only these 5 shades) we have 25 different regions possible to be compared. More combinations possible if we take into account the regions with missing information.

In one of the experiment similarity measure of a region having 5, lightgray bars with other 5 regions in database was computed. The database contained the following descriptions:

Desc#	shade	number
1.	lightgray	5
2.	gray	5
3.	black	1
4.	missing	4
5.	darkgray	missing

Note: Please refer to Fig 1 for the pictorial representation of these objects.

Following results were obtained when both attributes weighed same i.e. weight for shade and number is 1:

	Desc1	Desc2	Desc3	Desc4	Desc5
Vf	1.0	0.9	0.3	0.8	0.6
V_{min}	1.0	0.9	0.3	0.4	0.3
V_{max}	1.0	0.9	0.3	0.9	0.8

Note: Please refer to Fig 2 for the bar graph of similarity measures.

Following results were obtained when shade was weighed less than number i.e. weight of shade is .5 and weight of number is 1.

	Rec1	Rec2	Rec3	Rec4	Rec5
V	1.0	0.93	0.26	0.8	0.6
V_{min}	1.0	0.93	0.26	0.53	0.19
V_{max}	1.0	0.93	0.26	0.86	0.86

Remarks: 1. There is no difference in similarity components in Desc #1, #2 and #3 as there is no information missing.

2. Similarity measure (V) is 1.0 with Desc#1 showing that it exactly matches the current description and decreases from desc#1 to desc#2 to #3 as black is more different from lightgray than gray.

3. Vmin and Vmax for Desc#4 and Desc#5 show the minimum and maximum similarity possible when there is missing information.

4.The difference between Vmin and Vmax increases if the missing information weighs more than the other information and decreases if missing attribute weighs less. Ex. 4 and 5

Experiment#2

Objects are situations from block-world. The block-world has 3 blocks which can be stacked one on top of another. The possible attributes are color, on and top.

attribute	*values*
color	red, blue or green
on	block or table
top	clear or nonclear

The database for this experiment contained the following object descriptions:

Desc#	color1	color2	color3	on1	on2	on3	top1	top2
1.	red	blue	green	table	block	block	notclear	notclear
2.	red	green	blue	table	block	block	notclear	notclear
3.	red	blue	green	table	block	table	notclear	clear
4.	red	green	blue	table	table	table	clear	clear
5.	red	blue	green	table	block	missing	notclear	missing

This description was matched to each of the descriptions in database.

color1	color2	color3	on1	on2	on3	top1	top2
red	blue	green	table	block	block	notclear	notclear

Following similarity measures were obtained when all of the atributes weighed equal.

	Rec1	Rec2	Rec3	Rec4	Rec5
V	1.0	0.92	0.77	0.48	1.0
V_{min}	1.0	0.92	0.77	0.48	0.77
V_{max}	1.0	0.92	0.77	0.48	1.0

Remarks: 1. This experiment supports the results got from experiment#1.

2. Vmin of Desc#5 is equal to similarity measures of Desc#3 as known features of Desc#5 match that of Desc#3

Conclusion:

We have proposed in this paper a method of computing similarity measures and using it to retrieve best matching data from collections of records. The results from experiment look convincing. This technique has been successfully implemented in a portfolio system. At present we are implementing system in managing a real database of agricultural crops. The objective is to retrieve best-matching records which resemble to given data.

Further work is needed in this system. Similarity definitions are the backbone of whole procedure. One of the drawback is that these functions can be written only in Lisp. We would like to implement automatic programming techniques so that function definitions in mathematical form could be translated into Lisp code.

Bibliography

[1] Frederick Hayes-Roth: The Role of Partial and Best Matches in knowledge-based systems.

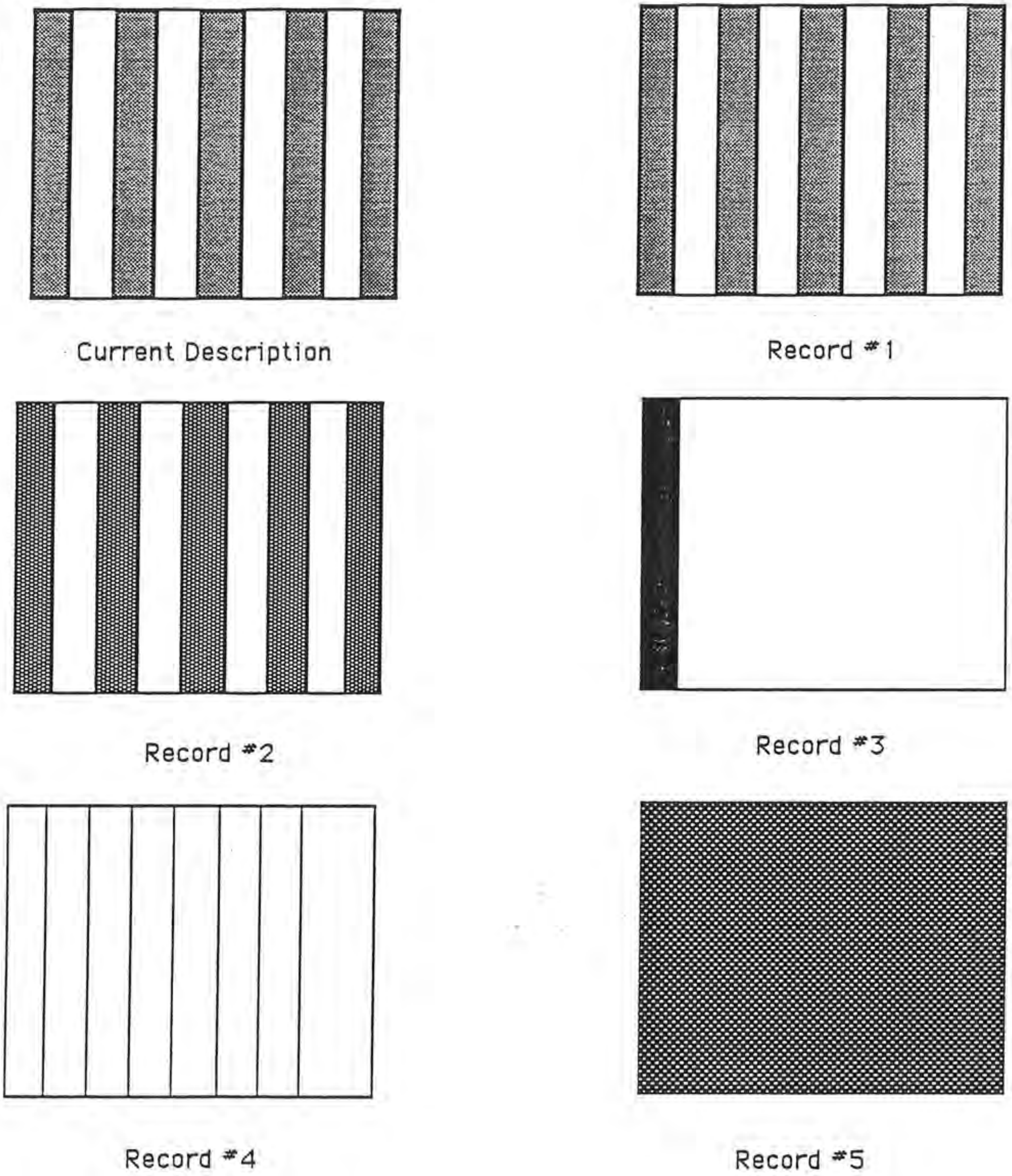
[2] W.G. Rudd and George R.Cross: Design of an expert system for Insect Pest Management.

[3] Barry Shane, Mitchel fry and Wilbur Widiucus: Consultation among multiple knowledge bases in an investment portfolio expert system.

[4] E.Rosch: Cognition and Categorization

[5] Bruce W. Porter, E.Ray Baress: Protos

[6] Keiichi ABE & Noborn Sugita: Distances between strings of symbols in Proc. of 6th Int. Conf. on Pattern Recognition.



Current Description

Record #1

Record #2

Record #3

Record #4

Record #5

Fig. 1. Pictorial Representation of Objects in Experiment 1

WEIGHTED PARTIAL PATTERN MATCHING
USER'S GUIDE

RUKMANI TEKCHANDANI
MARCH 20 1987

CONTENTS

1.TASK:

To provide facilities for managing database of object descriptions and to compute their similarity with the given object descriptions.

2.WHAT IS "WEIGHTED PARTIAL PATTERN MATCHING"?

Pattern matching is the process of comparing two patterns to see if one is similar to another. Objects can be physical objects, situations, facts or events. This method uses partial matching in the sense that objects do not have to be exactly similar to each other to match one another.

Objects are represented as a collection of features. Features may be assigned weights depending on their importance in the entire description. Similarity between objects is a linear combination of the similarity of features and their weights.

3.Terminology:

Object:

Object is an item which is to be compared. For example, chair, experiment, field description are some of the objects.

Attribute:

Each object can be described as a set of features. Features are also called attributes. Thus, leg is attribute of chair.

Value:

Value is value of attribute. Thus leg is an attribute which may have value 4.

Description: Description of an object is set of attribute-value pairs for each feature.

Example: If a chair has 4 legs and is made of wood then in can be described in feature-vector representation as

((legs 4) (made-of wood))

Database:

It is a collection of object descriptions.

Similiarity-Functions:

These functions compare given values and return a similiarity index (0 - 1) depending on how close the values of attributes are to each other.

Example:

(shade-distance gray darkgray)

.8

Weight:

Weight is a number between 0 to 1. It is assigned to attribute by user. Weight defines the importance of attribute in respect to remaining attributes of description.

Directory:

Directory is a list of attributes with their weights and function-names.

THEORY:

We use feature-vectors that consist of list of attribute-value pairs to describe objects. In comparing a pair of objects, we compute a similarity index for each of the attribute. We compute V, a weighted sum of the similarity indices, where the weights are assigned by the user. To account for the fact that attribute-values may not be known for some attributes, we compute two other measures of similarity. V_{min} is the value of V we could get, if all the missing values would have ended in contributing zero as similarity index, and V_{max}, which is the value of V we would have obtained, had all the missing values had the similarity index of 1.

Given two descriptions, let

N be the set of indices of all the attributes,

M be the set of all indices of attributes for which values are known in both the descriptions,

w_i be the weight for attribute i,

s_i be the similarity index of attributte i

Then,

we can formally define $SM(A,B)$ to be similarity measure of two objects A and B as follows:

$$SM(A,B) = (V, V_{\min}, V_{\max}) \tag{1}$$

where

$$V = \frac{\sum_{i \epsilon M} w_i s_i}{\sum_{i \epsilon M} w_i} \tag{2}$$

V is a weighted, normalized sum of similiarity indices for attributes with corresponding values in both objects.

$$V_{min} = \frac{\sum_{i \epsilon M} w_i \, s_i + \sum_{i \epsilon N-M} w_i \,.0}{\sum_{i \epsilon N} w_i} \tag{3}$$

V min is a weighted, normalized sum of similiarity indices for matching attributes and attributes whose values are missing in either one or both of objects. In calculating *Vmin* we assume that similarity index for attribute whose value is missing is zero.

$$V_{max} = \frac{\sum_{i \epsilon M} w_i \, s_i \; \sum_{i \epsilon N-M} w_i \,.1}{\sum_{i \epsilon N} w_i} \tag{4}$$

V max is a weighted, normalized sum of similiarity measures for matching attributes and attributes whose values are missing in either one or both of objects. In calculating *Vmax* we assume that similarity index for attribute whose value is missing is 1.

4.DEFINING THE INPUT AND OUTPUT:

Input: Database of descriptions and a current description which has to be matched with each description in the database.

Output:
For each description in database similarity measures are displayed in ascending order.

For example if Desc-X is matched with Desc1, Desc2, Desc3 in the database then output is:

	V	V_{min}	V_{max}
Desc1	-	-	-
Desc2	-	-	-
Desc3	-	-	-

5.HOW TO USE THE SYSTEM

This system is a tool to match descriptions. Before the system could be used for this purpose, it needs to know about:

1. Attributes which will be used to describe objects.
2. Weights of the attributes
3. Similiarity-functions.

Thus, system has to be initialized for a particular task by an expert. By expert, we mean a person who decides the attributes to be used to describe objects, give them weights and writes similiarity functions. This is called setting up the system. Once system is set up, it can be used by any user to match descriptions. Novice user can match observed event with stored events and find the best match, create new database and change weights of attributes if his goal is different from the expert.

6.GETTING STARTED

To avoid explaining everything about using 4404 AI machines, it is assumed you have some familiarity with these machines and how to use them. If not, you should ask a local expert for help. Assuming that you are logged in, you can restore the program from floppy disk.

```
$ restore +dl
$ lisp
-> (load 'match.l)

   t

-> (start)
```

create-directory
view-directory
edit-directory
create-database
view-database
edit-database
match
exit

7.SETTING UP THE SYSTEM:

1. Defining Similiarity-functions.

Similiarity-functions can be selected from the functions available in Functions-file or new functions can be written and appended to Funtions-file. Similiarity-functions accept two arguments which are to be compared and return a value in the range of 0 and 1 depending on the closeness of arguments. Following built-in functions are available::

1.1 Shade-distance:

This function compares different shades of gray. The shades known to to this function are verylightgray, lightgray, gray, darkgray and black.

1.2 Number-distance:

This function compares numbers between 0-5.

1.3 Color-distance:

This function compares different colors found in the spectrum. The colors in the spectrum are violet, indigo, blue, green, yellow, orange and red.

1.4 Shape-distance:

This function compares geometrical shapes. The shapes known to this function are sphere, cube, cone, pyramid.

1.5 Pos-distance:

This function calculates the how near two objects are placed in a room. The object can be in upper left corner, upper right corner, lower left corner, lower right corner and center.

1.6 User-defined functions:

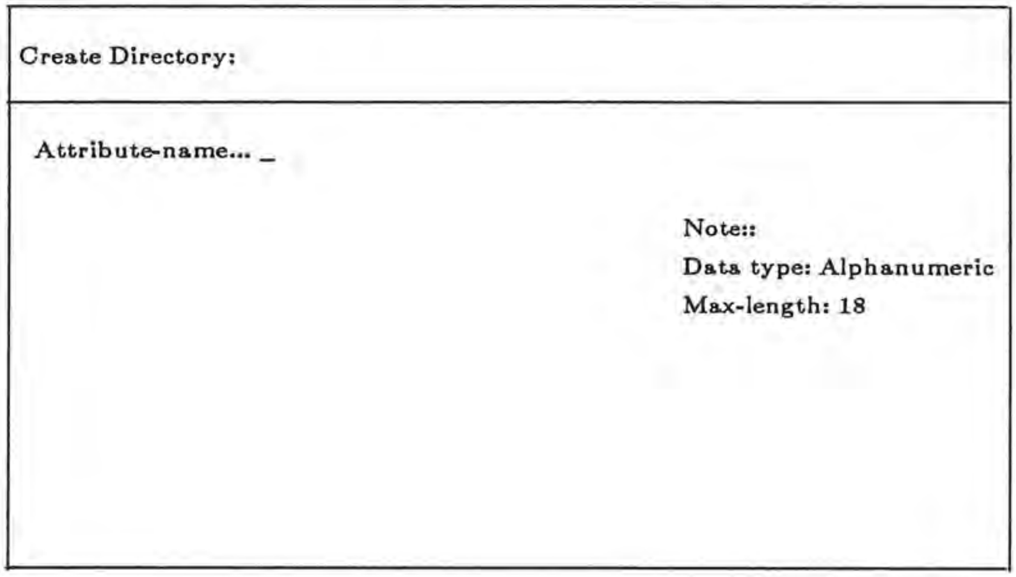
1. Append functions in Function-file.
2. Set *function-list* to include the new functions.

2. Setting Directory.

2.1 To enter attribute in directory:

If menu is being displayed then follow following steps else precede these steps by
-> (start)

1.Select " Create-directory".

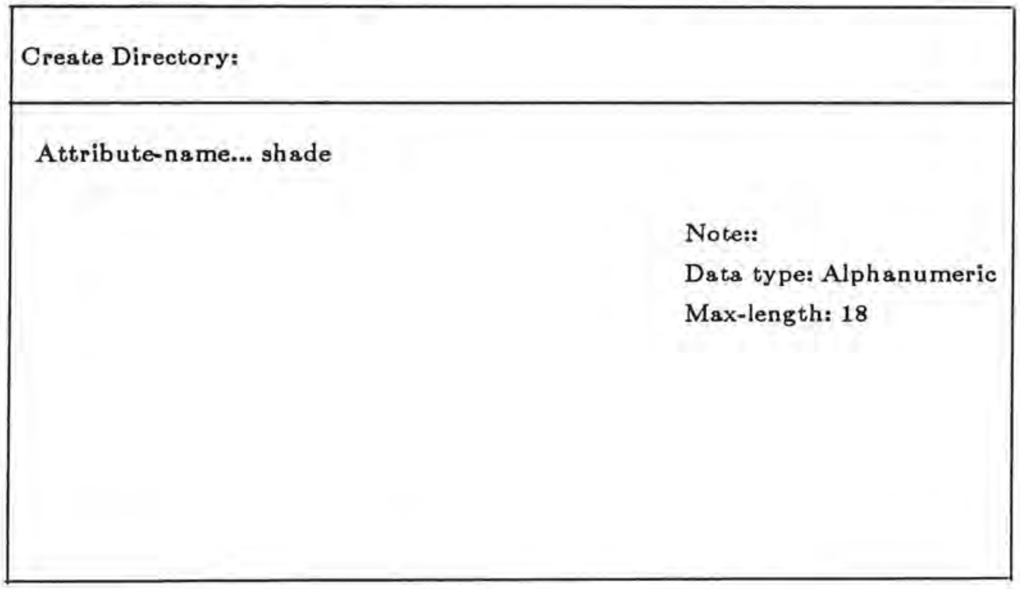


Create Directory:

Attribute-name... _

Note::
Data type: Alphanumeric
Max-length: 18

2.Type attribute-name.

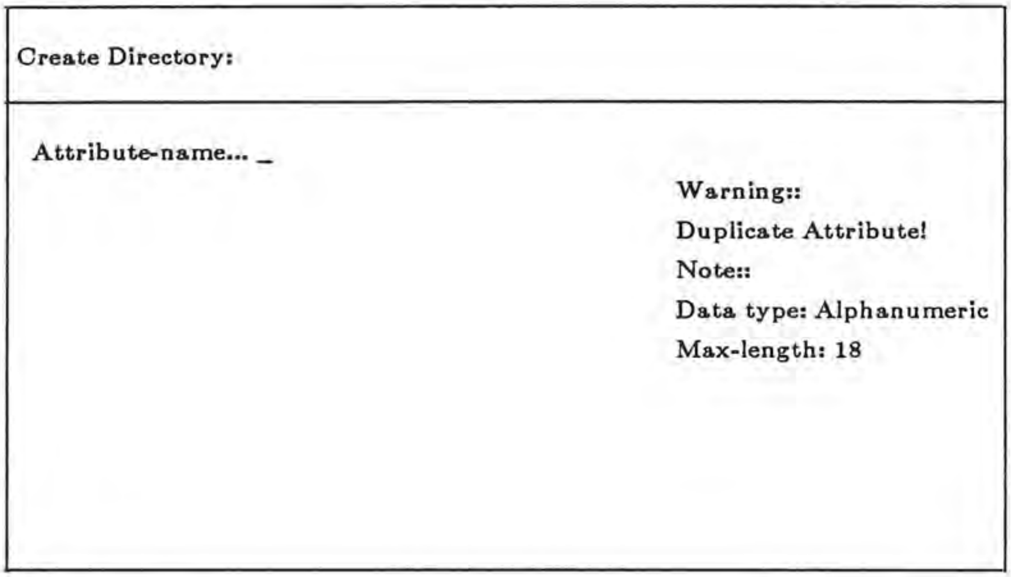


Create Directory:

Attribute-name... shade

Note::
Data type: Alphanumeric
Max-length: 18

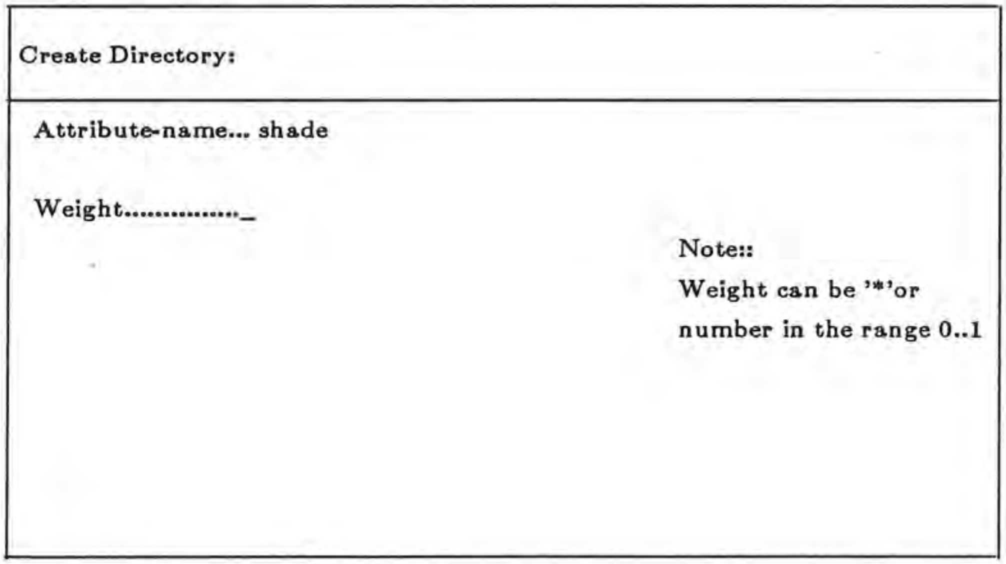
3. If the attribute is already in the directory then it is not entered again the message " duplicate attribute " is displayed. Now you can enter the attribute again.



```
Create Directory:

Attribute-name... _
                                          Warning::
                                          Duplicate Attribute!
                                          Note::
                                          Data type: Alphanumeric
                                          Max-length: 18
```

4. Type weight of attribute

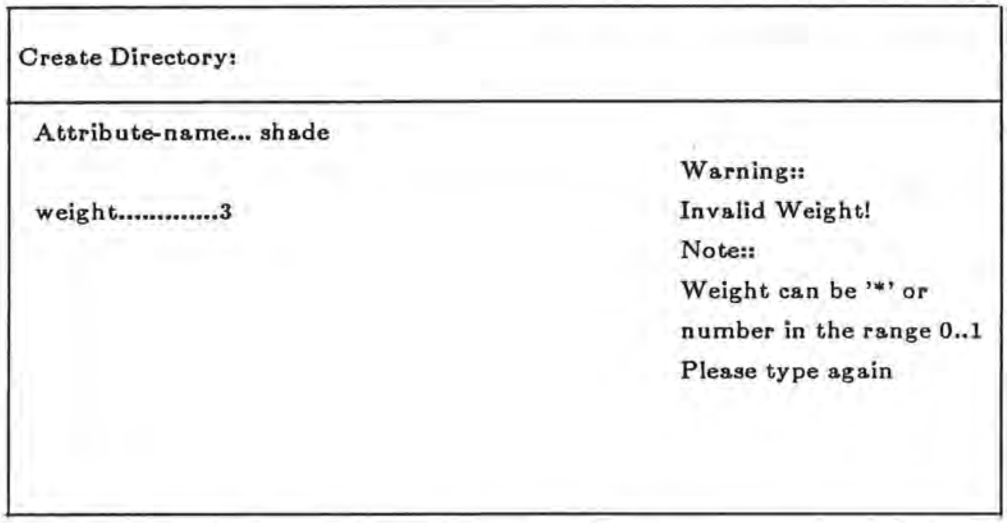


```
Create Directory:

Attribute-name... shade

Weight................_
                                          Note::
                                          Weight can be '*'or
                                          number in the range 0..1
```

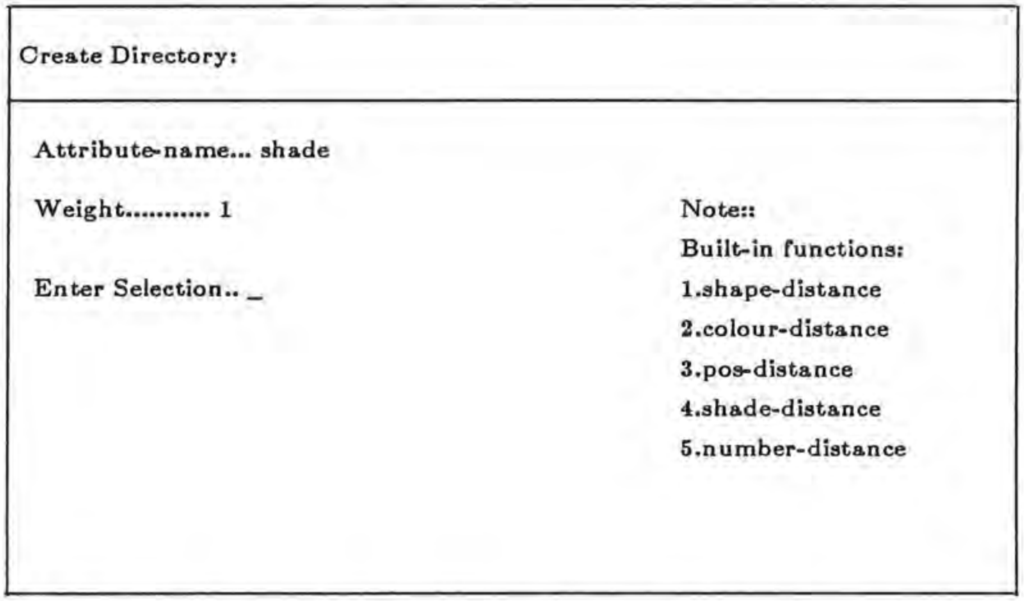
5.If the weight is not between 0 and 1 or not '' then invalid weight is displayed. Now enter weight again.*



```
Create Directory:

Attribute-name... shade
                                              Warning::
weight.............3                          Invalid Weight!
                                              Note::
                                              Weight can be '*' or
                                              number in the range 0..1
                                              Please type again
```

6. After you have entered correct weight then select function-number from the list of functions.



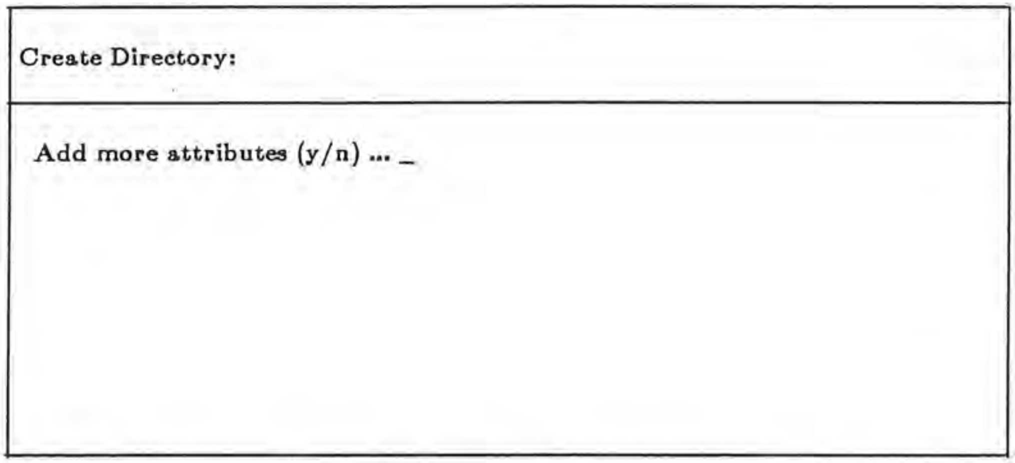
```
Create Directory:

Attribute-name... shade

Weight........... 1                           Note::
                                              Built-in functions:
Enter Selection.. _                           1.shape-distance
                                              2.colour-distance
                                              3.pos-distance
                                              4.shade-distance
                                              5.number-distance
```

7. If you selected a wrong number it won't be accepted and you will be asked to select again.

8. Select 'y' if you want to create more attributes, 'n' will take you back to menu.



```
Create Directory:

Add more attributes (y/n) ... _
```

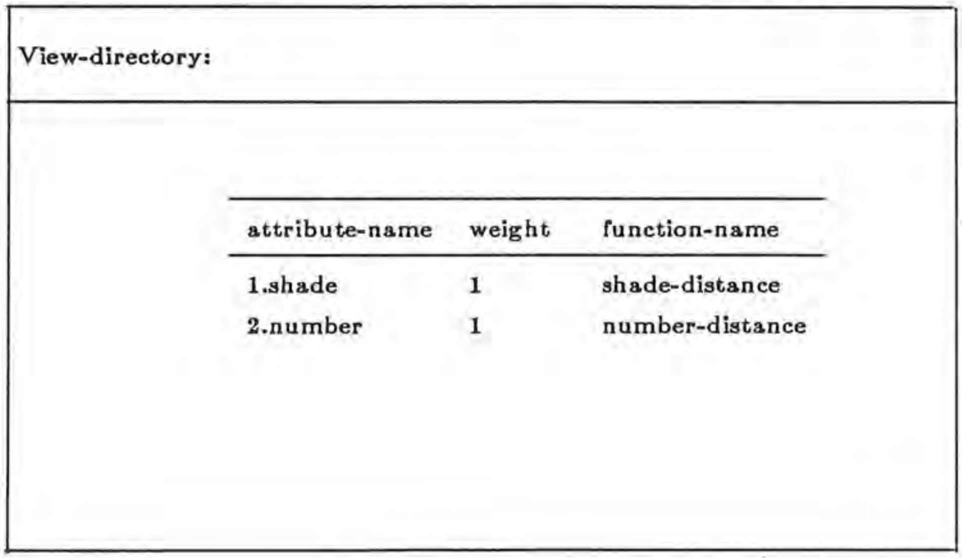
8. This finishes the routine of creating a directory entry.

2.2 To see contents of directory

If menu is being displayed then follow these steps else precede by

-> (start)

1. Select " View-directory".
You will get a display of directory-contents in tabular form.



```
View-directory:

            attribute-name   weight    function-name
            1.shade          1         shade-distance
            2.number         1         number-distance
```

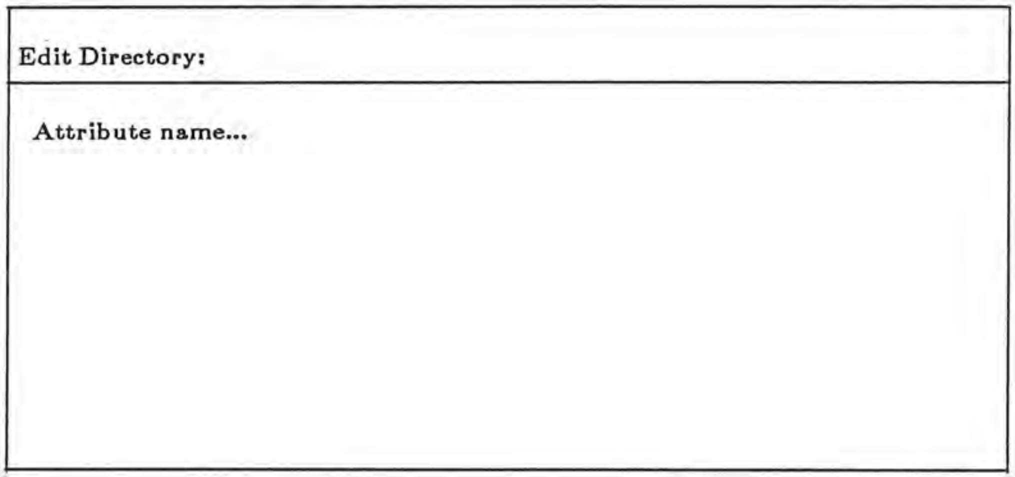
2. Press " return" to get back to menu.

2.3 To change contents of directory

If menu is being displayed then follow these steps else precede by

-> (start)

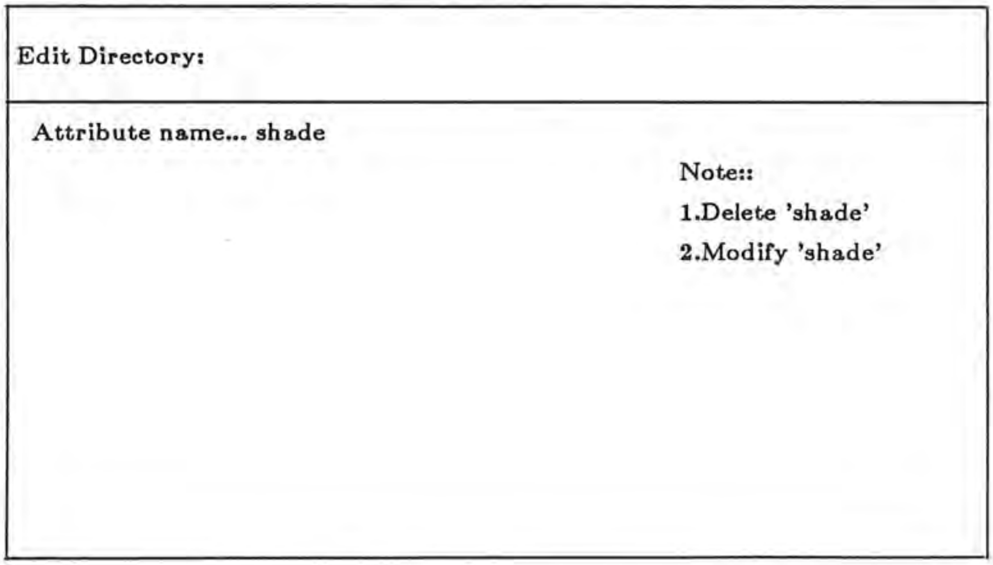
1.Select "Edit-directory".



Edit Directory:

Attribute name...

2.Type in the name of the attribute you want to edit. Then select '1' if you want to delete the attribute from directory or '2' if you want to change its weight or function-name.



Edit Directory:

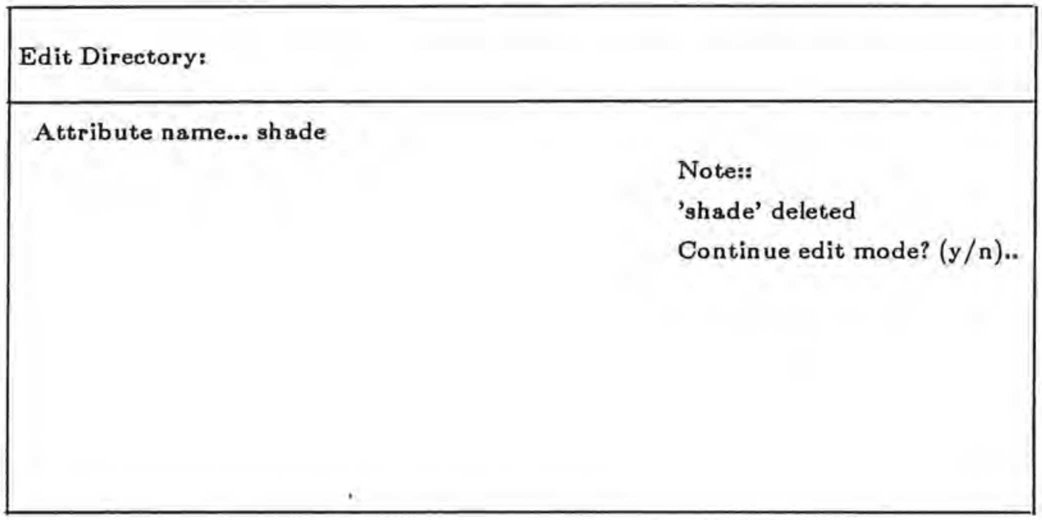
Attribute name... shade

Note::

1.Delete 'shade'

2.Modify 'shade'

3.If you selected '1', then attribute will be deleted and you get this view of screen.

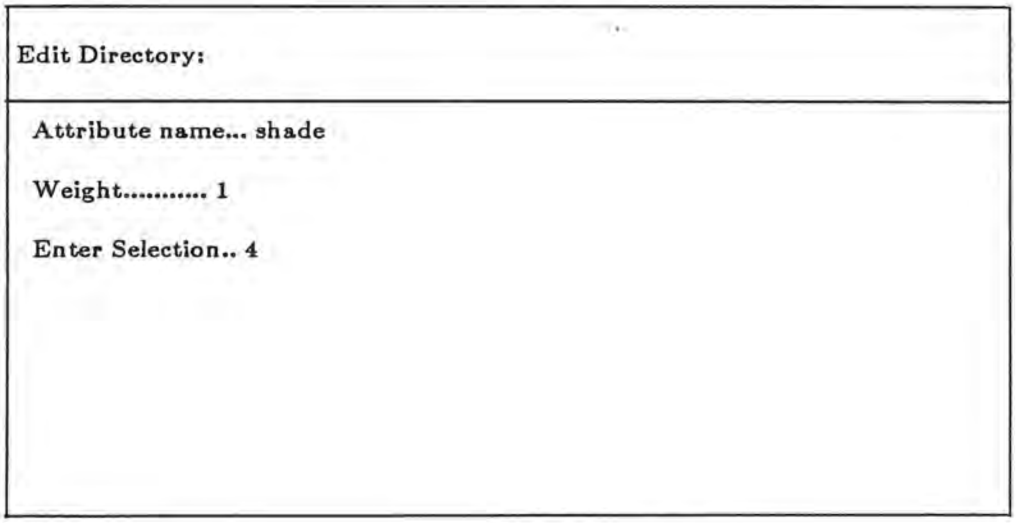


```
Edit Directory:

  Attribute name... shade
                                                      Note::
                                                      'shade' deleted
                                                      Continue edit mode? (y/n)..
```

4.'y' will get screen 1 'n' will get main menu.

5.If you selected '2' then weight and function-name is displayed, which can be modified.



```
Edit Directory:

  Attribute name... shade

  Weight........... 1

  Enter Selection.. 4
```

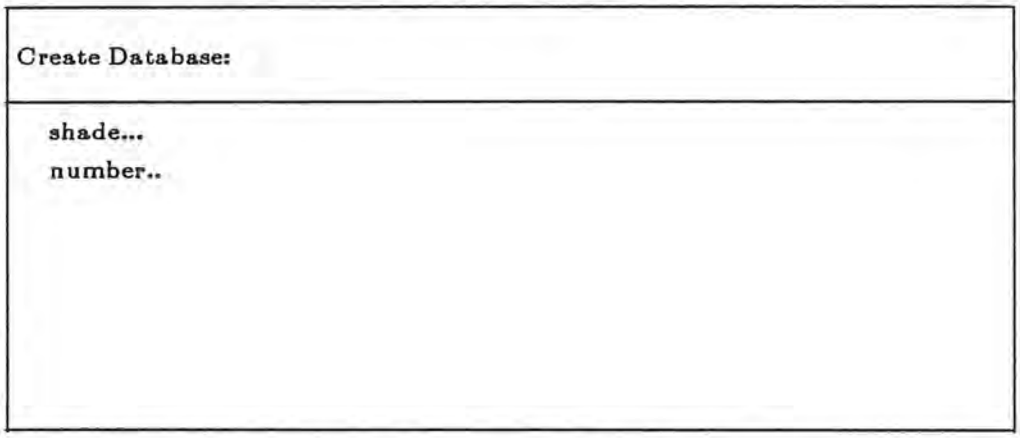
3.Setting Database:

3.1 To enter records in the database

If menu is being displayed, then proceed else precede by

-> (start)

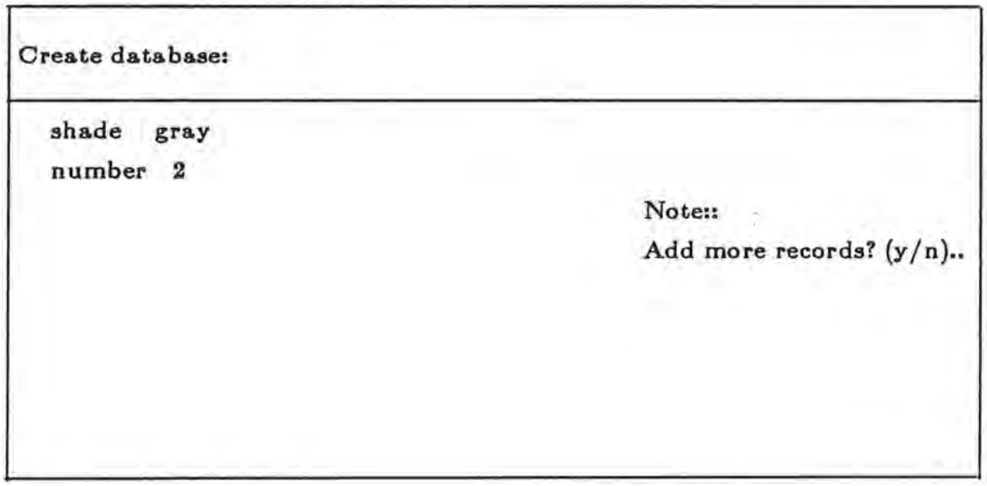
1.Select "Create Database". If directory has two attributes namely, shade and number then following screen will be displayed.



Create Database:

shade...
number..

2. Type in the values of attributes. Type 'y' if you want to add more records 'n' will get you menu.



Create database:

shade gray
number 2

Note::
Add more records? (y/n)..

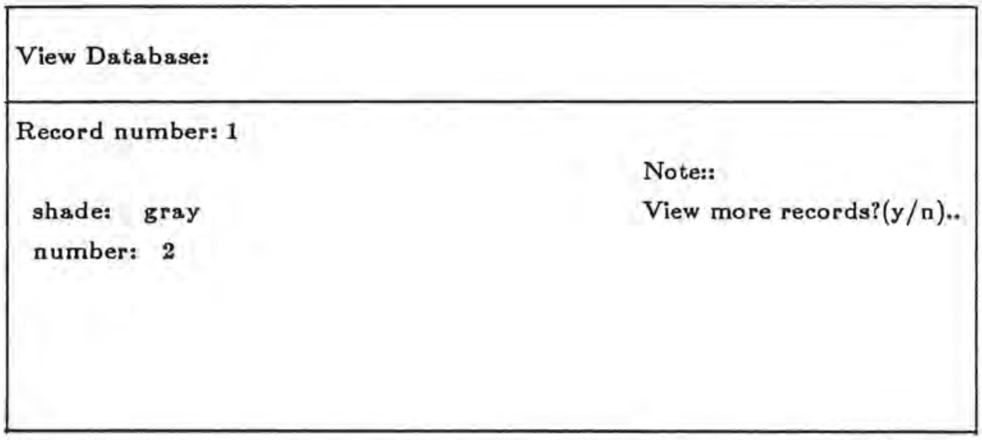
3. This finishes the routine of entering records in database.

3.2 To see records in database:

If menu is being displayed then do following, else precede by

-> (start)

1. Select " View Database"



```
View Database:
---------------------------------------------------------------
Record number: 1
                                              Note::
 shade:   gray                                View more records?(y/n)..
 number:  2
```

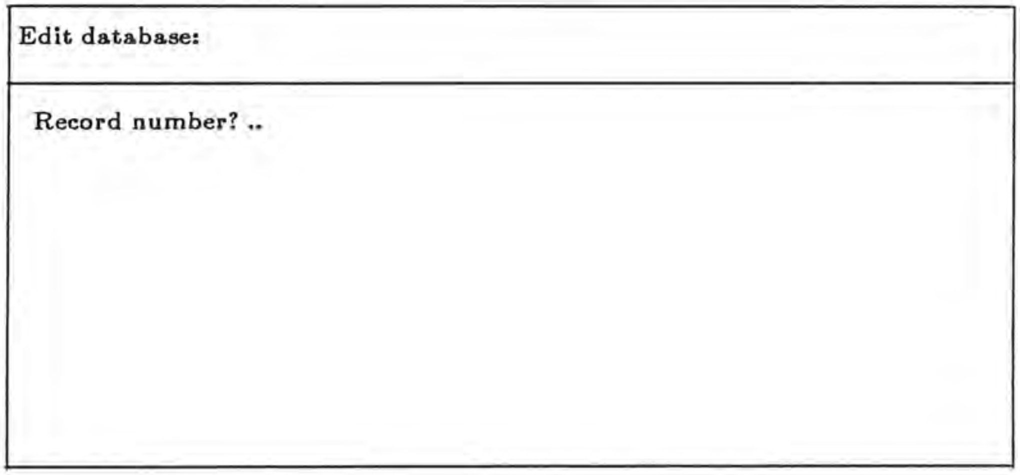
2. Type 'y' if you want to see more records, 'n' will display menu.

3.3 To edit records in database.

If menu is being displayed, then follow these steps else precede by

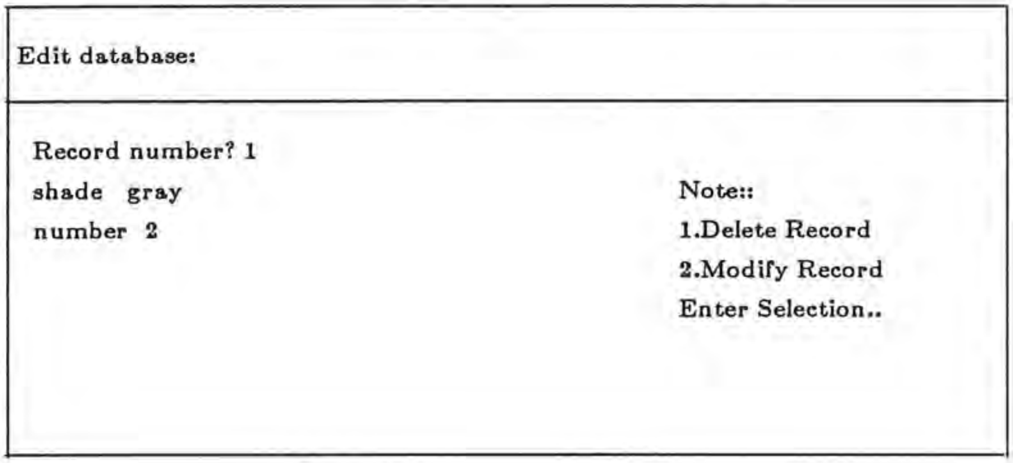
-> (start)

1. Select "Edit database"



```
Edit database:
---------------------------------------------------------------
 Record number? ..
```

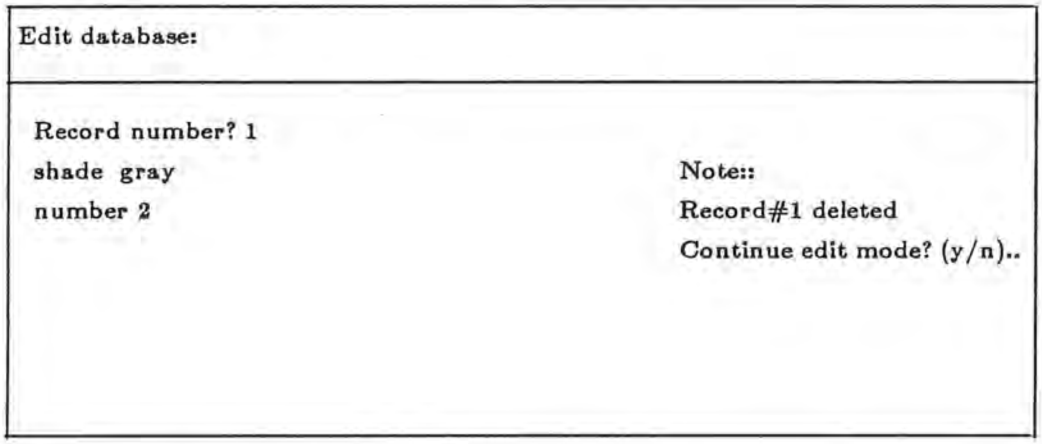
3.Type the record number to be edited. The record will be displayed with the message asking to delete or modify record.



```
Edit database:

Record number? 1
shade  gray                                         Note::
number  2                                           1.Delete Record
                                                    2.Modify Record
                                                    Enter Selection..
```

4.If 1 is selected, then record is deleted from database.



```
Edit database:

Record number? 1
shade  gray                                         Note::
number  2                                           Record#1 deleted
                                                    Continue edit mode? (y/n)..
```

5.Type 'y' to get more records for editing, 'n' will display menu.

6.If 2 is selected then record is displayed, and the values of attributes can be changed. After record is modified, type 'y' or 'n' as for delete.

Using System for pattern matching:

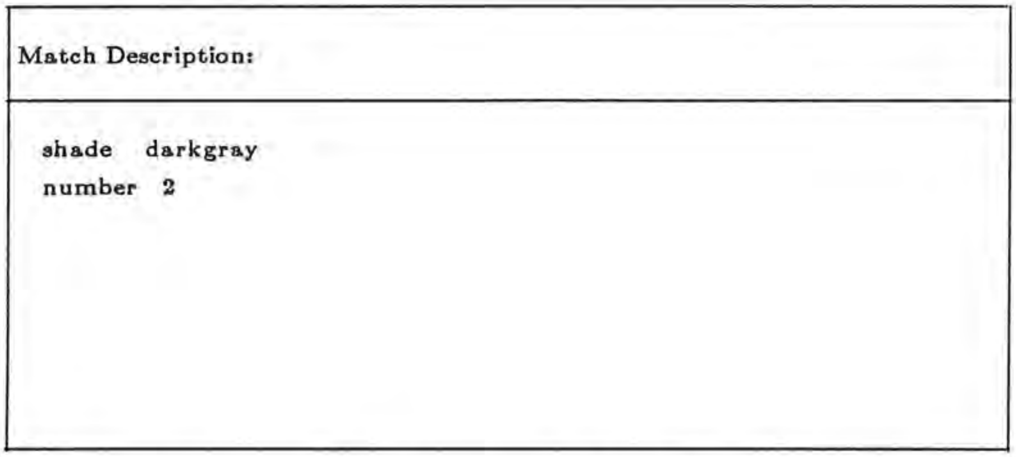
Once description-records are stored in database, these can be compared to observed description.

1. If menu is not being displayed, then

-> (start)
else go to step 2.

2. Select "match"

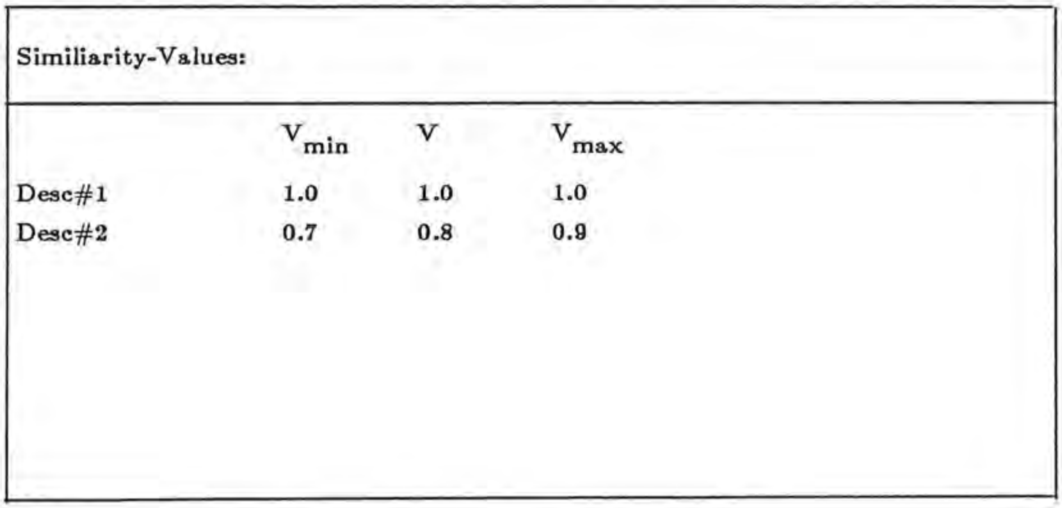
3. Enter the description to be matched.



```
Match Description:

  shade   darkgray
  number  2
```

3.The above description is compared to each stored description in database, and for each record you will get a similiarity-index which is a triple of V, Vmin and Vmax.The values can be sorted in ascending order on V value.



```
Similiarity-Values:

                 Vmin      V        Vmax
Desc#1           1.0       1.0      1.0
Desc#2           0.7       0.8      0.9
```

9. Demonstration

1. Use demo function instead of start 2. Create two entries in the directory:
   a) attribute-name: shade
      weight: anything between 0..1
      function-name: shade-distance
   b) attribute-name: number
      weight: anything between 0..1
      function-name: number-distance

3. Create 1-5 records in database.
   valid values for shade are: verylightgray, lightgray, gray, darkgray, black
   valid values for number is 1,2,3,4,5

4. Enter present description.

5. The pictures of descriptions are displayed alongwith the similarity measures and a bar graph is created to help comparing the similarity values.

```
;;; ==========================================================================
;;; START
;;; ==========================================================================
;;; (start)
;;;  This is the main routine. It turns on the Lisp Graphics Mode and sets the
;;;  graphic cursor. The text scrolling region is set to 5 lines and visible
;;;  graphic cursor at center of screen. Menu is displayed to select options.
;;;
;;;
(defun start()
   (init-graphics-mode :lines 5 :cursor-loc (make-Point x 300 y 200))
   (terpri)
   (message "Use any mouse button to select from menu")
   (menu-selection (menu-choose *menu*)))


;;; ==========================================================================
;;; *menu*
;;; ==========================================================================
;;;  *menu* is set to have the value returned by make-menu.
;;;  make-menu creates a menu from the item-list which contains the
;;;  following items each of which is associated with one slot of the menu.
;;;
;;;      create-directory, edit-directory, view-directory, create-database,
;;;      view-database, edit-database, match, exit.
;;;
;;;
(setq *menu*
      (make-menu '(create-directory view-directory edit-directory
                   create-database view-database edit-database match exit)))


;;; ==========================================================================
;;; MENU-SELECTION
;;; ==========================================================================
;;; (menu-selection <item>)
;;;   item:  list of menu items
;;;
;;;   Depending on the value returned from selecting a menu item,
;;;   control is passed by menu-selection to different routines.
;;;   If the selector is pressed and released outside of the menu,
;;;   menu is popped up again and waits for the selection.
;;;
;;;
(defun menu-selection (item)
   (cond
      ((eq item 'create-directory) (create-directory))
      ((eq item 'create-database) (create-database))
      ((eq item 'view-database) (view-database))
      ((eq item 'view-directory) (view-directory))
      ((eq item 'edit-directory) (edit-directory))
      ((eq item 'edit-database) (edit-database))
      ((eq item 'match) (match))
      ((eq item 'exit) (exit-graphics-mode))
      (t (menu-selection (menu-choose *menu*)))))


;;; ==========================================================================
;;;  CALL-MENU
;;; ========================================================================
;;;   (call-menu)
;;;    Call-menu prints a message to press "return" . If return key is pressed
;;;    it displays menu else it waits for key to be pressed.
;;;
;;;
(defun call-menu()
   (message  "Press RETURN to get menu  ")
   (cond
      ((eq (tyipeek) 13) (tyi) (start))
      (t (call-menu))))


;;; ========================================================================
;;; CREATE-DIRECTORY
;;; ========================================================================
;;;   (create-directory)
;;;    This function opens directory in append mode and sets up the screen.
;;;    It calls enter-directory to enter information in directory. When there
;;;    are no more attributes to be entered, it gets the menu to user.
;;;
(defun create-directory ()
   (heading "Create Directory: ")
   (append-file 'directory)
   (enter-directory 'y)
   (close my-outport)
   (start))


;;; ========================================================================
;;; ENTER-DIRECTORY
;;; ========================================================================
;;;   (enter-directory <ans>)
;;;    ans: 'y' or 'n'
;;;
;;;    This function creates an directory entry if answer is 'y' else returns
;;;    to main menu
;;;
;;;
(defun enter-directory (ans)
   (clear-text-region)
   (cond
      ((eq ans 'n))
      (t (write-in-dir)
         (enter-directory (get-answer)))))

;;;========================================================================
;;; WRITE-IN-DIR
;;;========================================================================
;;;   (write-in-dir)
;;;     This function gets attribute-name from the user. If it is not nil ( nil
;;;     is returned if the attribute is duplicate) then it gets weight
;;;     and name of the similarity function from the user and append this entry
;;;     in directory.
;;;
;;;
(defun write-in-dir ()
   (let ((a (get-attribute)))
      (cond
         ((eq a nil))
         (t (print (list a (get-weight) (get-function-name 0)) my-outport)))))

;;; =========================================================================
;;; GET-ANSWER
;;; =========================================================================
;;;  (get-answer)
;;;
;;; This function asks user if more attributes are to be added to directory.
;;; If answer is 'y' or 'n' then it returns the answer, else it prompts user
;;; to type in answer again.
;;;
(defun get-answer ()
   (CUP 20)
   (CUF 4) (format t "Add more attributes? (y/n)..")
   (let ((reply (my-ratom)))
      (cond
         ((or (eq reply 'y) (eq reply 'n)) (clear-text-region) reply)
         (t (CUF 45) (format t "Warning~%")
            (CUF 45) (format t "Wrong selection!")
            (CUP 2 32) (format t "    ")
            (get-answer)))))


;;;==========================================================================
;;; GET-ATTRIBUTE
;;; ==========================================================================
;;;  (get-attribute)
;;;
;;; This function gets attribute-name from the user. It displays a note to the
;;; user that attribute name can be any alphanumeric characters and the max.
;;; length is 18 characters. If it is valid name and if it is already not in
;;; directory then it is accepted.
;;;
;;;
(defun get-attribute ()
   (CUP 4)
   (CUF 45) (format t "Note::                         ~%")
   (CUF 45) (format t "Data type: Alphanumeric~%")
   (CUF 45) (format t "Max-length: 18~%")
   (CUP 2)  (format t "  Attribute-name...  ")
   (cond
      ((read-file 'directory) (if-in-file (check-length (my-ratom))))
      (t(check-length (my-ratom)))))

;;;=========================================================================
;;; IF-IN-FILE
;;; =========================================================================
;;;  (if-in-file <attribute>)
;;;   attribute: name of the attribute
;;;
;;;   This function calls duplicate-attribute if attribute is found in directory
;;;   else returns attribute.
;;;
;;;
(defun if-in-file (attribute)
   (cond
      ((eq (tyipeek my-inport) -1) (close my-inport) attribute)
      ((eq attribute (car (read my-inport))) (duplicate-attribute))
      (t (if-in-file attribute))))
```

```
;;; ==========================================================================
;;; DUPLICATE-ATTRIBUTE
;;; ==========================================================================
;;;  (duplicate-attribute)
;;;
;;;  This function displays warning about the attribute name found in the
;;;  directory and if user wants to add more attribute, calls get-attribute
;;;  else displays menu for next selection.
;;;
;;;
(defun duplicate-attribute ()
   (close my-inport)
   (CUP 2)
   (CUF 45) (format t "Warning::~%")
   (CUF 45) (format t "Duplicate Attribute!~%")
   (CUF 45) (format t "                         ~%")
   (CUF 45) (format t "                         ~%")
   (CUF 45) (format t "                         ~%")
   (CUF 45) (format t "                         ~%"))


;;; ==========================================================================
;;; CHECK-LENGTH
;;; ==========================================================================
;;; (check-length <attribute>)
;;;   attribute:  name of the attribute
;;;
;;;   If length of attribute-name is greater than 18 characters, this function
;;;   gives warning and asks user to type in the name again.
;;;
;;;
(defun check-length (attribute)
   (cond
      ((greaterp (length (explode attribute)) 18) (display-length-warning))
      (t attribute)))


;;; ==========================================================================
;;; DISPLAY-LENGTH-WARNING
;;; ==========================================================================
;;;  (display-length-warning)
;;;
;;;  This function displays warning that the name is more than 18 characters.
;;;
(defun display-length-warning ()
   (CUP 2)
   (CUF 45) (format t "Warning::~%")
   (CUF 45) (format t "Invalid length!")
   (CUP 8)
   (CUF 45) (format t "Please type again")
   (CUP 2)
   (CUF 20) (format t "                         ")
   (CUP 2)
   (CUF 20) (check-length (my-ratom)))


;;; ==========================================================================
;;; GET-WEIGHT
;;; ============================================================================
;;; (get-weight)
;;;
;;; This function prompts user to assign weight to attribute-name. It displays
;;; a note to the user that weight can be either '*' or a number in range of
;;;  0..1. It calls validate-weight to check if weight is valid.
;;;
;;;
(defun get-weight ()
   (CUP 2 45) (format t "                           ~%")
   (CUP 3 45)
   (format t "                                 ")
   (CUP 4)
   (CUF 45) (format t "Note::~%")
   (CUF 45) (format t "Weight can be '*'      ~%")
   (CUF 45) (format t "or number in range of 0..1~%")
   (CUP 4)  (format t "  Weight........... ")
   (validate-weight (my-ratom)))


;;; ============================================================================
;;; VALIDATE-WEIGHT
;;; ============================================================================
;;; (validate-weight <weight>)
;;;  weight: weight of the attribute
;;;
;;;  This function checks if it is valid weight, it returns weight itself
;;;  else prompts user to try again
;;;
;;;
(defun validate-weight (weight)
   (cond
      ((eq weight '*) weight)
      ((and (numberp weight) (not (or (greaterp weight 1.0)(minusp weight))))
        weight)
      (t (CUP 2)
         (CUF 45) (format t "Warning~%")
         (CUF 45) (format t "Invalid weight!")
         (CUP 7)
         (CUF 45) (format t "Please type again")
         (CUP 4)
         (CUF 20) (format t "             ")
         (CUP 4 21)
         (validate-weight (my-ratom)))))


;;; ============================================================================
;;;  GET-FUNCTION-NAME
;;; ============================================================================
;;;  (get-function-name <num>)
;;;  num : index of function name, initial value is 0 (dummy) if it is called
;;;  while entering entry in directory. It is the number of previous value
;;;  while in editing mode.
;;;
;;;  This function displays the function-names with their numbers. User
;;;  selects a number which is returned in num.
;;;
;;;
(defun get-function-name (num)
   (CUP 2)
   (CUF 45)  (format t "            ~%")
   (CUF 45)  (format t "            ~%")
   (terpri)
   (CUF 45)  (format t "Built-in functions:~%")
   (CUF 45)  (format t "                  ~%")
   (display-function-list *function-list* 7)
   (CUP 6)
   (format t "  Enter selection.. ")
   (cond
     ((eq (tyipeek) 13) (tyi) num)
     (t (check-selection (my-ratom)))))

;;; ==================================================
;;;  DISPLAY-FUNCTION-LIST
;;; ==================================================
;;; (display-function-list <f-list> <cursor>)
;;;  f-list: global function list
;;;  cursor: row number where starting to display functions
;;;
;;; This function displays names of the built-in functions on the screen.
;;;
;;;
(defun display-function-list (f-list cursor)
  (cond
    ((null f-list) t)
    (t (CUP cursor 45)
      (format t "~a. ~a   ~%" (caar f-list) (cadar f-list))
      (display-function-list (cdr f-list) (add1 cursor)))))

;;; ==================================================
;;; CHECK-SELECTION
;;; ==================================================
;;; (check-selection <num>)
;;;  num: function number
;;;
;;; The function-number selected is validated . If it is not, then user is
;;; asked to type it again.
;;;
;;;
(defun check-selection (num)
  (cond
    ((not (numberp num)) (select-function-again))
    ((or (greaterp num 15) (lessp num 1)) (select-function-again))
    (t (return-name num *function-list*))))


;;; ==================================================
;;; SELECT-FUNCTION-AGAIN
;;; ==================================================
;;;  (select-function-again)
;;;
;;;  This function gives a warning that function number selected is not valid
;;;  and prompts to type again.
;;;
(defun select-function-again ()
  (CUP 2)
  (CUF 45)  (format t "Warning::~%")
  (CUF 45)  (format t "Wrong selection!")
  (CUP 6)
  (CUF 20)  (format t "    ")
```

```
    (CUP 6 21)
    (check-selection (my-ratom)))

;;; ==========================================================================
;;; RETURN-NAME
;;; ==========================================================================
;;;  (return-name <num> <lst>)
;;;   num: function-number
;;;   lst: global function list
;;;
;;; This function returns name of function for the number choosen by user.
;;;
;;;
(defun return-name (num lst)
    (cond
       ((eq num (caar lst)) (nth 1 (car lst)))
       (t (return-name num (cdr lst)))))


;;; ==========================================================================
;;;  VIEW-DIRECTORY
;;; ==========================================================================
;;;  (view-directory)
;;;   View-directory helps user to view the contents of directory.
;;;
;;;
(defun view-directory ()
    (print-heading)
    (cond
       ((eq (read-file 'directory) nil)
        (message "No attributes in directory")
        (print-return-message))
       (t (read-dir 80 120 1)
         (close my-inport)
         (print-return-message))))


;;;==========================================================================
;;;PRINT-HEADING
;;;==========================================================================
;;;  (print-heading)
;;;   Print-heading prepares screen for displaying the contents of directory.
;;;
;;;
(defun print-heading ()
    (init-graphics-mode :lines 5)
    (draw-line (make-Point x 0 y 10) (make-Point x 625 y 10))
    (paint-string 10 20 "View-directory: ")
    (draw-line (make-Point x 0 y 35) (make-Point x 625 y 35))
    (draw-line (make-Point x 80 y 70) (make-Point x 500 y 70))
    (paint-string 80 90 "attribute-name")
    (paint-string 230 90 "weight")
    (paint-string 340 90 "function-name")
    (draw-line (make-Point x 80 y 120) (make-Point x 500 y 120)))


;;; ==========================================================================
;;;  READ-DIR
;;; ==========================================================================
;;; (read-dir <x> <y> <count>)
;;;  x : row number
;;;  y : column number
;;;  count: entry number
;;;
;;;  This function gets records from the directory to be displayed on screen.
;;;
;;;
(defun read-dir ( x y count)
   (cond
      ((eq (tyipeek my-inport) -1))
      ((greaterp y 370) (print-message)
       (print-heading)
       (write-line (read my-inport) x 120 count)
       (read-dir x  135 (add1 count)))
      (t (write-line (read my-inport) x y count)
         (read-dir x (plus y 15) (add1 count)))))

;;; ================================================================================
;;; WRITE-LINE
;;; ================================================================================
;;;  (write-line <lst> <x> <y> <count>)
;;;   lst: directory entry
;;;   x : row number
;;;   y : column number
;;;   count: number of entry
;;;
;;; This function displays lst on screen.
;;;
;;;
(defun write-line ( lst x y count)
   (cond
      ((eq (car lst) nil))
      (t (paint-string (diff x 22) (plus y 10) (my-concat count))
         (paint-string x (plus y 10) (car lst))
         (paint-string (plus x 150) (plus y 10) (cadr lst))
         (paint-string (plus x 260) (plus y 10) (caddr lst)))))


;;;================================================================================
;;;PRINT-MESSAGE
;;;================================================================================
;;;  (print-message)
;;;   Print-message prints message to press return to get the next screen
;;;   display.
;;;
(defun print-message ()
   (terpri)
   (message "Hit return to get next screen")
   (cond
      ((eq (tyi) 13))
      (t (print-message))))


;;; ===============================================================================
;;;   PRINT-RETURN-MESSAGE
;;; ===============================================================================
;;;  (print-return-message)
;;;   This function prints message on screen to return from view-mode
;;;
;;;
;;;
```

```
(defun print-return-message ()
   (terpri)
   (message "Hit RETURN to get menu-selection")
   (cond
      ((eq (tyi) 13) (start))
      (t (print-return-message))))


;;; ==========================================================================
;;;  EDIT-DIRECTORY
;;; ==========================================================================
;;; (edit-directory)
;;;  This function lets user delete or modify directory entries. It gets the
;;;  the attribute name from the user and displays the  directory entry of
;;;  that attribute from the directory. If the entry is not in the directory
;;;  it prints error message.
;;;
;;;
(defun edit-directory ()
   (heading "Edit Directory: ")
   (terpri)
   (format t "     Attribute-name.... ")
   (let ((name (my-ratom)))
      (cond
         ((eq (read-file 'directory) nil)
          (display-not-found-note name))
         (t (search-dir name)))))


;;; ==========================================================================
;;; DISPLAY-NOT-FOUND-NOTE
;;; ==========================================================================
;;; (display-not-found-note <name>)
;;; name: attribute-name
;;;
;;;  This function displays warning that given attribute has no entry in the
;;;  directory.
;;;
(defun display-not-found-note (name)
   (CUP 2)
   (CUF 45) (format t "Warning::~%")
   (CUF 45) (format t "'~a' not found in directory!" name))


;;; ==========================================================================
;;; DISPLAY-EDIT-MORE-NOTE
;;; ==========================================================================
;;; (display-edit-more-note)
;;;  This function asks user if more directory entries are to be edited.
;;;
(defun display-edit-more-note ()
   (CUP 4)
   (CUF 45) (format t "Note:: ~%")
   (terpri)
   (CUF 45) (format t "Continue edit mode? (y/n)..."))


;;; ==========================================================================
;;; GET-EDIT-ANSWER
;;; ==========================================================================
;;;  (get-edit-answer <reply>)
;;;   reply: 'y' or 'n'
;;;
;;;   This function gets answer for the above note. If answer is y, then next
;;;   attribute entry will be selected else menu is displayed.
;;;
(defun get-edit-answer (reply)
   (cond
      ((eq reply 'y) (clear-text-region) (edit-directory))
      ((eq reply 'n) (start))
      (t (CUP 3)
         (CUF 45) (format t "Wrong Selection!                   ")
         (CUP 6 73) (format t "   ")
         (CUP 6 73)  (get-edit-answer(my-ratom)))))


;;; =========================================================================
;;;  DISPLAY-DEL-MOD-NOTE
;;; =========================================================================
;;; (display-del-mod-note <lst>)
;;; lst: directory entry
;;;
;;;  This function asks user if the given entry has to be deleted or modified.
;;;
(defun display-del-mod-note( lst)
   (CUP 4)
   (CUF 45) (format t "Note:: ~%")
   (CUF 45) (format t "1. Delete ~a ~%" (car lst))
   (CUF 45) (format t "2. Modify ~a ~%" (car lst))
   (terpri)
   (CUF 45) (format t "Enter selection... ")
   (get-del-mod-answer (my-ratom) lst))


;;; =========================================================================
;;; GET-DEL-MOD-ANSWER
;;; =========================================================================
;;; (get-del-mod-answer <reply> <lst>)
;;; reply: 'y' or 'n'
;;; lst: directory entry
;;;
;;;  This function checks if reply is valid and then passes control to
;;;  respective functions.
;;;
;;;
(defun get-del-mod-answer (reply lst)
   (cond
      ((eq reply 1) (delete-attribute lst))
      ((eq reply 2) (modify-attribute lst))
      (t (CUP 2)
         (CUF 45) (format t "Warning::~%")
         (CUF 45) (format t "Wrong Selection!")
         (CUP 8 65) (format t "   ")
         (CUP 8 65) (get-del-mod-answer (my-ratom) lst))))


;;; =========================================================================
;;; SEARCH-DIR
;;; =========================================================================
;;; (search-dir <name>)
;;; name: attribute-name
;;;
;;; This function searches directory for the entry belonging to given
;;; attribute-name.
;;;
;;;
(defun search-dir ( name)
   (cond
      ((eq (tyipeek my-inport) -1)  (display-not-found-note name)
                                    (close my-inport)
                                    (display-edit-more-note)
                                    (get-edit-answer (my-ratom)))
      (t (search-dir-help (read my-inport) name))))


;;; ======================================================================
;;;  SEARCH-DIR-HELP
;;; ======================================================================
;;; (search-dir-help <lst> <name>)
;;; lst: list of directory-entries
;;; name: attribute-name
;;;
;;;  This function searches entry in the list of directory entries.
;;;
;;;
(defun search-dir-help (lst name)
   (cond
      ((eq name (car lst))  (close my-inport) (display-del-mod-note lst))
      (t (search-dir name))))


;;; ======================================================================
;;; DELETE-ATTRIBUTE
;;; ======================================================================
;;; (delete-attribute <lst>)
;;;  lst: directory-entry
;;;
;;;  The entry is deleted from the directory and from each record in the
;;;  database, all attribute-value pairs for this entry are deleted.
;;;
(defun delete-attribute (lst)
   (delete-from-dir lst)
   (delete-from-database lst)
   (CUP 5)
   (CUF 45) (format t "'~a' deleted!~%" (car lst))
   (CUF 45) (format t "                 ~%")
   (CUP 8)
   (CUF 45) (format t "                         ~%")
   (display-edit-more-note)
   (get-edit-answer (my-ratom)))


;;; ======================================================================
;;; DELETE-FROM-DIR
;;; ======================================================================
;;; (delete-from-dir <lst>)
;;; lst: directory-entry
;;;
;;;  This function deletes directory entry from the directory. It gets all
;;;  entries from directory and then writes back without given entry.
```

```
;;;
(defun delete-from-dir (lst)
   (read-file 'directory)
   (let ((dir-list (make-dir-list nil)))
      (close my-inport)
      (setq my-outport (outfile 'directory))
      (write-back-dir dir-list lst)
   (close my-outport)))


;;; ======================================================================
;;; MAKE-DIR-LIST
;;; ======================================================================
;;; (make-dir-list <lst>)
;;; lst: list of directory entries
;;;
;;;  This function reads directory and returns a list of all entries.
;;;
(defun make-dir-list (lst)
   (cond
      ((eq (tyipeek my-inport) -1) lst)
      ((null lst) (make-dir-list (cons (read my-inport) lst)))
      (t (make-dir-list (append lst (list (read my-inport)))))))


;;; ======================================================================
;;; WRITE-BACK-DIR
;;; ======================================================================
;;; (write-back-dir <dir-list> <lst>)
;;; dir-list: list of directory entries
;;; lst: entry to be deleted
;;;
;;;  This function writes the directory entries in directory except the one
;;;  whose attribute-name is same as the given entry.
;;;
(defun write-back-dir (dir-list lst)
    (cond
       ((null dir-list) t)
       ((eq (caar dir-list) (car lst)) (write-back-dir (cdr dir-list) lst))
       (t (print (car dir-list) my-outport)
          (write-back-dir (cdr dir-list) lst))))


;;;=======================================================================
;;;  DELETE-FROM-DATABASE
;;; ======================================================================
;;; (delete-from-database <lst>)
;;; lst: directory entry
;;;
;;;  This function deletes from database records the attribute-value pairs
;;;  whose attribute is same as attribute in given lst.
;;;
(defun delete-from-database (lst)
   (cond
      ((read-file 'dbase) (update-database (read my-inport) lst nil)
                           (close my-inport))
      (t 't)))


;;; ======================================================================
;;; UPDATE-DATABASE
;;; ===========================================================================
;    (update-database <dlist> <lst> <new-lst>)
;;, dlist: database records
;;; lst: directory entry
;;; new-lst: new database records
;;;
;;;  This function takes the old database records and makes a  new list with
;;;  deleted attribute value pairs.
;;;
;;;
(defun update-database ( dlist lst new-list)
    (cond
       ((null dlist) (update-file new-list))
       (t (update-database (cdr dlist) lst
                           (my-cons new-list (cons (caar dlist)
                           (my-delete lst (cdar dlist) nil)))))))


;;; ===========================================================================
;;; UPDATE-FILE
;;; ===========================================================================
;;; (update-file <new-lst>)
;;;  new-lst: updated database records
;;;
;;;  This function writes the new records in database file.
;;;
;;;
(defun update-file ( new-list)
   (write-file  'dbase)
   (print new-list my-outport)
   (close my-outport))


;;; ===========================================================================
;;;  MODIFY-ATTRIBUTE
;;; ===========================================================================
;;; (modify-attribute <lst>)
;;;  lst: directory entry
;;;
;;;  This function displays selected attribute entry and modifies the entry.
;;;
(defun modify-attribute (lst)
   (clear-text-region)
   (CUP 2) (format t "  Attribute-name  : ~a~%" (car lst))
   (CUP 4) (format t "  Weight......... : ~a~%" (cadr lst))
   (CUP 6) (format t "  Function-name.. : ~a~%"
                   (get-function-number (caddr lst) *function-list*))
   (read-file 'directory)
   (let ((dir-list (make-dir-list nil)))
      (close my-inport)
      (write-file 'directory)
      (write-modified-info dir-list lst)
      (close my-outport)
      (clear-text-region)
      (terpri)
      (format t "     Attribute-name.... ~a" (car lst))
      (CUP 5)
      (CUF 45) (format t "'~a' modified!" (car lst))
      (display-edit-more-note)
      (get-edit-answer (my-ratom))))

;;;===========================================================================
;;; WRITE-MODIFIED-INFO
;;;===========================================================================
;;; (write-modified-info <dir-list> <lst>)
;;; dir-list:  list of all entries in directory
;;; lst: entry to be modified
;;;
;;;  This function writes the directory entries in directory with the given
;;;  entry modified.
;;;
;;;
(defun write-modified-info (dir-list lst)
   (cond
      ((null dir-list))
      ((eq (caar dir-list) (car lst)) (write-new-info lst)
       (write-modified-info  (cdr dir-list) lst))
      (t (print (car dir-list) my-outport)
         (write-modified-info (cdr dir-list) lst))))


;;; ===========================================================================
;;;  WRITE-NEW-INFO
;;; ===========================================================================
;;; (write-new-info <lst>)
;;;  lst: directory entry to be modified
;;;
;;;  This function gets the changed information about weight and function name
;;;  from the user and makes a new entry
;;;
(defun write-new-info (lst)
   (CUP 4 21)
   (cond
      ((eq (tyipeek) 13) (tyi)
       (print (list (car lst) (cadr lst) (get-function-name (caddr lst)))
              my-outport))
      (t (print (list (car lst) (get-weight) (get-function-name (caddr lst)))
                my-outport))))

;;; ===========================================================================
;;;  GET-FUNCTION-NUMBER
;;; ===========================================================================
;;; (get-function-number <name> <flist>)
;;; name: function-name
;;; flist: function-list
;;;
;;;  For the given function-name, this function returns corresponding function
;;;  number.
;;;
(defun get-function-number (name flist)
   (cond
      ((eq name (cadar flist)) (caar flist))
      (t (get-function-number name (cdr flist)))))


;;; ===========================================================================
;;;  CREATE-DATABASE
;;; ===========================================================================
;;;
```

```
;;; (create-database)
;;;  This function lets user enter object descriptions in the database.
;;;
 efun create-database ()
    (heading "Create-Database:")
    (setq my-outport (outfile 'dbase))
    (setq my-inport (infile 'directory))
    (print (get-dbase-info (make-attribute-list nil) nil 'y 1) my-outport)
    (close my-inport)
    (close my-outport)
    (start))


;;; ========================================================================
;;;  MAKE-ATTRIBUTE-LIST
;;; ========================================================================
;;; (make-attribute-list <lst>)
;;; lst: list of attributes in directory
;;;
;;;  This function makes a list of attributes present in directory.
;;;
;;;
(defun make-attribute-list (lst)
   (cond
      ((eq (tyipeek my-inport) -1) lst)
      (t (make-attribute-list (my-cons lst (car (read my-inport)))))))


;;; ========================================================================
;;;  GET-DBASE-INFO
;;; ========================================================================
;;; (get-dbase-info <a-list> <lst> <reply> <count>)
;;;  a-list: attribute-list
;;;  lst: description lists
;;;  reply : 'y' or 'n'
;;;  count: record number
;;;
;;;  For each description to be entered, this function gets the values for
;;;  the attributes which are already known in directory and makes attribute
;;;  value lists and enters in database.
;;;
;;;
(defun get-dbase-info ( a-list lst reply count)
   (clear-text-region)
     (cond
        ((eq reply 'n) lst)
        (t (get-dbase-info a-list
              (my-cons lst (get-record-info a-list a-list count 2 nil))
                         (display-more-dbase-note)
                         (add1 count)))))


;;;========================================================================
;;; GET-RECORD-INFO
;;;========================================================================
;;; (get-record-info <lst1> <lst2> <count> <pos> <temp>)
;;; lst1: attribute-list
;;; lst2: record-description
;;; count: record number
;;; pos: cursor position
;;; temp: temporary list
;;;
;;;  This function returns one description with record number appended.
;;;
(defun get-record-info (lst1 lst2 count pos temp)
     (cond
        ((null lst1) (append (list count) (get-values lst2 temp 2 0)))
        ((greaterp pos 15)
         (get-record-info lst1 lst1 count 2 (get-values lst2 temp 2 0)))
        (t (CUP pos 4) (format t "~a..~%" (car lst1))
           (get-record-info (cdr lst1) lst2 count (add1 pos) temp))))


;;; ===========================================================================
;;; GET-VALUES
;;; ===========================================================================
;;; (get-values <a-list> <lst> <pos> <num>)
;;;  a-list: attribute-list
;;;  lst: list of attribute-value pairs
;;;  pos: cursor position where value is to be entered.
;;;  num : record number
;;;
;;;  This function gets values for attributes and makes a list.
;;;
;;;
(defun get-values (a-list lst pos num)
   (cond
      ((greaterp num 13) (clear-text-region) lst)
      ((null a-list) lst)
      (t(CUP pos 22)
       (get-values (cdr a-list) (get-pairs a-list lst)
                     (add1 pos)(add1 num)))))

;;; ===========================================================================
;;; GET-PAIRS
;;; ===========================================================================
;;; (get-pairs <a-list> <lst>)
;;;  a-list: list of attributes
;;;  lst: list of attribute-value pairs
;;;
;;;  This function returns attribute-value pairs.
;;;
;;;
(defun get-pairs (a-list lst)
   (cond
      ((eq (tyipeek) 13) (tyi) (my-cons lst (list (car a-list) nil)))
      (t (my-cons lst(list (car a-list) (my-ratom))))))

;;; ===========================================================================
;;;  DISPLAY-MORE-DBASE-NOTE
;;; ===========================================================================
;;; (display-more-database-note)
;;;  This function asks if more records are to be added to database.
;;;
;;;
(defun display-more-dbase-note ()
   (CUP 4 45)
   (format t " Note::~%")
   (CUF 45) (format t "Add more records? (y/n)..")
   (let ((reply (my-ratom))
      (cond
         ((or (eq reply 'n) (eq reply 'y))  reply)
         (t (help-dbase-answer)))))


;;; =====================================================================
;;; HELP-DBASE-ANSWER
;;;=====================================================================
;;;  (help-dbase-answer)
;;;   This function checks if answer is y or n, else asks the question
;;;
;;;
(defun help-dbase-answer()
     (CUP 2)
     (CUF 45) (format t "Warning::~%")
     (CUF 45) (format t "Wrong selection!~%")
     (CUP 5 70) (format t "    ")
     (display-more-dbase-note))


;;; =====================================================================
;;;  VIEW-DATABASE
;;; =====================================================================
;;;   (view-database)
;;;    This function lets you look at any selected description in database.
;;;
;;;
(defun view-database ()
    (heading "View Database")
    (cond
       ((probef 'dbase)
    (setq my-inport (infile 'dbase))
    (print-dbase (read my-inport))
    (close my-inport))
       (t (dbase-empty))))

;;; =====================================================================
;;;  DBASE-EMPTY
;;; =====================================================================
;;;  (dbase-empty)
;;;   This function displays message that there are no records in database
;;;
;;;
(defun dbase-empty()
    (init-graphics-mode :lines 5 :cursor-loc (make-Point x 300 y 200))
    (terpri)
    (message  "No records in database")
    (message "Use any mouse button to select from menu")
    (menu-selection (menu-choose *menu*)))

;;; =====================================================================
;;; PRINT-DBASE
;;; =====================================================================
;;;  (print-dbase <lst>)
;;;   lst: database descriptions
;;;
     This function displays a record from database.
;;;
(defun print-dbase ( lst)
    (cond
      ((null lst) (dbase-empty))
      (t ( clear-text-region)
         (CUP 2)
         (CUF 4) (format t "Record-number: ~a~% " (caar lst))
         (terpri)
         (print-record (cdr lst)(cdar lst) 1))))

;;; ========================================================================
;;; PRINT-RECORD
;;; ========================================================================
;;;  (print-record <dlist> <rlist> <num>)
;;;   dlist: rest of database
;;;   rlist: present record to be displayed
;;;   num: record number
;;;
(defun print-record (dlist rlist num)
   (cond
      ((null rlist) (display-more-records-note dlist))
      ((greaterp num 15) (display-more dlist rlist))
      (t (print-pairs (car rlist))
         (print-record dlist (cdr rlist) (add1 num)))))

;;;=======================================================================
;;; DISPLAY-MORE
;;;=======================================================================
;;;  (display-more <dlist> <rlist>)
;;;   dlist: rest of
(defun display-more ( dlist rlist)
   (CUP 20 4)
   (format t "Note::~%")
   (format t "     Hit return to continue: ")
   (cond
      ((eq (tyipeek) 13)  (tyi) (clear-text-region)
       (CUP 2 4)
       (print-record dlist rlist 1))
      (t(display-more dlist rlist))))
;;;  ======================================================================
;;;   PRINT-PAIRS
;;;  ======================================================================
;;;
(defun print-pairs (lst)
   (format t "     ~a:  " (car lst))
   (format t "~a~%" (cadr lst)))

;;;  ======================================================================
;;; DISPLAY-MORE-RECORDS-NOTE
;;;  ======================================================================
;;;
(defun display-more-records-note (lst)
   (CUP 4)
   (CUF 45) (format t "Note::~%")
   (CUF 45) (format t "View more records? (y/n)..")
   (let ((reply (my-ratom)))
      (cond
         ((eq reply 'y)  (print-dbase lst))
         ((eq reply 'n) (start))
         (t ( CUP 2)
            (CUF 45) (format t "Warning::~%")
            (CUF 45) (format t "Wrong selection!")
            (CUP 5 68) (format t "              ")
```

```
              (display-more-records-note lst)))))

;;; ==================================================================
;;;  EDIT-DATABASE
;;; ==================================================================
;;;
(defun edit-database ()
   (heading "Edit-database")
   (CUP 2 4) (format t "Record-number?..")
   (let ((reply (my-ratom)))
      (cond
         ((not (numberp reply)) (display-wrong-number-note))
         (t (get-record reply)))))

;;; ==================================================================
;;; DISPLAY-WRONG-NUMBER-NOTE
;;; ==================================================================
;;;
(defun display-wrong-number-note ()
   (CUP 2)
   (CUF 45) (format t "Warning::~%")
   (CUF 45) (format t "Wrong Selection!")
   (CUP 2 20) (format t "                            "))

;;; ==================================================================
;;; GET-RECORD
;;; ==================================================================
;;;
(defun get-record (num)
   (setq my-inport (infile 'dbase))
   (let ((lst (search-record num (read my-inport))))
      (cond
         ((eq lst nil) (CUP 2) (close my-inport)
                       (CUF 45) (format t "Warning::~%")
                       (CUF 45) (format t "Record #~a not in database~%" num)
                       (display-edit-more-note)
                       (get-dedit-answer(my-ratom)))
   (t (del-or-mod-record lst num)
   (close my-inport)))))

;;; ==================================================================
;;; SEARCH-RECORD
;;; ==================================================================
;;;
(defun search-record (num lst)
   (cond
      ((null lst) nil)
      ((eq num (caar lst)) (cdar lst))
      (t (search-record num (cdr lst)))))

;;; ==================================================================
;;; DEL-OR-MOD-RECORD
;;; ==================================================================
;;;
(defun del-or-mod-record ( lst num)
   (CUP 4)
   (CUF 45) (format t "Note::~%")
   (CUF 45) (format t "1. Delete record~%")
   (CUF 45) (format t "2. Modify record~%")
```

```
  (terpri)
  (CUF 45) (format t "Enter Selection..")
  (let ((reply (my-ratom))
    (cond
      ((eq reply 1) (delete-record num))
      ((eq reply 2) (modify-record lst num))
      (t (CUP 2)
        (CUF 45) (format t "Warning::~%")
        (CUF 45) (format t "Wrong selection!~%")
        (CUP 8 62) (format t "          ")
        (del-or-mod-record lst num)))))

;;; ===================================================================
;;; DELETE-RECORD
;;; ===================================================================
;;;
(defun delete-record (num)
  (cond
    ((probef 'dbase)
  (setq my-inport (infile 'dbase))
  (print (delete-help num (read my-inport) nil)
         (setq my-outport (outfile 'dbase)))
  (close my-outport)
  (close my-inport)(display-delete-note num )) (t 't)))

;;; ===================================================================
;;; DELETE-HELP
;;; ===================================================================
;;;
(defun delete-help (num lst lst1)
  (cond
    ((null lst) lst1)
    ((eq num (caar lst)) (delete-help num  (cdr lst) lst1))
    (t (delete-help num (cdr lst) (my-cons lst1 (changing-num (car lst) num)))))

(defun changing-num (lst num)
  (cond
    ((lessp (car lst) num) lst)
    (t (append (list (diff (car lst) 1)) (cdr lst)))))

;;; ===================================================================
;;; MODIFY-RECORD
;;; ===================================================================
;;;
(defun modify-record ( lst num)
  (read-file 'dbase)
  (let ((dir-list (read my-inport)))
    (close my-inport)
    (write-file 'dbase)
    (print (modify-dbase  dir-list (get-new-record lst lst nil num 2 25 ) num ni
    (close my-outport)
    (display-modify-note num)))

(defun modify-dbase (dir-list new-rec num new-lst)
  (cond
    ((null dir-list) new-lst)
    ((eq (caar dir-list) num) (modify-dbase (cdr dir-list) new-rec num (my-cons
    (t (modify-dbase (cdr dir-list) new-rec num (my-cons new-lst (car dir-list))
;;; ======================================================================
;;; GET-NEW-RECORD
;;; ======================================================================
;;;
(defun get-new-record (lst lst1 lst2 num pos y)
   (cond
      ((null lst) (get-help lst1 lst2 num 2 y))
      ((greaterp pos 15) (get-help lst1 lst2 num 2 y))
      (t (CUP pos 4) (format t "~a~%" (caar lst))
         (CUU 1) (CUF 24) (format t "~a~%"(cadar lst))
         (get-new-record (cdr lst)lst1  lst2 num  (add1 pos) y))))

;;; ======================================================================
;;; GET-NEW-RECORD
;;; ======================================================================
;;;
(defun get-help (lst lst1 num x y)
   (CUP x y)
   (cond
      ((null lst) (append (list num) lst1))
      ((greaterp x 15) (clear-text-region) (get-new-record lst lst lst1 num 2 y))
      ((eq (tyipeek) 13) (tyi) (get-help (cdr lst) (my-cons lst1 (car lst))
                                          num (add1 x) y))
      (t (get-help (cdr lst) (my-cons lst1 ( list (caar lst) (my-ratom)))
                          num (add1 x ) y))))

;;; ======================================================================
;;;  DISPLAY-MODIFY-NOTE
;;; ======================================================================
;;;
(defun display-modify-note (num)
   (CUP 4)
   (CUF 45) (format t "Note::~%")
   (CUF 45) (format t "Record #~a modified   ~%" num)
   (display-edit-more-note)
   (CUP 8 45) (format t "                         ")
   (CUP 6 73)
   (get-dedit-answer (my-ratom)))

;;; ======================================================================
;;;  DISPLAY-DELETE-NOTE
;;; ======================================================================
;;;
(defun display-delete-note (num)
   (CUP 4)
   (CUF 45) (format t "Note::~%")
   (CUF 45) (format t "Record #~a deleted   ~%" num)
   (display-edit-more-note)
   (CUP 8 45) (format t "                         ")
   (CUP 6 73)
   (get-dedit-answer (my-ratom)))

;;; ======================================================================
;;; GET-DEDIT-ANSWER
;;; ======================================================================
;;;
(defun get-dedit-answer(reply)
   (cond
      ((eq reply 'y) (clear-text-region) (edit-database))
      ((eq reply 'n) (start))
      (t (CUP 3)
         (CUF 45) ( format t "Wrong selection!                    ")
         (CUP 6 73) (format t "    ")
         (CUP 6 73) (get-edit-answer (my-ratom)))))

;;; ===========================================================================
;;; MATCH
;;; ===========================================================================
;;;
;;;  This is the main routine which gets the description from the user and
;;;  compares it to each description present in the database.
;;;
(defun match ()
   (heading "Match-descriptions:")

   (read-file 'directory)
   (get-attribute-info)
   (close my-inport)

   (read-file 'directory)
   (setq my-inport1 (infile 'dbase))
   (let ((alist (make-attribute-list nil))
         (dlist (read my-inport1)))
      (match-help dlist (get-desc alist alist nil 2)))

   (close my-inport1)
   (close my-inport)
   (call-menu))

  ;===========================================================================
;;; D-MATCH
;;;===========================================================================
;;;  This is the second version of match routine described above. This is
;;;  instead of match when program is used for demonstration.
;;;
(defun D-match ()
   (heading "Match-descriptions:")

   (read-file 'directory)
   (get-attribute-info)
   (close my-inport)

   (read-file 'directory)
   (setq my-inport1 (infile 'dbase))
   (let ((alist (make-attribute-list nil))
         (dlist (read my-inport1)))
      (D-match-help dlist (get-desc alist alist nil 2)))

   (close my-inport1)
   (close my-inport)
   (call-menu))

;;; ===========================================================================
;;; GET-ATTRIBUTE-INFO
;;; ===========================================================================
  ;
  ;  This function reads information from directory and puts function names
;;;  and weights as properties to attributes. Thus it  reduces searching
;;;  time.
```

```
;;;
(defun get-attribute-info ()
   (cond
      ((eq (tyipeek my-inport) -1) t)
      (t (add-attribute (read my-inport))
         (get-attribute-info))))

;;; ====================================================================
;;;  ADD-ATTRIBUTE
;;; ====================================================================
;;; This function gets function-name and weight of attribute from directory
;;;  and puts them on the property list of attribute. This saves the
;;;  the searching time.
;;;
(defun add-attribute (lst)
   (putprop (car lst) (caddr lst) 'fname)
   (putprop (car lst) (cadr lst) 'weight))

;;; ====================================================================
;;; GET-DESC
;;; ====================================================================
;;;
;;;  This function displays attributes known in directory on screen and gets
;;;  the values from the user. It returns the description to the calling
;;;  program.
;;;
(defun get-desc (alist alist1 temp pos)
    (cond
       ((null alist) (get-values alist1 temp 2 0))
       ((greaterp pos 15) (get-desc alist alist (get-values alist1 temp 2 0) 2))
       (t (CUP pos 4) (format t "~a..~%" (car alist))
          (get-desc (cdr alist) alist1 temp (add1 pos)))))

;;;====================================================================
;;; MATCH-HELP
;;;====================================================================
;;;
(defun match-help (database desc)
   (let ((sdatabase (sort-database database nil))
         (sdesc     (sort-data desc)))
       (results (sort-results (compare sdatabase sdesc nil) nil))))

(defun sort-results(lst slist)
   (cond
      ((null lst) (sort-help (sortcar slist '>) nil))
      (t (sort-results (cdr lst)
                       (my-cons slist (append (cadar lst) (list(caar lst))))))))


(defun sort-help (lst slist)
   (cond
      ((null lst) slist)
      (t (sort-help (cdr lst) (my-cons slist (list (car (last (car lst)))
                                (delete (car (last (car lst))) (car lst)))))))
;;; ====================================================================
;;; D-MATCH-HELP
;;; ====================================================================
;;
;;;  This function sorts each record of database and description in ascending
;;;  order by their attributes. this helps in finding the missing attributes.
;;;  then it calls compare to compare descriptions.
;;;
(defun D-match-help (database desc)
   (let ((sdatabase (sort-database database nil))
         (sdesc     (sort-data desc)))
        (screen-demo (compare sdatabase sdesc nil) database desc 20 50)))

;;; ========================================================================
;;; SORT-DATA
;;;========================================================================
;;; This function sorts the given list by its first elements in ascending
;;; order.
;;;
(defun sort-data (lst)
   (sortcar lst nil))

;;; ========================================================================
;;; SORT-DATABASE
;;;========================================================================
;;; This function sorts each record in database.
;;;
(defun sort-database (database lst)
   (cond
      ((null database))
      (t (cons (cons (caar database) (sort-data (cdar database)))
               (sort-database (cdr database) lst)))))
;;; ========================================================================
;;; COMPARE
;;; ========================================================================
;;;  This function compares each description in database to given description
;;;  by calling match-desc which for each description returns a triple of
;;;  similiarity measures. All these lists are then made into one list and
;;;  returned.
;;;
(defun compare (database desc lst)
   (cond
      ((null database) lst)
      ((eq database t) lst)
      (t (compare (cdr database) desc
                  (my-cons lst (match-desc (cdar database) desc
                                           (caar database)))))))

;;; ========================================================================
;;; MATCH-DESC
;;; ========================================================================
;;;
;;;  This function matches the given descriptions. It creates a list which is
;;;  list of three lists on the basis of attributes common to both descriptions
;;;  or present only in either of them.
;;;
(defun match-desc (recl rec2 num)
   (let (( new-list (make-lists recl rec2 nil nil nil)))
      (calc-sim new-list num)))

;;; ========================================================================
;;;  MAKE-LISTS
;;; ========================================================================
;;;  Given two records, this function compares both by their attributes. The
;;;  common attributes to both, are put into mlist and if found only in recl
;;;  put into alist else in blist. Then it returns list of these three lists.
```

```
;;;
(defun make-lists (rec1 rec2 mlist alist blist)
    (cond
       ((and (null rec1)(null rec2))
        (list mlist alist blist))
       ((and (eq (cadar rec1) nil)
             (eq (cadar rec1) nil))
        (make-lists (cdr rec1) (cdr rec2) mlist alist
                    (append blist (list (car rec1)))))
       ((or (eq (cadar rec1) nil)
            (eq (cadar rec2) nil))
        (make-lists (cdr rec1) (cdr rec2) mlist
                    (append alist (list (car rec1))) blist))
       (t(make-lists (cdr rec1) (cdr rec2)
                     (append mlist (merge (car rec1) (car rec2)))
                    alist blist))))

;;;=========================================================================
;;; MERGE
;;;=========================================================================

(defun merge (lst1 lst2)
   (list (append lst1 (cdr lst2))))

;;;==========================================================================
;;;  CALC-SIM
;;;==========================================================================

(defun calc-sim (lst num)
    (let ((a (float (calc-a (car lst)0)))
          (b (float (calc-b (car lst)0)))
          (c (float (calc-c (cdr lst)0))))
    (list num (list (round (quotient a b))
          (round (quotient a (plus b c)))
          (round (quotient (plus a c) (plus b c))))))

;;; ========================================================================
;;; CALC-A
;;; ========================================================================

(defun calc-a (lst num)
   (cond
     ((null lst) num)
     (t(calc-a (cdr lst)(plus num (times (get (caar lst) 'weight)
                              (funcall (get (caar lst) 'fname) (car lst)))))))

;;; ========================================================================
;;;CALC-B
;;;========================================================================
;;;
(defun calc-b (lst num)
   (cond
        ((null lst) num)
        ((eq lst nil) num)
        (t (calc-b (cdr lst)(plus num (get (caar lst) 'weight))))))

;;; ========================================================================
;;; CALC-C
;;; ========================================================================
;;;
;;;
```

```
(defun calc-c (lst num)
      (cond
        ((null lst)num)
        ((eq lst nil) num)
        (t(calc-c (cdr lst) (plus num (calc-b (car lst) 0))))))

;;; ================================================================
;;; ROUND
;;; ================================================================
;;;
(defun round (num)
  (quotient ( float (fix  ( times num 1000.0))) 1000.0))

;;;================================================================
;;; RESULTS
;;;================================================================
;;;
(defun results (lst)
   (init-graphics-mode :lines 5)
   (draw-line (make-Point x 0 y 10) (make-Point x 625 y 10))
   (paint-string 10 20 "Partial-Match Values:")
   (draw-line (make-Point x 0 y 40) (make-Point x 625 y 40))
   (draw-line (make-Point x 300 y 40) (make-Point x 300 y 400))
   (paint-string 60 55 " V      Vmin      Vmax")
   (results-help  lst  1 70 80)
   (draw-graph lst))


;;;================================================================
;;;RESULTS-HELP
;;;  ==============================================================
;;
(defun results-help (lst count x y)
   (cond
      ((null lst))
      ((greaterp count 5))
      (t (paint-string 5 y "Desc#")
         (paint-string 45 y (caar lst))
         (write-string x y (explode (caadar lst)) 0 0)
         (write-string (plus 60 x) y (explode (cadadar lst)) 0 0)
         (write-string (plus 120 x) y (explode (caddadar lst)) 0 0)
         (results-help (cdr lst) (add1 count) x (plus 25 y)))))

;;;================================================================
;;; WRITE-STRING
;;;================================================================
;;;
(defun write-string (x y num cnt flag)
   (cond
      ((eq cnt 3))
      ((eq num nil))
      ((equal (car num) '|.|)
       (paint-string x y (car num))
       (write-string (plus 8 x) y (cdr num) cnt flag))
      ((eq flag 1)
       (paint-string x y (car num))
       (write-string (plus 8 x) y (cdr num) (add1 cnt) flag))
      (t (paint-string x y (car num))
         (write-string (plus 8 x) y (cdr num) (add1 cnt) flag))))
```

```
;;; ===================================================================
;;; SCREEN-DEMO
;;; ===================================================================
;;;
(defun screen-demo (lst database desc xval yval)
   (init-graphics-mode :lines 5)
   (draw-line (make-Point x 0 y 200) (make-Point x 625 y 200))
   (demo1 lst database desc xval yval)
   (draw-rooms1 20 250 desc)
   (draw-graph lst))

;;; ===================================================================
;;; DEMO1
;;; ===================================================================
;;;
(defun demo1 (lst database desc xval yval)
   (cond
      ((null database) 'done)
      (t (draw-rooms xval yval (cdar database)(caar database) (car lst))
         (demo1 (cdr lst) (cdr database) desc (plus 120 xval) yval))))

;;; ===================================================================
;;; DRAW-ROOMS
;;; ===================================================================
;;;
(defun draw-rooms (xval yval dlist num lst)
    (draw-line (make-Point x xval y yval ) (make-Point x (plus 100  xval) y yval))
    (draw-line (make-Point x xval y yval) (make-Point x xval y (plus 100 yval)))
    (draw-line (make-Point x (plus 100 xval) y yval) (make-Point x (plus 100 xval)
    (draw-line (make-Point x (plus 100 xval) y (plus 100 yval)) (make-Point x xval
    (draw-stripes xval yval (get-number  dlist) (get-shade dlist))
    (paint-string xval (diff yval 20) "Rec#")
    (paint-string (plus 35 xval) (diff yval 20) num)
    (paint-string xval (plus yval 110) "V:")
    (paint-string (plus 20 xval) (plus yval 110) (caadr lst))
    (paint-string xval (plus yval 120) "V-min:")
    (paint-string (plus xval 45) (plus yval 120) (cadadr lst))
    (paint-string xval (plus yval 130) "V-max:")
    (paint-string (plus xval 45) (plus yval 130) (caddadr lst)))

;;;===================================================================
;;; DRAW-ROOMS1
;;;===================================================================
;;;
(defun draw-rooms1 (xval yval dlist)
    (draw-line (make-Point x xval y yval ) (make-Point x (plus 100  xval) y yval))
    (draw-line (make-Point x xval y yval) (make-Point x xval y (plus 100 yval)))
    (draw-line (make-Point x (plus 100 xval) y yval) (make-Point x (plus 100 xval)
    (draw-line (make-Point x (plus 100 xval) y (plus 100 yval)) (make-Point x xval
    (draw-stripes xval yval (get-number  dlist) (get-shade dlist)))


;;;===================================================================
;;; DRAW-STIPES
;;; ===================================================================
;;;
(defun draw-stripes (xval yval num shade)
    (cond
      ((not (numberp num)) (dummy xval yval shade))
      ((eq num 0) 'done)
      (t(fill-up xval yval shade)
        (draw-stripes (plus 20 xval) yval (diff num 1) shade))))

;;; ===========================================================================
;;; GET-SHADE
;;; ===========================================================================
;;;
(defun get-shade (dlist)
   (cond
      ((null dlist) 'missing)
      ((eq (caar dlist) 'shade) (cadar dlist))
      (t (get-shade (cdr dlist)))))

;;; ==========================================================================
;;; GET-NUMBER
;;; ==========================================================================
;;;
(defun get-number (dlist)
   (cond
      ((null dlist) 'missing)
      ((eq (caar dlist) 'number) (cadar dlist))
      (t (get-number (cdr dlist)))))

;;;==========================================================================
;;; FILL-UP
;;; =========================================================================
;;;
(defun fill-up (xval yval shade)
   (cond
      ((eq shade 'missing)
       (draw-line (make-Point x (plus 10 xval) y yval)
                  (make-Point x (plus 10 xval) y (plus 100 yval)))
       (draw-line (make-Point x (plus 20 xval) y yval)
                  (make-Point x (plus 20 xval) y (plus 100 yval))))
      ((eq shade 'gray)
       (draw-rectangle (make-Rect x (plus 10 xval) y yval  w 10 h 100)
                       :halftone GrayHalftone))
      ((eq shade 'darkgray)
       (draw-rectangle (make-Rect x (plus 10 xval) y yval  w 10 h 100)
                       :halftone DarkGrayHalftone))
      ((eq shade 'lightgray)
       (draw-rectangle (make-Rect x (plus 10 xval) y yval  w 10 h 100)
                       :halftone LightGrayHalftone))
      ((eq shade 'verylightgray)
       (draw-rectangle (make-Rect x (plus 10 xval) y yval  w 10 h 100)
                       :halftone VeryLightGrayHalftone))
      (t (draw-rectangle (make-Rect x (plus 10 xval) y yval w 10 h 100)))))

;;;==========================================================================
;;; DUMMY
;;;==========================================================================
;;;
(defun dummy (xval yval shade)
   (cond
      ((eq shade 'gray)
       (draw-rectangle (make-Rect x xval y yval w 100 h 100) :halftone GrayHalfton
      ((eq shade 'darkgray)
       (draw-rectangle (make-Rect x xval y yval w 100 h 100) :halftone DarkGrayHal
      ((eq shade 'lightgray)
      (draw-rectangle (make-Rect x xval y yval w 100 h 100) :halftone LightGrayHa
     ((eq shade 'verylightgray)
      (draw-rectangle (make-Rect x xval y yval w 100 h 100) :halftone VeryLightGr
     (t (draw-rectangle (make-Rect x xval y yval w 100 h 100)))))

;;;=========================================================================
;;; DRAW-GRAPH
;;;=========================================================================
;;;
(defun draw-graph (lst)
   (draw-graph-struct1 350 220)
   (draw-bars1 lst 350 1))

;;;=========================================================================
;;; DRAW-GRAPH-STRUCT1
;;; ========================================================================
;;;
(defun draw-graph-struct1 (xval yval)
   (draw-line (make-Point x xval y yval)
              (make-Point x xval y (plus yval 160)))
   (draw-line (make-Point x xval y (plus yval 160))
              (make-Point x (plus xval 200) y (plus yval 160)))
   (setq p1 (make-Point x (diff xval 5) y (plus yval 140)))
   (setq p2 (make-Point x xval y (plus yval 140)))
   (draw-help1 p1 p2 0))

;;;=========================================================================
;;; DRAW-HELP1
;;;=========================================================================
(defun draw-help1 (p1 p2 count)
   (cond
     ((eq count 10) (write-num1 320 220 0 '("1.0" "0.9" "0.8" "0.7"
                                            "0.6" "0.5" "0.4" "0.3"
                                            "0.2" "0.1")))
     (t (draw-line p1 p2)
        (draw-help1 (make-Point x (Point->x p1) y (diff (Point->y p1) 15))
                    (make-Point x (Point->x p2) y (diff (Point->y p2) 15))
                    (add1 count)))))

;;;=========================================================================
;;; WRITE-NUM1
;;;=========================================================================
;;;
(defun write-num1 (x y count num)
   (cond
      ((eq count 10) t)
      (t (paint-string x y (car num))
         (write-num1 x (plus y 15) (add1 count) (cdr num)))))

;;;=========================================================================
;;; DRAW-BARS1
;;;=========================================================================
;;;
(defun draw-bars1 (lst xval cnt)
   (cond
      ((null lst))
      ((greaterp cnt 5))
      (t (draw-Vmin-bar (cadar lst) (plus xval 20))
         (draw-V-bar (cadar lst) (plus xval 25) (caar lst))
         (draw-Vmax-bar (cadar lst) (plus xval 30))
        (draw-bars1 (cdr lst) (plus xval 40) (add1 cnt)))))

;;;============================================================================
;;; DRAW-V-BAR
;;;============================================================================
;;;
(defun draw-V-bar (lst x-val num)
   (draw-rectangle (make-Rect x x-val y (fix (diff 380 (times 160 (car lst))))
                              w 5 h (fix (times 160 (car lst))))
                              :halftone GrayHalftone)
   (paint-string x-val 385 num))

;;;============================================================================
;;; DRAW-VMIN-BAR
;;;============================================================================

(defun draw-Vmin-bar (lst x-val)
   (draw-rectangle (make-Rect x x-val y (fix (diff 380 (times 160 (cadr lst))))
                              w 5 h (fix (times 160 (cadr lst))))
                              :halftone LightGrayHalftone))

;;;============================================================================
;;; DRAW-VMAX-BAR
;;;============================================================================

(defun draw-Vmax-bar (lst x-val)
   (draw-rectangle (make-Rect x x-val y (fix(diff 380 (times 160 (caddr lst))))
                              w 5 h (fix (times 160 (caddr lst))))
                              :halftone DarkGrayHalftone))


;;; ===========================================================================
;;; MY-CONCAT
;;; ===========================================================================
;;;  (my-concat <count>)
;;;
;;;  RETURNS:  If the argument to my-count is one-digit number, then it
;;;            returns argument concatenated by a space before it and period
;;;            after it. If it is two-digit number, then it returns the
;;;            argument with period after it.
;;;
;;;  ARGUMENT:
;;;  1. count: integer
;;;
;;;  Example:  -> my-concat(8)
;;;                8.
;;;            -> my-concat(10)
;;;               10.
;;;
(defun my-concat (count)
   (cond
      ((lessp count 10) (concat " " count "."))
      (t(concat count "."))))

;;; ===========================================================================
;;;  MY-CONS
;;; ===========================================================================
;;; (my-cons <lst1> <lst2>)
;;;
```

```
;;;   RETURNS: list of both argumnets.
;;;
;;;   ARGUMENTS:
;;;   1. lst1: nil, atom or list
;;;   2. lst2: nil, atom or list
;;;
;;;  Example     -> (my-cons 2 nil)
;;;                 (2)
;;;              -> (my-cons nil 2)
;;;                 (nil . 2)
;;;              -> (my-cons '(a) '(b))
;;;                 (a (b))
;;;              -> (my-cons '(a) 'b)
;;;                 (a b)
;;;
(defun my-cons (lst1 lst2)
   (cond
      ((eq lst1 nil) (list lst2))
      ((eq lst2 nil) (list lst1))
      (t (append lst1 (list lst2)))))

;;; ===========================================================================
;;; MESSAGE
;;; ===========================================================================
;;;  (message <string>)
;;;
;;;  Side-effect:  Prints the string in the center of the text scrolling region
;;;
(defun message (string)
   (msg (B (quotient (- 80 (flatsize string)) 2)) string N))


;;; ===========================================================================
;;; MY-DELETE
;;; ===========================================================================
;;;   (my-delete <lst1> <lst2> <lst3>)
;;;
;;;   RETURNS: <lst3> is new list created which all the lists from <lst2>
;;;           except those whose first elements are same as first elements
;;;           in <lst1>
;;;
;;;   ARGUMENTS:
;;;      lst1 : list of atoms
;;;      lst2 : list of lists, each list is list of atoms
;;;      lst3 : new list created
;;;
;;;   Example: -> (my-delete '(a b) '((d v) (c f) (a x) (e g)) nil)
;;;                ((d v) (c f) (e g))
;;;
(defun my-delete (lst1 lst2 lst3)
   (cond
      ((null lst2) lst3)
      ((eq (car lst1) (caar lst2)) (my-delete lst1 (cdr lst2) lst3))
      (t (my-delete lst1 (cdr lst2) (my-cons lst3 (car lst2))))))


;;; ===========================================================================
;;;  READ-FILE
;;; ===========================================================================
;;;   (read-file <filename>)
;;;
;;;  RETURNS: Sets my-inport to port for reading file. If an error occurs, then
;;;           returns nil.
;;;
;;;  ARGUMENT:
;;;  1. filename: file to be opened for reading
;;;
(defun read-file (filename)
   (cond
      ((null (setq my-inport (car (errset (infile  filename) nil)))) nil)
      (t 'my-inport)))


;;; ==========================================================================
;;; WRITE-FILE
;;; ==========================================================================
;;;   (write-file <filename>)
;;;
;;;  RETURNS: Sets my-outport to port for writing. The previous contents are
;;;           lost.
;;;
;;;  ARGUMENTS:
;;;  1. filename: file to be opened for writing
;;;
(defun write-file (filename)
   (setq my-outport (outfile filename)))



;;; ==========================================================================
;;; APPEND-FILE
;;; ==========================================================================
;;;   (append-file <filename>)
;;;
;;;  RETURNS: Sets my-outport to port for writing. File pointer is set to end
;;;  end of file.
;;;
;;;  ARGUMENTS:
;;;  1. filename: file to be opened in append mode
;;;
(defun append-file (filename)
   (setq my-outport (outfile filename 'append)))


;;; ==========================================================================
;;;  HEADING
;;; ==========================================================================
;;;  (heading <string>)
;;;
;;;  RETURNS: t
;;;
;;;  SIDE-EFFECT: Changes the text scrolling region to 29 lines and displays
;;;  given string in the left corner of the display region.
;;;
;;;  ARGUMENT:
;;;  1. string: string to be printed
;;;
```

```
(defun heading (string)
   (init-graphics-mode :lines 29)
   (draw-line (make-Point x 0 y 10) (make-Point x 630 y 10))
   (paint-string 10 20 string))


(defun my-ratom ()
   (let((a(ratom)))
      (tyi) a))



;;;=============================================================================
;;;
;;;
;;;   FILE:  /lisp/examples/draw.l
;;;   ===========================
;;;
;;;
;;;   INIT-GRAPHICS-MODE
;;;   ------------------
;;;
;;;      (init-graphics-mode :lines <num> :cursor-loc <point>)
;;;
;;;      Turn on Lisp graphics mode and sets the graphic region. Text scrolling
;;;      region is <num> lines and a visible graphic cursor is at <point>
;;;
;;;
;;;
;;;   EXIT-GRAPHICS-MODE
;;;   ------------------
;;;
;;;      (exit-graphics-mode)
;;;
;;;      Restore full screen text scrolling. The screen is cleared and the
;;;      text cursor is located in the upper left-hand corner of the display.
;;;
;;;
;;;   DRAW-LINE
;;;   ---------
;;;
;;;      (draw-line point1 point2)
;;;
;;;      Draw a line from point1 to point2 which are Point structures
;;;      created by make-Point.
;;;
;;;
;;;   CLEAR-TEXT-REGION
;;;   -----------------
;;;
;;;      (clear-text-region)
;;;
;;;      Erase the scrolling text region, and home the cursor to the upper left
;;;      corner of the region.
;;;
;;;
;;;    FILE:  /lisp/examples/menu.l
;;;    ===========================
;;;
```

```
;;;   MAKE-MENU
;;;   ---------
;;;
;;;      (make-menu <item-list>)
;;;
;;;      Create a menu from the item-list. A menu is a symbol with associated
;;;      structures and values maintained on a property list. The item-list is
;;;      a list of items, each of which is associated with one slot of menu.
;;;
;;;   MENU-CHOOSE
;;;   -----------
;;;      (menu-choose  <menu>)
;;;
;;;      Pop up a menu and wait for a selection. A <menu> is an object returned
;;;      by make-menu. The menu is centered at the location of the graphic-
;;;      cursor. If the selector is pressed and released outside of the menu
;;;      nil is returned.
;;;
;;;
;;;   PAINT-STRING
;;;   ------------
;;;      (paint-string <x> <y> <string>)
;;;
;;;      Paint the specified <string>. The upper left corner of the first
;;;      character image is located at <x> <y>.
;;;========================================================================

(defun start1 () (tyi) (start)

;;;========================================================================
```