November 10, 1983

83-30

A Description and User's Guide for CLUSTER/2
A Program for Conjunctive Conceptual Clustering

Robert Stepp

University of Illinois
Department of Computer Science
Intelligent Systems Group

ABSTRACT

This report presents a comprehensive reference for using the program CLUSTER/2 for constructing classifications of arbitrary objects. Specifically, given a set of object descriptions (in the form of attribute-value pairs) the program structures the objects into a hierarchy of classes. Each class is characterized by a conjunctive description involving selected object attributes. Object descriptions and control parameters are presented to the program in the form of relational tables. This document contains the definitions of the relational table syntax and the values that can be entered. Sample input and the corresponding program output generated by CLUSTER/2 are discussed.

1. Introduction

CLUSTER/2 is a program for automatically constructing conceptual classifications. It is the successor to the program CLUSTER/PAF [1],[2]. CLUSTER/2 may be applied to a wide variety of problem domains. Experimentation with the program has included varied practical problems such as classifying Spanish folksongs [3], classifying microcomputers [4], and reconstructing soybean disease categories [4]. Input to CLUSTER/2 consists of a set of attribute vectors, (one vector per input event), definitions of the types of variables and their value sets, and a specification of the clustering quality criterion. The program generates a classification hierarchy of the data events along with a conjunctive description of each cluster at each level of the hierarchy. The program forms clusters that have a conjunctive description and optimize the given criterion of clustering quality. The program forms both clusters and their descriptions, and thus behaves quite unlike programs that implement conventional numerical taxonomy techniques. The theoretical background for CLUSTER/2 can be found in [4] and [5].

2. Input file content

CLUSTER/2 is run in batch mode. All parameters are provided from a single input file. The program creates one output file which is a report of the obtained clusters along with optional trace information which shows what alternative clusterings were considered while obtaining the final reported solution.

The input data are in the form of relational tables according to a standard syntax. A table consists of three parts: a table-identifier that is composed of one of the predefined table names, a line of column-headings (composed of predefined column names), and lines of data values arranged such that the order of the data on each line matches the order of the column headings. Rather than having a prescribed order, the order of the columns is determined by the order of the column-heading names. In the example tables that follow, the columns are arranged in a convenient order. The user of CLUSTER/2 may freely permute the order of the columns in any table.

Data items in one line of a table are separated by "white space" which is one or more of the characters: space, comma, exclamation point. There are three types of data values: integer, real, and character. Each type has a required format which is described below. In the table definitions that follow, the symbols <i>, <r>, and <c> will denote columns that take integer, real, and character data, respectively. These editorial symbols are not placed in the input file.

Some tables have default values which are used whenever a column is absent or when a row contains no value in a particular column. Since the relational table syntax uses no punctuation (just white space), a "place holder" symbol must be used whenever a value in a row is missing. The place holder symbols are '$' and '?', meaning "not applicable" and "unknown", respectively.

2.1. Integer data values

Integer data are entered as a signed integer number such as 10 or -3. For very large integers, scientific notation may be used in the following form: <sign><mantissa> E <exponent>. Examples of the scientific form are 1E6 (one million) and -2.3E7 (-23 million). In the scientific form, the given value must be integer after the application of the exponent. If a non-integer value is entered, an error message is produced and the program terminates.

2.2. Real data values

Real data are entered using any of the forms provided for integer data and also forms that have fractional parts, such as 2.3 and -3.487E2.

2.3. Character data values

Character data may be entered several ways. When only a single word is to be entered it may be written without surrounding quote marks provided that the word contains no "white space" characters and does not begin with $ or ?. For example, the character data values RED and BLUE may be entered this way. Character data may also be enclosed either by " or by '. The same quote character must both begin and end the character string. The other quote character may be used within the string. Character string data may be up to 80 characters long though some parameters receive their value from only the first character or the first 10 characters in the string. The following strings are legal.

onewordwithoutwhitespace
"A string containing 'apostrophe' characters"
'A string containing "quote" characters'

3. Relational table formats

3.1. Title table

The "title" table is used to provide the caption for the data analysis experiment. The title lines are written at the top of the output report. A sample title table is shown below. The word entered under "TYPE" is ignored in the present implementation. You may enter as many lines of title data as you want.

```
TITLE                                             {table identifier}
TYPE       TEXT                                   {column-headings}
<c>        <c>                                    {editorial symbols denoting data type}
MAIN      "ANALYSIS OF DISEASE DATA"              {1st data row}
SUBTITLE  "CHARCOAL ROT AND ROOT ROT"             {2nd data row}
```

CLUSTER/2 is initialized so that it is prepared to receive rows of the TITLE table without the need to give the table-identifier and the column headings (as if the table identifier and column-headings of the TITLE table had already been read in). This feature allows the input file to begin with the first title data line itself but also allows for the complete entry of the TITLE table exactly as given above. An input file that begins as shown below utilizes the implied definition of the TITLE table.

```
MAIN      "THIS IS MY MAIN-TITLE"
SUB       "THIS IS MY SUB-TITLE"
```

3.2. Parameters table

The PARAMETERS table indicates what you want CLUSTER/2 to do. This table must precede all other tables except TITLE. Each line of the PARAMETERS table requests a full iteration of the clustering algorithm, specifying possibly different cluster organizations (e.g., hierarchical or flat), numbers of clusters, and clustering criterion values, etc. Two simple PARAMETERS tables are shown below. They both rely on default settings for almost all parameters.

```
PARAMETERS
 K     CRITERION
<i>    <c>          {This table implies a flat (one-level) cluster organization
 3     ALPHA         and asks for two such clusterings, one with 3 clusters and
 4     ALPHA         one with 4 clusters. The user must define the "ALPHA"
                     clustering quality criterion (see section 3.3).}
```

PARAMETERS	
COVERTYPE	CRITERION
<c>	<c>
HIERARCHICAL	Q23

{This table requests a hierarchical cluster organization. CLUSTER/2 will build a classification hierarchy using the clustering quality criterion "Q23" (which must be further defined). The number of clusters formed at each level of the hierarchy will be in the default range of 2 to 4 clusters and no further partitioning will be done for clusters that contain no more than 4 events.}

The PARAMETERS table has many other columns which are optional. The following table shows all columns that are available along with the default values for each column.

PARAMETERS

K	MINK	MAXK	CRITERION	TRACE	H1	H2	H3	INITMETHOD	NIDSPEED	COVERTYPE	BASE	PROBE	MAXHEIGHT	MINSIZE	BETA
<i>	<i>	<i>	<c>	<c>	<i>	<i>	<i>	<c>	<c>	<c>	<i>	<i>	<i>	<i>	<r>
2	2	4	ALPHA	OFF	3	2	3	RANDOM	FAST	DISJOINT	2	2	99	4	2.0

The editorial symbols denoting type (e.g., <i>, <c>) are not part of the table and are never written in the input. The table can contain as many lines of data as desired. The columns are defined below.

BASE
BASE and PROBE control the clustering stopping criterion. For each number of clusters considered in finding a solution, CLUSTER/2 always forms BASE number of clusterings. Then PROBE more clusterings are generated (see PROBE below). If any of the PROBE number of clusterings is better than any previous clustering the better cluster description is remembered and another PROBE number of clusterings are generated. The algorithm works such that for each clustering a unique set of seed events is selected. When PROBE clusterings are generated without producing one that is better than a previously obtained one, the algorithm stops and reports the descriptions of the BASE number of best clusterings that were obtained. Increasing BASE increases the number of alternative clusterings that are reported.

BETA
BETA specifies the relative importance of good fit of the cluster descriptions to the data for different numbers of clusters formed. The system chooses the optimized number of clusters by measuring the fit multiplied by the number of clusters raised to the BETA power. This measure has been denoted S and is defined:

$$S = \text{sparseness} \times k^{\beta}$$

where sparseness is the measure of fit and k is the number of clusters. The optimal clustering is the one with the minimum value of S. Increasing BETA increases improvement of fit demanded for greater numbers of clusters (i.e., it tends to cause CLUSTER/2 to select clusterings with fewer clusters).

COVERTYPE
COVERTYPE specifies the desired organization of clusters. The specification DISJOINT causes the generation of a "flat," single-level clustering having a particular number of clusters, or the optimized number selected from a specified range. DISJOINT clusters do not overlap; each input event occurs in at most one cluster. The specification INTERSECTING causes the generation of a "flat," single-level clustering but with clusters that may intersect. This feature is not present in the current implementation. The specification HIERARCHICAL causes the generation of a multi-level hierarchy of clusters. The clustering algorithm is applied recursively to clusters formed until either the hierarchy contains a number of levels equal to parameter MAXHEIGHT or the cluster to be partitioned fails to have more than

MINSIZE number of events.

CRITERION — CRITERION specifies the name of the relational table defining a criterion of clustering quality. Such tables are described in section 3.3.

H1 — Parameters H1 to H3 control search heuristics. If these parameters are set to "infinity" (this cannot actually be done) then the results of the search for the best clustering are absolutely optimal. Since only trivial problems can be handled in this manner, given the combinatorial explosion that occurs, H1 to H3 are assigned small values and the heuristic search process yields a good, optimized solution but one that is not necessarily optimal. H1 specifies the number of potential cluster descriptions maintained during the process of generating alternative cluster descriptions. In terms of the clustering algorithm (see [4]), H1 gives the number of complexes kept in each partial star. Increasing H1 causes more descriptions to be considered. Values greater than 10 are usually unwarranted, as they do not generally yield an improved solution.

H2 — Parameter H2 specifies the maximum number of potential cluster descriptions produced for each cluster. The actual number maintained may be smaller than H2 since the search tree is tapered on one side and has fewer alternative solutions. In terms of the clustering algorithm, H2 gives the maximum number of complexes in a completed star. The affects of H2 are similar to those of H1. Typical values are also less than 10.

H3 — Parameter H3 controls the persistence of the search for optimized clusterings. The method inspects clusterings according to their Path-Rank-Order [4]. When the number of successively ordered clusterings inspected without finding a new, most optimized clustering reaches H3, the search terminates. Increasing H3 extends the search, possibly enabling it to find better results.

INITMETHOD — INITMETHOD specifies the technique to be used to select the first seed events. In the current implementation, the only value allowed is RANDOM.

K — K specifies the desired number of clusters. The value given for K is placed into both MINK and MAXK (see below).

MAXK — MAXK specifies the maximum number of clusters to form when generating a clustering. The algorithm tries clusterings containing a number of clusters ranging from MINK to MAXK.

MINK — MINK specifies the minimum number of clusters to form.

MAXHEIGHT — MAXHEIGHT specifies the maximum allowed height of a cluster hierarchy.

MINSIZE — MINSIZE specifies the minimum size (in terms of number of events) of a cluster that is subject to recursive clustering into smaller parts. Only clusters that contain more than MINSIZE number of events are further partitioned.

NIDSPEED — NIDSPEED specifies the technique used to make Non-disjoint clusters Into Disjoint ones. The value FAST causes CLUSTER/2 to use seed events as well as previously determined cluster descriptions to form an "against set" for building alternative cluster descriptions for clusters. This improves the speed of the program but places high weight on the clusters that happen to be formed first. This may lead to less-optimized results. The value SLOW causes CLUSTER/2 to use a cluster-adjusting algorithm (NID) whenever the clusters created using only seed events as the "against set" intersect one another. The cluster-adjustment algorithm adds to the solution time but the clusters are not weighted according to the order in which they are formed and can be more optimized.

PROBE — PROBE controls the persistence of the cluster formation process. The system forms a candidate clustering and compares it (according to the clustering quality criterion) to previously generated clusterings. The system performs PROBE number of such cycles looking for a better clustering. Each time such a clustering is found, another PROBE cycles are performed until PROBE number of successive cycles fails to find an improved solution. The algorithm also halts if all combinations of events have been considered as seeds. Increasing PROBE permits the program to explore more combinations of seed events and may permit the program to discover better clusterings.

TRACE — The TRACE column controls the generation of intermediate clustering descriptions in the output report file. The value of OFF limits output to just the final cluster descriptions. The value ON requests the output of supplemental descriptions for each intermediate clustering that is produced.

PRINT — The PRINT column may be defined in lieu of the TRACE column. The values entered under the PRINT column explicitly control the printout contents. The value given is a single character string composed of the letters A,C,E,I,N,P,T,V. Except for letter T, the order of the letters is unimportant. The letter I controls the printing of intermediate results and is totally equivalent to the ON setting of the TRACE option (above). If letter I is not present in a particular row of the PARAMETERS table, that part of the run is made with no printing of intermediate results. Each letter activates a different type of printout during the running of CLUSTER/2. If the PARAMETERS table contains many rows of data, the system processes the values in the PRINT column as a single specification, as if they were concatenated to form a single string of letters. Each letter has the following meaning:

A - enable all print functions
C - print the input -CRITERION tables
E - print the input EVENTS table
I - print intermediate results
N - print the input -NAMES, -ONAMES, -STRUCTURE tables
P - print the PARAMETERS table
T - print the TITLE table ONLY; this negates all other options
V - print the DOMAINS and VARIABLES tables

The initial setting is PC which normally prints PARAMETERS and -CRITERION tables. Giving a value such as E (or CPE) adds to the current print option the request to print the EVENTS table. All options except T are additive; T cancels all optional printout except for the TITLE data. The value TE first eliminates the current (PC default) option and then establishes just the E option, causing only the EVENTS table to be printed with the TITLE data that is always printed.

The PARAMETERS table directs the work performed by CLUSTER/2. Any combination of parameter values that is to be performed is entered as a separate row. The clustering algorithm is repeated for each row, using the parameters given on that row.

3.3. Criterion table

The -CRITERION table form may appear more than once in the input file. Each occurrence is given a unique name which consists of a specific part followed by a general part (-CRITERION). There must be a -CRITERION table for each name specified in the CRITERION column of the PARAMETERS table. From the previous PARAMETERS examples the table ALPHA-CRITERION and possibly the table Q23-CRITERION would need to be entered, depending on which names (ALPHA or Q23 or others) were used in the PARAMETERS table. Either one of the following two tables could be used to define the "ALPHA" criterion.

ALPHA-CRITERION

#	CRIT#	TOL
<i>	<i>	<r>
1	8	0.3
2	2	0.0

ALPHA-CRITERION

#	CRITERION	TOL
<i>	<c>	<r>
1	PSPARSENESS	0.3
2	DISJOINTNESS	0.0

The program applies the elementary evaluation criteria in a user-defined order. The column # specifies the order of application of the elementary criteria specified in the CRIT column. The numbers in the # column must be in numerical order, starting with 1.

CRIT#   Either the CRIT# column or the CRITERION column is used to specify which elementary criterion is applied. If the CRIT# column is used, the criteria must be indicated by criterion number. If the CRITERION column is used, the criteria must be indicated by criterion name. The elementary criterion numbers may be taken from the following definitions (see sec. 3.3.1). The elementary criteria are combined according to the values specified in the -CRITERION table to form the *Lexicographical Evaluation Functional (LEF)* [5] used to evaluate the clustering quality.

CRITERION   The CRITERION column may be used in lieu of the CRIT# column. When using CRITERION, the elementary criteria are identified by name, rather than by number (see sec 3.3.1).

TOL   TOL is a positive real number, normally less than or equal to 1.0. If TOL is larger than 1.0, it is an absolute tolerance. When evaluating the LEF, clusterings within this amount of the best one are judged to be equivalent. If TOL is less than or equal to 1.0, the tolerance is taken as the indicated fraction of the difference between the best and worst clustering scores for the elementary criterion. For example, with TOL set to 0.3 then clusterings scoring within 30% of the best clustering (on a scale from the best score to the worst score) are judged to be equivalent. A TOL of 1.0 causes all clusterings to be judged the same; a TOL of 0.0 causes no clusterings to be judged the same unless their scores are absolutely identical.

TOLERANCE   TOLERANCE is a synonym for TOL.

3.3.1. Elementary criteria available in CLUSTER/2

The following criteria are defined in CLUSTER/2. These criteria are component parts of a user-specified Lexicographical Functional with tolerances (LEF) [5]. They are specified in a -CRITERION table by their index number or name. CLUSTER/2 optimizes the generated clusterings by favoring clusters whose descriptions have the lowest scores for the elementary criteria specified in the LEF. If a negative elementary criteria index is given, or if "-" is placed directly in front of the name, the negative of the score is used. As CLUSTER/2 favors clusters with descriptions that have the lowest negatives, the effect is the same as favoring clusters whose descriptions have the highest scores.

The elementary criteria are presented in order by criterion index number. The "input specification" annotation gives the keyword used in the CRITERION column. Only the unique part of the name (capitalized) need be entered.

1. Sparseness (input specification: SParseness)

The sparseness of clustering is the sum of the sparsenesses of the complexes which comprise the clustering. The sparseness of a complex is the number of events it covers which are not given in the input data. Such events are called "unobserved" events. The sparseness is computed by calculating the "area" of the complex and subtracting from it the number of points which correspond to given (observed) events. The "area" of a complex is the number of points (each corresponding to a possible event) in the subset of the event space covered by the complex.

Example: Suppose that complex $[x_1=2,4][x_3=4..7]$ covers 4 observed events.
(assume the variables are x_1, x_2, x_3, and that the domain of x_2 contains 3 values)

The "area" of the complex is the product of the numbers of values in the references for each variable. In this example, area = 2 x 3 x 4 = 24 (2 values for x_1, 3 values for x_2–its entire domain, 4 values for x_3). The sparseness of this complex is "area" - #observed events or 24-4=20.

2. Disjointness (input specification: DISjointness)

The disjointness (degree of intersection) of a clustering is the sum of the degrees of intersection of each pair of complexes in the clustering. The degree of intersection of two complexes is the total number of selectors that involve the same variable and have intersecting reference sets. The following example involves three variables x_1, x_2, x_3.

Example:

α_1: $[x_1=1..3][x_2=4]$

α_2: $[x_1=2..5][x_2=2]$

α_3: $[x_1=4][x_2=3][x_3=2]$

The degree of intersection of complex α_1 with α_2 is 4 because the references used with variable x_1 in each complex contain intersecting values and the selectors omitted for x_3 also intersect (a "dropped" selector is equivalent to one with all possible domain values in the reference list). The degree of intersection of complex α_3 with α_1 is 2 because two selectors (for x_3) intersect. (In α_1 the missing selector for variable x_3 is presumed to have a reference set equal to the entire domain of x_3).

3. Number of events occurring in more than 1 complex (input specification: Multcov)

This criterion is for use only when considering intersecting clusters. It is not used in the current implementation.

4. Balance (input specification: Balance)

This criterion measures the unevenness in the cluster populations. It is the sum of the deviations from uniform distribution of observed events in each complex. For one complex, this deviation is calculated as the absolute value of the difference between the actual number of observed events covered and 1/k of the total number of the observed events, where k is the number of complexes (clusters).

5. Commonality (input specification: Commonality)

This criterion measures the number of common attributes in the cluster descriptions by counting the number of selectors in all clusters (dropped selectors are not counted). The negative of the number of selectors is used in order to maximize the commonality by minimizing the numerical score.

6. Dimensionality (input specification: DIMensionality)

This criterion measures the number of variables which take on different values in every complex. The number reported for each complex is the number of variables in the complex which occur in selectors in all complexes of the clustering with mutually disjoint reference values.

7. Simplicity (input specification: SImplicity)

The simplicity of a clustering is the sum of the simplicities of all selectors contained in all complexes in the clustering. The simplicity of one selector is the sum of the cost of the variable plus the cost of the reference list. Variable costs are provided by the user via the VARIABLES table (see Sec. 3.4) or given the default cost of 1. Reference list costs are computed according to the type of the domain. For structured domains, the cost is 0. For linear domains, the cost is 0 if only one value occurs in the reference list, otherwise the cost is 1. For nominal domains, the cost is the number of values in the reference list minus 1.

8. Projected Sparseness (input specification: Psparseness)

Projected sparseness is calculated just like regular sparseness, but only some of the dimensions of the event space are considered. When calculating the "area" used in calculating projected sparseness, only the variables which are present (not dropped) in at least one complex of the clustering are considered. This is equivalent to imagining that the domain sizes of all universally-dropped variables is just 1.

Example: Suppose that $[x_1=2,4][x_3=4..7]$ covers 4 observed events.

The factors used in computing the "area" of the complex depend, in part, on other complexes in the clustering. If, for this example, the variable x_5 has a domain containing 6 values and appears in a selector in some cluster description then it is used to calculate the "area" of the complex as follows:

$$\text{"area"} = 2 \times 4 \times 6$$

where 2 and 4 are the numbers of values in the reference lists for variables x_1 and x_3, respectively, and 6 is the number of levels in variable x_5.

The projected sparseness is then (2x4x6) - 4 = 44. Projected sparseness may be negative.

9. Number of exceptional events (input specification: Except)

Certain situations cause the formation of a clustering which fails to cover all the events. Any events not covered are placed into an exceptions category, which is shown on the printout. Most elementary criteria are applied to each cluster description (i.e., to each complex) and the sum over all cluster descriptions is used to evaluate an entire clustering. The number of exceptional events criterion is an exception and is applied only to the entire clustering. When applied to individual complexes, this criterion produces a score of 0 since the exceptional events list is a property of a clustering, not an individual cluster.

3.4. Domains table

The DOMAINS table is used to define variable domains that are likely to apply to several variables. Variables may be defined entirely with a VARIABLES table (described next) or with the help of the DOMAINS table. When many variables have exactly the same domain type and number of values, it is more convenient to define the domain first, and then reference it in the VARIABLES table. A simple domain definition table is shown below.

```
DOMAINS
 NAME    TYPE    LEVELS
 <c>     <c>      <i>
 COLOR   NOM       4
 SIZE    LIN       3
```

NAME	NAME specifies the name of the domain. In the sample table above, two different domains are defined: color and size.
TYPE	TYPE specifies the type of the domain. Only the first letter of the word given is significant: N means nominal, L means linear, S means structured. See the discussion of TYPE below for further information.
LEVELS	Levels gives the number of distinct values in the domain. If the number of levels is j, the domain consists of the integer values from 0 to j-1.

An optional column COST may be used to indicate the cost of a variable having this domain.

3.5. Variables table

The VARIABLES table is used to describe the attribute variables used to describe each event. A simple variables table is shown below.

VARIABLES

#	TYPE	LEVELS
<i>	<c>	<i>
1	NOM	4
2	LIN	3
3	LIN	5

\# — The # column gives the variable ordinal. The above table defines the variables X_1, X_2, and X_3. The ordinal values must be in ascending order, starting with 1.

TYPE — TYPE specifies the type of the domain of each variable. Only the first letter of the word given under TYPE is significant. Nominal (or just N) denotes an unordered qualitative domain. Linear (or just L) denotes an ordered quantitative domain. Structured (or just S) denotes a tree-ordered (or graph-ordered) domain. If the Structured domain type is specified, a -STRUCTURE table must be entered, see below.

LEVELS — LEVELS gives the number of distinct values in the domain. If the number of levels is j, CLUSTER/2 requires that the data values be integers from 0 up to j-1. The event descriptions are entered as a vector of integers, one integer for each variable.

In printing the descriptions for each cluster, CLUSTER/2 refers to the variables as X1, X2, X3, etc. The name of the variable in the report can be changed by adding a NAME column to the VARIABLES table. The following table defines the variables SHAPE, SIZE, WEIGHT.

VARIABLES

#	TYPE	LEVELS	NAME
<i>	<c>	<i>	<c>
1	NOM	4	SHAPE
2	LIN	3	SIZE
3	LIN	5	WEIGHT

Another optional column is COST which can accept any real value. COST is used in calculating the score for elementary criterion number 7, the "simplicity" of a cluster description. If the COST column is not entered, all variables have the default cost of 1.

To indicate that a variable has a domain identical to a predefined domain (defined via the DOMAINS table) the name of the variable is entered in a special way (this requires that the NAME column be present in the VARIABLES table). The name is entered as <variablename>-<domainname>, i.e., as one character string composed of the desired variable name, a hyphen, and a domain name that was defined in the DOMAINS table. When this is done, the TYPE, LEVELS, and COST for the variable are taken from the domain definition regardless of whether specified in the VARIABLES table. If any defined domain has associated -NAMES, -ONAMES, or -STRUCTURE tables (all described below) then these tables must precede the VARIABLES table.

3.6. Events table

The EVENTS table holds the attribute vectors for all objects or situations being studied. Each object or situation is defined by one row of integer values (subsequent sections tell how to handle symbolic attribute values). Each attribute value must be placed in its corresponding column and must be in the range from 0 up to LEVELS-1, where LEVELS denotes the number of levels for that attribute as specified in the VARIABLES table. The headings for the EVENTS table are the names of the variables. If the variables are defined without the NAME column option, the variable names are of the form Xn, where n is the variable ordinal (e.g., X1). If the variable is defined with the NAME column, the given name is used without any hyphen or domain name part (e.g., SHAPE). The following EVENTS table shows the form used with variables that are not given a NAME.

EVENTS

#	X1	X2	X3
<i>	<i>	<i>	<i>
1	0	1	1
2	2	1	0
3	1	3	3
4	1	2	3

The same table must be entered in the following form if the variables are given names. The heading keywords must match the values from the NAME column of the VARIABLES table (they need not be in the same order).

EVENTS

#	SHAPE	SIZE	WEIGHT
<i>	<i>	<i>	<i>
1	0	1	1
2	2	1	0
3	1	3	3
4	1	2	3

The # column gives the event number. The values in this column must be in sequence and must begin with the value 1.

When working with events described by very long attribute vectors, the number of columns in the event table (one column per attribute) may make the tables unmanageably wide. CLUSTER/2 permits such tables to be split into left and right halves (or left, middle, right, etc.). To make use of this feature, the EVENTS table is specified several times, with different column-headings (for different variables) used in each EVENTS table. (Each table contains the # column). All of the separate EVENTS tables must have the same number of rows and the rows must be in exactly the same order. CLUSTER/2 collects the values from the ith row of each table into the attribute description of the ith event.

The optional column WT my be entered to provide a weight for each event. The weight may be used to indicate the relative frequency of observation of the event. The use of weighted events alters the calculation of sparseness. Each observed event counts according to its weight, so a complex with an "area" of 4 and covering 2 observed events receives a sparseness of 4 minus the combined weights of the covered events. With large weights, the sparseness calculation may produce negative values (this causes no problems). Negative sparseness should be interpreted as a measure of fit something like weight density. The more weight of covered events, the more negative the sparseness.

3.7. Names table

A -NAMES table may be entered for any variable or domain definition. The specific part of the table name is the name of the variable (either Xn form or as given in the NAMES column of the VARIABLES table) or domain. The -NAMES table specifies a symbolic name to be used in lieu of the integer variable value in *both* the reported cluster descriptions *and the input of the event data (i.e., values for this variable in an* EVENTS table must be symbolic rather than integer). The -NAMES table is illustrated below.

SHAPE-NAMES

VALUE	NAME
<i>	<c>
0	SQUARE
1	RECTANGLE
2	CIRCLE
3	OVAL

For linearly ordered variables and domains, the VALUE integer establishes the ordering of symbols within the domain. If a single value denotes a range of observed features (e.g., the value 3 denotes a temperature in the range from 68 to 72) the name should be written "lowname..highname" (e.g., "68..72"). The use of the .. range indicator allows CLUSTER/2 to combine names appropriately when reporting an interval value of a linearly ordered variable.

3.8. Onames table

The -ONAMES table may be entered in lieu of a -NAMES table. The -ONAMES table specifies output-only names for the integer values of the domain of the variable. If the input data is numerical, an -ONAMES table will allow the output to be symbolic while the input remains numerical. (Using the -NAMES table requires the input to be symbolic rather than numerical.) An -ONAMES table contains the same columns as a -NAMES table.

3.9. Structure table

Variables that are defined as Structured TYPE have a tree-structured (or graph-structured) domain. The LEVELS value given in the VARIABLES table (or DOMAINS table) refers to the number of discrete leaf values in the tree (or graph). Higher-order node values (generalized values of the variable) are defined by the -STRUCTURE table. The specific part of the table name is the name of the variable (either the Xn form or as given in the NAME column of the VARIABLES table).

The -STRUCTURE table takes two forms, depending on the use of an associated -NAMES table. If no -NAMES table is used for the variable, the -STRUCTURE table looks like this:

SHAPE-STRUCTURE

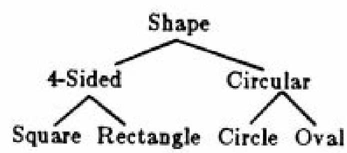
VALUE	SUBVALUE	(SUBVALUE)	(SUBVALUE)	. . .
<i>	<i>	<i>	<i>	
4	2	3		
5	0	1		

The VALUE columns and the first SUBVALUE column are required. Additional SUBVALUE columns may be used by including SUBVALUE several times in the column-heading list. (In the above illustration the parenthesis around SUBVALUE denotes that the column is optional. The parenthesis is not used in the input file.) For each row in the -STRUCTURE table, the value in the VALUE column is defined as the parent of the values in the SUBVALUE columns. Thus each row of the -STRUCTURE table is defining one (or more) links in the tree-structured domain. Any nodes or leaf values that have no explicitly defined parent are implicitly parented by the root of the tree. The values that occur under

VALUE must be greater than any regular (leaf) value for the variable. If the LEVELS column of the VARIABLES table specifies j levels, the leaf values in the domain are denoted by values from 0 to j-1, and the values j, j+1, j+2, ... could appear in the VALUE column of the -STRUCTURE table to denote generalized domain values. In the example above, the variable SHAPE has four levels (values 0 to 3). Two new values (4 and 5) are defined as generalizations of values 2 and 3 and values 0 and 1, respectively.

If a -NAMES table is given for the variable (the -NAMES table must precede the -STRUCTURE table for the same variable) the following form of the -STRUCTURE table is to be used.

```
SHAPE-STRUCTURE
NAME       SUBNAME    SUBNAME
<c>         <c>         <c>
4-SIDED    SQUARE     RECTANGLE
CIRCULAR   CIRCLE       OVAL
```



The tree on the right illustrates the semantics of this -STRUCTURE table. This latter form of the -STRUCTURE table has all the options of the previous form, but with symbolic (character data) names for all values. The names given to the defined generalized domain values must be unique among all names (including leaf names) in the same domain.

If a generalized value must be the parent to a great number of subvalues (or subnames) the table may become unmanageable. In such cases it is permitted to repeat the VALUE (or NAME) entry on adjacent lines, giving part of the subvalue (subname) list on each line. CLUSTER/2 merges the definitions together into one specification of a generalized value for all of the subvalues mentioned.

4. A sample input file

The following sample input file makes use of the relational table forms described above. Some comments about the interpretation of this example are given below.

```
MAINTITLE   "MICROCOMPUTERS"

PARAMETERS
   K   TRACE   CRITERION  NIDSPEED  COVERTYPE
   2   ON         DS        SLOW     DISJOINT
   $   OFF        FC        FAST     HIERARCHICAL
```

DS-CRITERION

#	CRITERION	TOLERANCE
1	DIS	0.3
2	COM	0.0
3	PSPARS	0.0

FC-CRITERION

#	CRITERION	TOLERANCE
1	PSPARS	0.3
2	SIM	0.0

VARIABLES

#	NAME	TYPE	LEVELS
1	MP	STR	13
2	RAM	LIN	4
3	ROM	LIN	7
4	DISP	STR	4
5	KEYS	LIN	5

MP-NAMES

VALUE	NAME
0	8080A
1	6502
2	Z80
3	1802
4	6502C
5	6502A
6	68000
7	6800
8	6805
9	6809
10	8048
11	Z8000
12	HP

MP-STRUCTURE

NAME	SUBNAME	SUBNAME	SUBNAME	SUBNAME
6502X	6502	6502C	6502A	$
8080X	8080A	Z80	8048	$
8BIT	6502X	8080X	1802	6800
8BIT	6805	6809	$	$
16BIT	68000	Z8000	HP	$

RAM-ONAMES

VALUE	NAME
0	16K
1	32K
2	48K
3	64K

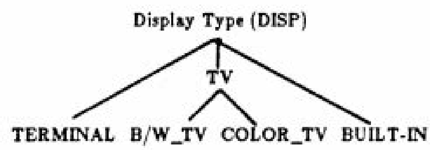
```
ROM-ONAMES
VALUE  NAME
  0     1K
  1     4K
  2     8K
  3     10K
  4     11K..16K
  5     26K
  6     80K

DISP-ONAMES
VALUE  NAME
  0     TERMINAL
  1     "B/W_TV"
  2     "COLOR_TV"
  3     BUILT-IN

DISP-STRUCTURE
NAME           VALUE  SUBVALUE  SUBVALUE
 TV              4       1         2

KEYS-ONAMES
VALUE  NAME
  0     52
  1     53..56
  2     57..63
  3     64..73
  4     92


EVENTS
 #    MP       RAM   ROM   DISP   KEYS
 1    6502      2     3     2      0
 2    6502      2     3     2      2
 3    6502A     1     4     2      3
 4    Z80       2     1     1      2
 5    8080A     3     0     3      3
 6    Z80       3     2     3      3
 7    HP        1     6     3      4
 8    Z80       3     2     0      2
 9    6502      1     3     1      1
10    6502C     2     3     1      1
11    Z80       2     4     1      1
12    Z80       2     4     3      3
<end of file>
```



The tables used in the above file are PARAMETERS, DS-CRITERION, FC-CRITERION, VARIABLES, MP-NAMES, MP-STRUCTURE, RAM-NAMES, ROM-NAMES, DISP-NAMES, DISP-STRUCTURE, KEYS-NAMES, and EVENTS. The entire clustering algorithm is to be performed two times (because there are two rows in the PARAMETERS table). The two different user-defined criterion names (DS and

FC) have -CRITERION tables which define the elementary criteria used. The variables MP, RAM, ROM, DISP, and KEYS have names associated with the integer data values. There is a -NAMES table for each variable. The variables MP and DISP have structured domains. There is a -STRUCTURE table for each of these two variables.

The table MP-STRUCTURE presents a graph-structured domain in which the generalized values 6502x (members of the 6502 family of microprocessors), 8080x, 8bit, and 16bit are defined. The inset shows the graph-structured domain of the DISP variable as defined by the DISP-STRUCTURE table.

5. Program output corresponding to sample input file

The beginning of the output listing shows the PARAMETERS table and the two -CRITERION tables. Since the PARAMETERS table contains two data rows, the CLUSTER/2 program will be invoked twice, once for k=2 for a "flat" clustering, and once for a hierarchical clustering in which the best k in the range 2 to 4 will be determined (using the default beta value of 3.0). Parts of the remaining listing that are of particular interest have been marked with circled numbers on the right-hand side.

Location "1" in the listing marks the beginning of the first clustering, which is identified as "experiment 1." The criterion applied is "ds" as defined by the DS-CRITERION table. The number of clusters is set (by the user) to 2. Because the trace mode is on (see first data row of the PARAMETERS table), each clustering that is formed is described in the listing under the heading "intermediate results...". The column "iter" gives the clustering iteration number while the column "cplx#" gives the number of each cluster formed (here, only two clusters will be formed). Under the column "vl-rule" is the description of the cluster using the names specified in the various -NAMES tables. In iteration 1, the first cluster consists of microcomputer systems that use an 8bit microprocessor (mp), that have 32K to 48K of ram, that have 10K to 16K of rom, that use a TV for a display device, and that have 52 to 56 keys on their keyboards. The column "events covered" indicates that this first cluster covers input events (rows in the EVENTS table) numbered 1, 9, 10, and 11. The "costs" columns show the scores obtained by each cluster on the elementary criteria in the criterion of clustering quality defined by the -CRITERION table for the "ds" criterion. A "totals" row on the listing shows the total scores for the entire clustering.

When clustering quality improvement ceases, CLUSTER/2 reports the best clusterings obtained. For the "experiment 1" this report is at location "2" in the listing. Because the default value for BASE is used (see PARAMETERS table), two alternative clusterings are shown, with the best one listed first. Since the first experiment specified k=2 and a disjoint (flat) cover type, the processing for experiment 1 is complete.

Location "3" marks the beginning of "experiment 2" which uses a different clustering quality criterion (called "fc" by the user). In experiment 2, a hierarchical clustering will be generated, with k determined by the program upon considering clusterings with k ranging from 2 to 4. The trace option is off for experiment 2, so only the best clusterings appear on the listing. (See the second line of the PARAMETERS table for these parameter settings).

The microcomputers are first clustered into two categories. Then, at locations "4" and "5" in the listing, they are clustered into three and four categories, respectively. At location "6" in the listing, the best value of k is determined by comparing the "S-scores" for each of the clusterings for k=2 to k=4. The S-score is the sparseness multiplied by k raised to the beta power. In this example, beta has the default value of 3.0. For experiment 2, the best top-level clustering in the cluster hierarchy is the one obtained with k=3. Its description is shown at location "6".

Of the three clusters formed at the top level of the hierarchy, one contains just a single event and undergoes no further partitioning. The other clusters contain 5 and 6 events, respectively. The events in the first cluster of the top level of the hierarchy are further clustered, leading to the final result shown at location "7". The best value for k for this second-level clustering was 3. The clusters are too small (number of events not above parameter minsize) to be partitioned any further.

Following location "7", the output is shown for the clustering of the second first-level cluster. The final results for this second-level clustering is shown at location "8" in the listing. With beta set to 3.0, the best clustering is with k=4.

output from conjunctive conceptual clustering program cluster/2 last upgrade: 11/10/83

parameters

mink	maxk	trace	h1	h2	h3	initmethod	nidspeed	covertype	criterion	base	probe	beta	maxheight	minsize
2	2	on	3	2	3	random	slow	disjoint	ds	2	2	3.0	99	4
2	4	off	3	2	3	random	fast	hierarchial	fc	2	2	3.0	99	4

ds-criterion

#	criterion	tolerance
1	dis	0.30
2	com	0.00
3	pspars	0.00

fc-criterion

#	criterion	tolerance
1	except	0.00
2	pspars	0.30
3	sim	0.00

variables

#	type	levels	cost	name
1	structured	13	1	mp
2	linear	4	1	ram
3	linear	7	1	rom
4	structured	4	1	disp
5	linear	5	1	keys

mp-names

value	name
0	8080a
1	6502
2	z80
3	1802
4	6502c
5	6502a
6	68000
7	6800
8	6805
9	6809
10	8048
11	z8000
12	hp

mp-structure

value	name	subname	subname	subname
13	6502x	6502	6502c	6502a
14	8080x	8080a	z80	8048
15	8bit	8080a	6502	z80
15	8bit	1802	6502c	6502a
15	8bit	6800	6805	6809
15	8bit	8048	$	$
16	16bit	68000	z8000	hp

ram-onames

value	name
0	16k
1	32k
2	48k
3	64k

rom-onames

value	name
0	1k
1	4k
2	8k
3	10k
4	11k..16k
5	26k
6	80k

disp-onames

value	name
0	terminal
1	b/w_tv
2	color_tv
3	built-in

```
 disp-structure
value  name              subname      subname
  4    tv                b/w_tv       color_tv


 keys-onames
value   name
  0      52
  1      53..56
  2      57..63
  3      64..73
  4      92


 events
  #  mp      ram  rom        disp       keys
  1   6502    48k  10k         color_tv  52
  2   6502    48k  10k         color_tv  57..63
  3   6502a   32k  11k..16k    color_tv  64..73
  4   z80     48k  4k          b/w_tv    57..63
  5   8080a   64k  1k          built-in  64..73
  6   z80     64k  8k          built-in  64..73
  7   hp      32k  80k         built-in  92
  8   z80     64k  8k          terminal  57..63
  9   6502    32k  10k         b/w_tv    53..56
 10   6502c   48k  10k         b/w_tv    53..56
 11   z80     48k  11k..16k    b/w_tv    53..56
 12   z80     48k  11k..16k    built-in  64..73

                                                    microcomputers
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

experiment 1: k=2, criterion=ds
                                                                                                                     1
intermediate results...
iter/cplx#  vl-rule                                     seed ------------costs------------- events covered
                                                                   dis        com     pspars
  1    1  [mp=8bit][ram=32k..48k][rom=10k..16k]          1         4         -5        156 1,9,10,11
          [disp=tv][keys=52..56]
  1    2  [mp<>1802][ram=32k..64k][keys=57..92]          2         4         -3       3016 2,3,4,5,6,7,8,12
  1   totals                                                       8         -8       3172
    (750 ms)

  2    1  [mp<>8080x][ram=32k..48k][rom=10k..80k]        9         4         -4       1194 1,2,3,7,9,10
          [disp<>terminal]
  2    2  [mp=8080x][ram=48k..64k][rom=1k..16k]         12         4         -5        264 4,5,6,8,11,12
          [disp<>color_tv][keys=53..73]
  2   totals                                                       8         -9       1458
    (2600 ms)

  3    1  [mp<>8080x][ram=32k..48k][rom=10k..80k]        2         4         -4       1194 1,2,3,7,9,10
          [disp<>terminal]
  3    2  [mp=8080x][ram=48k..64k][rom=1k..16k]          6         4         -5        264 4,5,6,8,11,12
          [disp<>color_tv][keys=53..73]
  3   totals                                                       8         -9       1458
    (2300 ms)

  4    1  [mp=hp][ram=32k][rom=80k]                      7         2         -5          0 7
          [disp=built-in][keys=92]
  4    2  [mp=8bit][ram=32k..64k][rom=1k..16k]          11         2         -4       2389 1,2,3,4,5,6,8,9,10,11,12
          [keys=52..73]
  4   totals                                                       4         -9       2389
    (2217 ms)

  5    1  [mp=hp][ram=32k][rom=80k]                      7         2         -5          0 7
          [disp=built-in][keys=92]
  5    2  [mp=8bit][ram=32k..64k][rom=1k..16k]           4         2         -4       2389 1,2,3,4,5,6,8,9,10,11,12
          [keys=52..73]
  5   totals                                                       4         -9       2389
    (2255 ms)

  6    1  [mp=hp][ram=32k][rom=80k]                      7         2         -5          0 7
          [disp=built-in][keys=92]
  6    2  [mp=8bit][ram=32k..64k][rom=1k..16k]           5         2         -4       2389 1,2,3,4,5,6,8,9,10,11,12
          [keys=52..73]
  6   totals                                                       4         -9       2389
    (2367 ms)


the 2 best clusterings follow...    (13117 ms)
                                                                                                                     2
iter/cplx#  vl-rule                                     seed ------------costs------------- events covered
                                                                   dis        com     pspars
  4    1  [mp=hp][ram=32k][rom=80k]                      7         2         -5          0 7
          [disp=built-in][keys=92]
  4    2  [mp=8bit][ram=32k..64k][rom=1k..16k]          11         2         -4       2389 1,2,3,4,5,6,8,9,10,11,12
          [keys=52..73]
  4   totals                                                       4         -9       2389

  2    1  [mp<>8080x][ram=32k..48k][rom=10k..80k]        9         4         -4       1194 1,2,3,7,9,10
          [disp<>terminal]
  2    2  [mp=8080x][ram=48k..64k][rom=1k..16k]         12         4         -5        264 4,5,6,8,11,12
          [disp<>color_tv][keys=53..73]
  2   totals                                                       8         -9       1458

 (22 stars built)

for the best solution above, s=  1.9e+04
```

* *

experiment 2: k=2, criterion=fc 3

the 2 best clusterings follow... (13283 ms)

iter/cplx#		vl-rule	seed	except	pspars	sim	events covered
2	1	[mp<>8080x][ram=32k..48k][rom=10k..80k] [disp<>terminal]	1	0	1194	6	1,2,3,7,9,10
2	2	[mp=8080x][ram=48k..64k][rom=1k..16k] [disp<>color_tv][keys=53..73]	12	0	264	8	4,5,6,8,11,12
2	totals			0	1458	14	
4	1	[mp<>8080x][ram=32k][rom=10k..80k] [disp<>terminal][keys=53..92]	7	0	477	7	3,7,9
4	2	[mp=8bit][ram=48k..64k][rom=1k..16k] [keys=52..73]	11	0	1591	7	1,2,4,5,6,8,10,11,12
4	totals			0	2068	14	

(22 stars built)

for the best solution above, s= 1.2e+04

* *

experiment 2: k=3, criterion=fc 4

the 2 best clusterings follow... (60084 ms)

iter/cplx#		vl-rule	seed	except	pspars	sim	events covered
6	1	[mp=6502x][ram=32k..48k][rom=10k..16k] [disp=tv][keys=52..73]	2	0	91	8	1,2,3,9,10
6	2	[mp=8080x][ram=48k..64k][rom=1k..16k] [disp<>color_tv][keys=53..73]	6	0	264	8	4,5,6,8,11,12
6	3	[mp=hp][ram=32k][rom=80k] [disp=built-in][keys=92]	7	0	0	5	7
6	totals			0	355	21	
5	1	[mp=6502x][ram=32k..48k][rom=10k..16k] [disp=tv][keys=52..73]	9	0	91	8	1,2,3,9,10
5	2	[mp=8080x][ram=48k..64k][rom=1k..16k] [keys=53..73]	6	0	354	7	4,5,6,8,11,12
5	3	[mp=hp][ram=32k][rom=80k] [disp=built-in][keys=92]	7	0	0	5	7
5	totals			0	445	20	

(76 stars built)

for the best solution above, s= 9.6e+03

* *

experiment 2: k=4, criterion=fc 5

the 2 best clusterings follow... (76517 ms)

iter/cplx#		vl-rule	seed	except	pspars	sim	events covered
3	1	[mp=6502x][ram=32k..48k][rom=10k] [disp=tv][keys=52..63]	1	0	32	7	1,2,9,10
3	2	[mp<>8080x][ram=32k][rom=11k..80k] [disp<>terminal][keys=64..92]	3	0	178	7	3,7
3	3	[mp=z80][ram=48k][rom=4k..16k] [disp<>terminal][keys=53..73]	4	0	33	7	4,11,12
3	4	[mp=8080x][ram=64k][rom=1k..8k] [disp<>tv][keys=57..73]	6	0	33	7	5,6,8
3	totals			0	276	28	
2	1	[mp=6502x][ram=48k][rom=10k][disp=tv] [keys=52..63]	1	0	15	6	1,2,10
2	2	[mp<>8080x][ram=32k][rom=10k..80k] [disp<>terminal][keys=53..92]	9	0	477	7	3,7,9
2	3	[mp=z80][ram=48k][rom=4k..16k] [disp<>terminal][keys=53..73]	12	0	33	7	4,11,12
2	4	[mp=8080x][ram=64k][rom=1k..8k] [disp<>tv][keys=57..73]	6	0	33	7	5,6,8
2	totals			0	558	27	

(89 stars built)

for the best solution above, s= 1.8e+04

with beta= 3.00 the best clustering at this level is for k=3 6

iter/cplx#		vl-rule	seed	except	pspars	sim	events covered
6	1	[mp=6502x][ram=32k..48k][rom=10k..16k] [disp=tv][keys=52..73]	2	0	91	8	1,2,3,9,10
6	2	[mp=8080x][ram=48k..64k][rom=1k..16k] [disp<>color_tv][keys=53..73]	6	0	264	8	4,5,6,8,11,12
6	3	[mp=hp][ram=32k][rom=80k] [disp=built-in][keys=92]	7	0	0	5	7
6	totals			0	355	21	

```
beginning classification below hierarchy path 1-0
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

panic finding new seeds
experiment 2: k=2, criterion=fc

panic finding new seeds
premature end of clustering: seeds exhausted

the 2 best clusterings follow...   (6850 ms)

iter/cplx#  vl-rule                                          seed ------------costs------------- events covered
                                                                     except     pspars        sim
  3    1   [mp=6502x][ram=32k..48k][rom=10k]                   2         0         32          7 1,2,9,10
           [disp=tv][keys=52..63]
  3    2   [mp=6502a][ram=32k][rom=11k..16k]                   3         0          0          5 3
           [disp=color_tv][keys=64..73]
  3   totals                                                             0         32         12

  2    1   [mp=6502x][ram=48k][rom=10k][disp=tv]               1         0         15          6 1,2,10
           [keys=52..63]
  2    2   [mp=6502x][ram=32k][rom=10k..16k]                   9         0         34          7 3,9
           [disp=tv][keys=53..73]
  2   totals                                                             0         49         13

 (18 stars built)

for the best solution above, s=  2.6e+02
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

panic finding new seeds
experiment 2: k=3, criterion=fc

panic finding new seeds
premature end of clustering: seeds exhausted

the 2 best clusterings follow...   (6867 ms)

iter/cplx#  vl-rule                                          seed ------------costs------------- events covered
                                                                     except     pspars        sim
  2    1   [mp=6502][ram=48k][rom=10k]                         1         0          1          6 1,2
           [disp=color_tv][keys=52..63]
  2    2   [mp=6502x][ram=32k..48k][rom=10k]                   9         0          4          6 9,10
           [disp=b/w_tv][keys=53..56]
  2    3   [mp=6502a][ram=32k][rom=11k..16k]                   3         0          0          5 3
           [disp=color_tv][keys=64..73]
  2   totals                                                             0          5         17

  1    1   [mp=6502][ram=48k][rom=10k]                         1         0          0          5 1
           [disp=color_tv][keys=52]
  1    2   [mp=6502x][ram=32k..48k][rom=10k]                   2         0         21          7 2,9,10
           [disp=tv][keys=53..63]
  1    3   [mp=6502a][ram=32k][rom=11k..16k]                   3         0          0          5 3
           [disp=color_tv][keys=64..73]
  1   totals                                                             0         21         17

 (21 stars built)

for the best solution above, s=  1.4e+02
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
experiment 2: k=4, criterion=fc

panic finding new seeds
premature end of clustering: seeds exhausted

the 1 best clusterings follow...   (7516 ms)

iter/cplx#  vl-rule                                          seed ------------costs------------- events covered
                                                                     except     pspars        sim
  1    1   [mp=6502][ram=48k][rom=10k]                         1         0          0          5 1
           [disp=color_tv][keys=52]
  1    2   [mp=6502][ram=48k][rom=10k]                         2         0          0          5 2
           [disp=color_tv][keys=57..63]
  1    3   [mp=6502a][ram=32k][rom=11k..16k]                   3         0          0          5 3
           [disp=color_tv][keys=64..73]
  1    4   [mp=6502x][ram=32k..48k][rom=10k]                   9         0          4          6 9,10
           [disp=b/w_tv][keys=53..56]
  1   totals                                                             0          4         21

 (30 stars built)

for the best solution above, s=  2.6e+02

with beta=  3.00 the best clustering at this level is for k=3
```

7

```
iter/cplx#  vl-rule                                    seed ------------costs------------- events covered
                                                            except    pspars        sim
  2    1  [mp=6502][ram=48k][rom=10k]                    1       0         1           6 1,2
          [disp=color_tv][keys=52..63]
  2    2  [mp=6502x][ram=32k..48k][rom=10k]              9       0         4           6 9,10
          [disp=b/w_tv][keys=53..56]
  2    3  [mp=6502a][ram=32k][rom=11k..16k]              3       0         0           5 3
          [disp=color_tv][keys=64..73]
  2   totals                                                    0         5          17

beginning classification below hierarchy path 2-0

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

experiment 2: k=2, criterion=fc


the 1 best clusterings follow...    (6900 ms)

iter/cplx#  vl-rule                                    seed ------------costs------------- events covered
                                                            except    pspars        sim
  1    1  [mp=z80][ram=48k][rom=4k..16k]                 4       0        33           7 4,11,12
          [disp<>terminal][keys=53..73]
  1    2  [mp=8080x][ram=64k][rom=1k..8k]                5       0        33           7 5,6,8
          [disp<>tv][keys=57..73]
  1   totals                                                    0        66          14

 (16 stars built)

for the best solution above, s=  5.3e+02

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

experiment 2: k=3, criterion=fc


the 1 best clusterings follow...    (14200 ms)

iter/cplx#  vl-rule                                    seed ------------costs------------- events covered
                                                            except    pspars        sim
  1    1  [mp=z80][ram=48k][rom=4k..16k]                 4       0         6           7 4,11
          [disp=b/w_tv][keys=53..63]
  1    2  [mp=8080a][ram=64k][rom=1k]                    5       0         0           5 5
          [disp=built-in][keys=64..73]
  1    3  [mp=z80][ram=48k..64k][rom=8k..16k]            6       0        21           8 6,8,12
          [disp<>tv][keys=57..73]
  1   totals                                                    0        27          20

 (31 stars built)

for the best solution above, s=  7.3e+02

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

experiment 2: k=4, criterion=fc

panic finding new seeds

the 2 best clusterings follow...    (18017 ms)

iter/cplx#  vl-rule                                    seed ------------costs------------- events covered
                                                            except    pspars        sim
  2    1  [mp=z80][ram=48k][rom=4k..16k]                 4       0         6           7 4,11
          [disp=b/w_tv][keys=53..63]
  2    2  [mp=8080a][ram=64k][rom=1k]                    5       0         0           5 5
          [disp=built-in][keys=64..73]
  2    3  [mp=z80][ram=48k][rom=11k..16k]               12       0         0           5 12
          [disp=built-in][keys=64..73]
  2    4  [mp=z80][ram=64k][rom=8k][disp<>tv]            8       0         2           6 6,8
          [keys=57..73]
  2   totals                                                    0         8          23

  1    1  [mp=z80][ram=48k][rom=4k..16k]                 4       0         6           7 4,11
          [disp=b/w_tv][keys=53..63]
  1    2  [mp=8080a][ram=64k][rom=1k]                    5       0         0           5 5
          [disp=built-in][keys=64..73]
  1    3  [mp=z80][ram=48k..64k][rom=8k..16k]            6       0         4           7 6,12
          [disp=built-in][keys=64..73]
  1    4  [mp=z80][ram=64k][rom=8k]                      8       0         0           5 8
          [disp=terminal][keys=57..63]
  1   totals                                                    0        10          24

 (40 stars built)

for the best solution above, s=  5.1e+02


with beta=  3.00 the best clustering at this level is for k=4                                        8

iter/cplx#  vl-rule                                    seed ------------costs------------- events covered
                                                            except    pspars        sim
  2    1  [mp=z80][ram=48k][rom=4k..16k]                 4       0         6           7 4,11
          [disp=b/w_tv][keys=53..63]
  2    2  [mp=8080a][ram=64k][rom=1k]                    5       0         0           5 5
          [disp=built-in][keys=64..73]
  2    3  [mp=z80][ram=48k][rom=11k..16k]               12       0         0           5 12
          [disp=built-in][keys=64..73]
  2    4  [mp=z80][ram=64k][rom=8k][disp<>tv]            8       0         2           6 6,8
          [keys=57..73]
  2   totals                                                    0         8          23
```

REFERENCES

[1] Michalski, R. S., Stepp, R. E., Diday, E., A RECENT ADVANCE IN DATA ANALYSIS: Clustering Objects into Classes Characterized by Conjunctive Concepts, Invited chapter in *Progress in Pattern Recognition,* Vol. 1, L. Kanal and A. Rosenfeld, Eds., pp. 33-55, 1981.

[2] Michalski, R. S., Stepp, R., Revealing conceptual structure in data by inductive inference, in Machine Intelligence 10, eds. J. E. Hayes, D. Michie, Y.-H. Pao, Ellis Horwood, Chichester, Halsted Press (John Wiley), New York, 1981.

[3] Michalski, R. S., Stepp, R., An application of AI techniques to structuring objects into an optimal conceptual hierarchy, Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, Canada, August 24-28, pp. 460-465, 1981.

[4] Michalski, R. S., Stepp, R. E., AUTOMATED CONSTRUCTION OF CLASSIFICATIONS: Conceptual Clustering versus Numerical Taxonomy, to appear in IEEE Pattern Analysis and Machine Intelligence, 1983.

[5] Michalski, R. S., KNOWLEDGE ACQUISITION THROUGH CONCEPTUAL CLUSTERING: A theoretical framework and an algorithm for partitioning data into conjunctive concepts, A Special Issue on Knowledge Acquisition And Induction, International Journal of Policy Analysis and Information Systems, Vol. 4, No. 3, pp. 219-244, 1980.

Appendix I: CLUSTER/2 Program Listing

Index to procedures and functions

```
program cluster2(input,output);

(* this is the conceptual clustering program CLUSTER/2.  input is in the form
   of relational tables.  output is in a report form, with a maximum line
   length specified by the lnelength constant.

   a complete description of the relational table input format is in
   "a description and user's guide for CLUSTER/2, a program for conjunctive
   conceptual clustering" by robert stepp *)

(* globol definitions *)

const
      verdate      = '11/08/83';           (* date of last change *)
      setsize      = 58;                  (* maximum number of levels *)
      treesize     = 255;                 (* max nodes in search tree *)
      alfasize     = 20;                  (* 10 for cyber implementation *)
      lalfasize    = 80;                  (* long alfa size in chars *)
      maxevents    = 590;                 (* max number of events *)
      numevsets    = 10;                  (* set to maxevents div 59 *)
      maxk         = 8;                   (* max number of clusters *)
      maxvars      = 80;                  (* max number of variables *)
      maxcrit      = 9;                   (* max criterium number *)
      maxlevels    = 59;                  (* max levels in one variable *)
      maxbase      = 8;                   (* max alternative solutions *)
      lnelength    = 130;                 (* output line length *)

type
      alfa         = packed array[1..alfasize] of char;
      alfa10       = packed array[1..10] of char;
      lalfa        = packed array[1..lalfasize] of char;
      mopcd        = (clr,all,union,diff,inter);(* bit op codes *)

      scops        = (ne,le,eq,ge);      (* op codes for function scomp *)

      initmethod   = (random,ormeth);    (* initial seed selection *)
                                         (* random: pick k events at random *)
                                         (* ormeth: find k optimal reps *)

      disjspeed    = (slow,fast);        (* disjointness methods *)
                                         (* slow: use complete search *)
                                         (* fast: use abbreviated search *)

      covertype    = (disjoint,intersecting,hierarchical);
                                         (* disjoint: flat, non-overlapping *)
                                         (* intersecting: flat, overlapping *)
                                         (* hierarchical: disjoint at each level*)

      domaintype   = (nominal, linear, cyclic, structured);
                                         (* nominal: nominal scale *)
                                         (* linear: partial ordered scale *)
                                         (* cyclic: linear with wrap-around *)
                                         (* structured: has generalization tree *)

      ccode        = (ind, c1c2, c2c1);(* code returned by cmpcov *)
                                         (* ind: c1 and c2 are independent *)
                                         (* c1c2: c1 is covered by c2 *)
```



```
                                         (* c2c1: c2 is covered by c1 *)

      seedtype     = (central, distant);(* seed selection type *)
                                         (* central:select most central events *
                                         (* distant:select most distant events *

      conversion   = (noval,charval,intval,realval);
                                         (* noval: dont input anything *)
                                         (* charval: input string *)
                                         (* intval: input integer *)
                                         (* realval: input floating point *)

      (* the following codes are used to maintain use counts for procedures *)
      stattyp      = (q00,qcd,qcl,qcr,qcv,qed,qfx,qgz,qit,qnd,qnl,qnz, qre,qr
                      qrs,qrn,qrv,qrw,qsp,qsq,qsr,qss,qst,qtm,qtr, qdt,qcg,qtp,
                      qbc,qsd,qgh,qcq,qsv,qlast);(* stat counters *)

      dbugset      = set of stattyp; (* procedure ids that write trace data *)

      cftypes      = (cfchanged, cfmarknid, cfintree, cffree);
                      (* cfchanged - complex shape changed *)
                      (* cfmarknid - mark used by nid *)
                      (* cfintree - this complex in search tree *)
                      (* cffree - tells if complex is being used *)

      comfigs      = set of cftypes;    (* complex special indicators *)

      varidx       = 1..maxvars;        (* index by variable number *)

      treeidx      = 0..treesize;       (* index by tree node *)

      refval       = 0..setsize;        (* range of selector values *)

      selmapunit   = set of refval;     (* indicates covered events *)

      evtmap       = array[0..numevsets] of selmapunit;
                      (* map of covered events *)
                      (* note: the bit map of covered events is broken into
                         several set parts to permit running on the cyber
                         which has a short setsize constraint *)

      pseedmap     = ^seedmap;

      seedmap      = record
         next          : pseedmap;        (* to next seed map *)
         seeds         : evtmap;          (* map of one seed set *)
                      end;

      pcomplex     = ^complex;

      complex      = record             (* storage for vl1 complex *)
         next          : pcomplex;      (* pointer to next complex *)
         cflags        : comfigs;       (* misc indicators *)
         area          : real;          (* area of the complex *)
         cost          : integer;       (* cost or weight of complex *)
         critvals      : array[0..maxcrit] of real;(* vals of crits *)
         map           : evtmap;        (* bit map of covered events *)
```



```
         selectors     : array[varidx] of selmapunit; (* selectors *)
                      end;

(* note on internal representation of a complex:

   the above record type contains and array of selmapunits.  each selmapunit
   is one selector in the complex.  (the name selmapunit comes from the
   fact that this same data type is a selector-unit and a map-unit, for
   compatibility of other primitive functions.)  each selmapunit (each
   selector) is a set of values, often called the reference list of the
   selector.  internally, all values are in the range of 0 to 'setsize'
   where 'setsize' is a constant normally set to 58.  each complex may
   contain up to 'maxvars' number of selmapunits (selectors).

   For example, the complex [x1=2][x2=3..4][x3=1 v 4] is stored internally
   in the first three elements of the array 'selectors' in the above record
   type.  in pascal notation, these three sets are [2], [3,4], and [1,4].
   The other sets (that are elements of 'selectors') are not used.

   each complex also contains a bit-map of the event subset that the complex
   covers.  if a complex covers 8 events, then there will be 8 numbers (of
   events) in the set called the 'evtmap' of the complex.  since the pascal
   sets contain only up to 'setsize' different values (not enough to handle
   the quantity of events desired), each 'evtmap' is actually composed of
   a small array of 'selmapunits' which are logically ganged together to act
   as one large set.  the primitives 'mcomp', 'mop', 'mcard' are provided to
   manipulate these 'map' data aggregates (note the primitives all start with
   'm').

   the 'area' of a complex is the number of points in the event space it
   covers.  the 'cost' field is used in complexes that represent single
   input events.  'cost' in the record comes from the 'wt' column in the
   events table.

   each complex also carries the scores on the lef criteria in the vector
   called 'critvals'.  only those scores for which the criteria is active
   in the lef are evaluated and given a value.
*)

      pcplxlist    = ^cplxlist;

      cplxlist     = record                (* master complex list *)
         next          : pcplxlist;        (* next cplxlist *)
         cplx          : pcomplex;         (* pointer to complex *)
                     end;

      pcover       = ^cover;

      cover        = packed record(* represents clstrs covering complexes *)
         clstrs        : 0..maxk;(* number of clusters this cover *)
         critvals      : array[0..maxcrit] of real;  (* cover costs *)
         pcluster      : packed array[1..maxk] of pcomplex; (* the complexes *)
         exception     : evtmap;         (* exceptional event list *)
         pseed         : packed array[1..maxk] of pcomplex;
                          (* pointers to new seeds for the next clusters *)
         pathdata      : packed array[1..maxk] of 0..maxk;
                          (* path through search tree *)
```



```
         iteration     : integer;       (* iteration on which produced *)
         rank          : integer;       (* hierarchy level for this cover *)
         subcovers     : packed array[1..maxk] of pcover;
                          (* pointers to next hierarchy level *)
         parent        : pcover; (* pointer to parent in hierarchy *)
         pcnum         : 0..maxk; (* the cluster number 1..k *)
                     end;

      pstruct      = ^struct;

      struct       = record           (* units used to build structure trees *)
         next          : pstruct;     (* pointer to next structure entry *)
         nodedefn      : selmapunit;  (* structure node definition *)
         name          : alfa;        (* name of this node *)
         valu          : integer;     (* internal value of this node *)
(* note on structure tree data:

   each internal node in the tree structure of a domain is a generalization of
   certain underlying leaf values in the domain.  the struct data type is usec
   to hold the definitions of internal nodes.  leaf nodes are handled as
   ordinary domain values using other (common) data structures.

   for each structured domain or variable, there is a linked list of internal
   node definitions, each being one instance of the above record type.  each
   such node has a 'valu' which is unique among all internal nodes and leaf
   values.  if the domain has 10 levels (leaf values 0..9) then the smallest
   'valu' for an internal node is 10.  'name' is the name of the value which :
   used in the print procedures whenever a variable takes on this generalized
   value.  'nodedefn' is a 'selmapunit' like a selector.  in particular, it i:
   exactly the equivalent selector for the internal node specified in terms o:
   the leaf values that lie under it in the structure tree of the domain.
   note that this approach purposely does not capture the identity of underly:
   internal nodes, but only the underlying leaf values.
*)
                     end;

      pname        = ^name;

      name         = record             (* units used to hold value names *)
         names         : array[1..maxlevels] of alfa;
                          (* names given to numeric values *)
                     end;

      pdomain      = ^domain;

      domain       = packed record  (* unit that defines a domain *)
         name          : alfa;          (* name of domain *)
         cost          : integer;       (* cost of this domain type *)
         domain        : pdomain;       (* pointer to next domain *)
         structlist    : pstruct;       (* pointer to 1st struc element *)
         dtype         : domaintype;    (* type of domain *)
         maxvalue      : refval ;       (* number of levels *)
         inames        : pname;         (* pointer to input names table *)
         onames        : pname;         (* pointer to output names table *)
         nwidth        : 0..alfasize;   (* length of longest name *)
                     end;
```

```
     variables    = array[varidx] of pdomain; (* variables defn data *)

     pcrit        = ^critlist;

     critlist     = record              (* criteria specification *)
        name         : alfa;            (* criteria name *)
        next         : pcrit;           (* pointer to next crit spec. *)
        ccnt         : 0..maxcrit;      (* length of criterion list *)
        clist        : array[0..maxcrit] of integer; (* list of criteria *)
        tlist        : array[0..maxcrit] of real; (* list of tolerances *)
                    end;

     ppara        = ^para;

     para         = record             (* parameters record *)
        next         : ppara;
        mink, maxk   : integer;         (* number of clusters *)
        debug        : dbugset;         (* trace/debug value *)
        h1, h2, h3   : integer;         (* heuristic search parameters *)
        initmthd     : initmethod;      (* initialization method *)
        aidspeed     : disjspeed;       (* speed of aid process *)
        covertp      : covertype;       (* type of covers *)
        critname     : alfa;            (* name of partial star crit *)
        base         : integer;         (* number of base interations *)
        probe        : integer;         (* number of probe iterations *)
        beta         : real;            (* beta value *)
        maxht        : integer;         (* maxheight of hierarchy *)
        minsize      : integer;         (* min cluster size *)
                    end;

     ptitle       = ^title;

     title        = record              (* title information *)
        next         : ptitle;         (* to additional lines *)
        length       : integer;        (* character count *)
        text         : lalfa;           (* title text *)
                    end;

     pstat        = ^stat;

     stat         = record              (* activity statistics *)
        c            : array[stattyp] of integer; (* procedure use counts *)
                    end;

(* the following static records hold static data (much like global data).
   there is only one instance of each static record throughout the life
   of program execution.  the data are place here, rather than in global
   data areas for two reasons:
     1)  the static data found below is private to a few primitive routines
         (each static area is private to a different set of routines).
     2)  the reduction in global data (by placing it in these static records)
         makes separate compilation of program segments easier.
*)

     sstatic      = record
        crit         : critlist;  (* criterion list for star trimming *)
```



```
        starcount    : integer;   (* count of stars produced *)
        clusadj      : integer;
        (* cluster number to adjust when trying to gen. unique seeds *)
        usedseeds    : pseedmap;  (* head of list of used seed sets *)
                    end;

     ostatic      = record
        freeclst     : pcomplex;        (* chain of free complexes *)
        numfree      : integer;         (* number of free complexes *)
        numgotten    : integer;         (* number of complexes built *)
        cwork        : pcomplex;        (* temp-work complex *)
        wted         : boolean;         (* weighted events present *)
        eventcol     : integer;         (* events table continuation col *)
                    end;

     mstatic      = record
        h1, h2       : integer;  (* heuristic search parameters *)
                    end;

     alfafld      = alfa;  (* altername type name for alfa *)

(* gblvars segment *)

(* global variable definitions *)

var
     postat       : ^ostatic;
     psstat       : ^sstatic;
     pmstat       : ^mstatic;
     pastat       : pstat;

     titles       : ptitle;            (* list of title text *)

     criteria     : pcrit;             (* list of crit defns *)

     parameters   : ppara;             (* parameters list *)

     events       : pcomplex;          (* list of events *)
     numevents    : integer;           (* number of events *)

     domains      : pdomain;           (* list of domain defns *)

     dbug         : dbugset;           (* trace control variable *)

     prtflags     : set of 'a'..'z';   (* print control flags *)
     nv           : integer;           (* number of variables *)

     clstrs       : integer;            (* number of clusters *)

     pvariables   : ^variables;        (* pointer to variables defns *)

     context      : pcomplex;          (* list of already-selected complexes*)

     level        : integer;           (* current level in search tree *)

     levelmap     : evtmap;            (* union of events covered by context *
```



```
     cvcontext    : pcover;            (* current cover *)

     allevents    : evtmap;            (* map of all events *)

(*$e+ ask for real external names *)
(*$r- without reduce *)

(* forward segment *)

(* operational primitives *)

function newcplx                       (* get a new complex (nil if unavail.)*)
       : pcomplex;
     forward;

function freecplx                      (* free a complex *)
     (c             : pcomplex): pcomplex;
     forward;

function numcplx                       (* returns no. of free complexes *)
       : integer;
     forward;

procedure prt                          (* display a complex *)
     (c             : pcomplex;
      n             : integer);
     forward;

procedure prtcrit                     (* print crit name *)
     (n             : integer;
      var chn       : alfa);
     forward;


procedure prtset                       (* list elements in a selector *)
     (s             : selmapunit;
      dlm           : char);
     forward;

procedure prtrlt                       (* print relational table *)
     (tid           : integer;
      tname         : alfa);
     forward;

procedure statnam
     (i             : stattyp;
      var n         : alfa);
     forward;

procedure reduce                       (* shrink complex to minimum size *)
     (c             : pcomplex);
     forward;

procedure setmap                       (* form map of events covered by c*)
     (c             : pcomplex);
     forward;
```



```
function wmcard
     (var map       : evtmap;
      op            : integer): integer;
      forward;

procedure setlevelmap                  (* calculate map of cluster complexes*)
     (p             : pcomplex);
      forward;

function numsel                        (* count selectors in complex *)
     (c             : pcomplex): integer;
      forward;

function mcard
     (var map       : evtmap): integer;
      forward;

procedure mop
     (var m1        : evtmap;
      op            : mopcd;
      var m2, m3    : evtmap);
      forward;

procedure masgn
     (var m         : evtmap;
      b             : integer);
      forward;

function min
     (e             : integer;
      var m         : evtmap): boolean;
      forward;

function mcomp
     (var m1        : evtmap;
      t             : mcops;
      var m2        : evtmap): boolean;
      forward;

function maplow
     (var m         : evtmap): integer;
      forward;

procedure prtmap
     (var m         : evtmap);
      forward;

procedure prtbmsg
(* this procedure operates only on the cyber.  it displays a message about
   the current processing iteration.  for the cyber, prtbmsg is provided
   by a compass routine.  this procedure is a dummy, used on other systems *)
     (var text      : alfa;
      iwhere        : integer;
      iwait         : integer);
(*    fortran; *)   (* cyber only *)

begin
```

```
      text[i] := ' ';
      iwait := iwhere + iwait;
      end;

function domof
(* this function returns the pointer to the domain record for a domain
   or variable of a given name *)
     (id             : alfa): pdomain;
      forward;

function iabs
     (v              : integer): integer;
      forward;

(* definitions of conceptual primitives *)

function tcover                         (* true if c2 covers c1 *)
     (c1,c2          : pcomplex): boolean;
      forward;

function cmpcov                         (* compare coverage *)
     (c1, c2         : pcomplex): ccode;
      forward;

function reflow                         (* lowest value for given selector *)
     (s              : selmapunit): integer;
      forward;

function refhigh                        (* highest value for given selector *)
     (s              : selmapunit): integer;
      forward;

procedure refunion                      (* c1<- c2 u c3 *)
     (c1, c2, c3     : pcomplex);
      forward;

procedure genrliz                       (* generalize selectors of c *)
     (c              : pcomplex);
      forward;

function intrsct                        (* true if c1 and c2 intersect *)
     (c1, c2         : pcomplex): boolean;
      forward;

procedure extend                        (* extend s1 against s2 for var v *)
     (var rslt       : selmapunit;
      s1, s2         : selmapunit;
      v              : varidx);
      forward;

function refnum                         (* count of elements if reference list *)
     (s              : selmapunit;
      v              : varidx): integer;
      forward;

function reflev                         (* count of values in actual ref set*)
     (s              : selmapunit): integer;
```



```
      forward;

function degisct                        (* count selectors which intersect *)
     (c1, c2         : pcomplex): integer;
      forward;

function bestc                          (* select the best complex *)
     (var head       : pcomplex;
      critl          : critlist;
      cn             : integer): pcomplex;
      forward;

procedure trim                          (* trim complexes to q best *)
     (q              : integer;
      var clist      : pcomplex;
      critl          : critlist);
      forward;

function syndist                        (* compute syntactic distance *)
     (c1, c2         : pcomplex): real;
      forward;


(* major subroutines *)

function star                           (* function to build a star *)
     (iei            : integer;
      cv             : pcover;
      h1, h2         : integer): pcomplex;
      forward;

procedure starcr                       (* set intermediate criteria *)
     (cr             : pcrit);
      forward;

function numstar                        (* get number of stars *)
       : integer;
      forward;

function candist                       (* to choose representative events *)
     (cv             : pcover;
      rep            : seedtype): boolean;
      forward;

procedure critval                       (* returns cost via given crit num *)
     (c              : pcomplex);
      forward;

(* ibmpc segment *)
(* this function returns the cardinality of set s *)

(*function card
      (s: selmapunit): integer;

var
      i, c:  integer;
```



```
begin
      c := 0;
      for i:=0 to setsize do if i in s then c := c+1;
      card := c;
      end;
*)

(* this function returns the time in cpu tenths of seconds *)
(* the ibmpc has no clock -- zero is always returned *)

(*function clock: integer;

begin
      clock := 0;
      end;
*)

procedure strasgn
(* this procedure assigns an alfa string the value of an alfa10 string *)
     (var str1       : alfa;
      str2           : alfa10);

(* this procedure assigns a string of length 10 to an alfa *)

var
      i, j           : integer;

begin
      if alfasize>10 then j:=10 else j:=alfasize;
      for i:=1 to j do str1[i] := str2[i];  i := j;
      while (i<alfasize) do begin
         i := i+1;      str1[i] := ' ';
         end;
      end;
(*$1'cluster/2 major subroutines' *)

function star;                          (* build a star *)
(*   (iei           : integer;
      cv            : pcover;
      h1, h2        : integer): pcomplex;
*)

(* build a star about event e1, staying clear of any other seeds *)

var
      nstar          : pcomplex;        (* new star pointer *)
      ostar          : pcomplex;        (* old star pointer *)
      q, r           : pcomplex;        (* working pointers *)
      iseed          : integer;         (* index to seed *)
      e1, e2         : pcomplex;        (* pointer to current events *)
      i              : integer;
      reference      : selmapunit;      (* selector reference set *)
      numcplxs       : integer;         (* number of complexes in star *)

begin
      with psstat^ do c[qsr] := c[qsr]+1;
```



```
      (* locate event e1 via iei; increment number of stars build *)
      e1 := cv^.pseed[iei];        psstat^.starcount := psstat^.starcount + 1;

      if qsr in dbug then writeln(' star: about event ',maplow(e1^.map):1);

      (* begin with a unit complex *)
      ostar := newcplx;       nstar := nil;
      with ostar^ do begin
         for i := 1 to nv do selectors[i] := [0..pvariables^[i]^.maxvalue];
         end;

      (* start with first seed in against set *)
      iseed := 1;       e2 := cv^.pseed[iseed];
      numcplxs := 0;

      (* compute partial stars until against set is exhausted *)
      while (e2<>nil) do begin

         if iseed<>iei then begin (* skip over the focus of attention seed *)
            (* trim number of complexes from previous partial star *)
            if numcplxs>h1 then trim(h1,ostar,psstat^.crit);

            (* find all previous complexes that cover (intersect) with an
               event in the against set; process each such complex *)
            while (ostar<>nil) do if intrsct(ostar,e2) then begin

               (* for each selector that takes different values in each event
                  (the one being covered and the one from the against set)
                  find the maximal extension of its reference list that covers
                  the event being covered (the focus of attention) and does
                  not cover the against set event *)
               for i := 1 to nv do
               if []=(e1^.selectors[i]*e2^.selectors[i]) then begin
                  extend(reference,e1^.selectors[i], e2^.selectors[i],i);
                  r := newcplx;  (* get a new complex to receive result *)
                  (* if unable to get new complex, trim to free old ones *)
                  if (r=nil) and (nstar<>nil) then begin
                     trim(h1,nstar,psstat^.crit);      r := newcplx;
                     end;
                  if r=nil then writeln('star: no free complexes remained ',
                     'after trimming -- results impaired')
                  else begin
                     (* logically multiply the extended selector and the old
                        complex to form a new one that is placed into the
                        new star complex list whose head is the pointer nstar
                     r^ := ostar^;       numcplxs := numcplxs+1;
                     r^.next := nstar;  nstar := r;
                     r^.selectors[i] := r^.selectors[i] * reference;
                     reduce(r);  (* reduce the new complex *)
                     end;
                  end;   (* for *)

               ostar := freecplx(ostar);   (* free old ostar complex *)
               end(* if intrsct *)

            (* if the old complex did not cover against set event e2, then
               it does not need multiplying and is just copied to nstar *)
```

```
            else begin                       (* just copy from ostar to nstar *)
               q := ostar;       ostar := ostar^.next;        q^.next := nstar;
               numcplxs := numcplxs+1;                           nstar := q;
               end;

            (* prepare for next partial star; nstar becomes ostar *)
            ostar := nstar;        nstar := nil;
            end;

         iseed := iseed + 1;  (* go on to next seed in against set *)

         (* for the first k iterations, the against event is a seed event.
            on the k+1 iteration if nidspeed is fast the against event is the
            first complex in the cover taken from the context list.  if
            nidspeed is slow, nil is used and the iterations stop.  otherwise
            on the k+2 and higher iterations, the against event is the next
            complex on the context list, until the entire list has been used.
            there are at most 2k-1 iterations in that case. *)
         if iseed<=clstrs then e2 := cv^.pseed[iseed]
         else if iseed>clstrs+1 then e2 := e2^.next
         else if parameters^.nidspeed=fast then e2 := context
         else e2 := nil;
         end;

      (* finally, trim to number of complexes requested via h2 *)
      trim(h2,ostar,psstat^.crit);        star := ostar;
      end;(*star*)


procedure starcr;                      (* save star trimming criterion spec. *)
(*    (cr             : pcrit);
*)

begin
      if psstat=nil then new(psstat);
      psstat^.starcount := 0;        psstat^.clusadj := 0;
      psstat^.usedseeds := nil;     psstat^.crit := cr^;
      end;

function numstar;        (* report the number of stars built since last report *)
(*          : integer;
*)

begin
      numstar := psstat^.starcount;        psstat^.starcount := 0;
      end;

function cendist;                (* select new seeds -- central or distant *)
(*    (cv             : pcover;
      rep             : seedtype): boolean;
*)

(* new seeds are selected by measuring the sum of syntactic distance from
   each seed to all others.  central seeds have minimum syntactic distance
   sums.  distant seeds have maximum sums.  the set of k seed events must
   not duplicate any previously used seed set.  the seed set is compared with
   all previously generated seed sets and if there is a match, the function
```



```
   is repeated after one event is temporarily discarded.  if all seeds are
   discarded in one cluster, the message 'panic finding new seeds is output'
   and the attention shifts to another complex in hope of finding an
   alternative event to use.  if none can be found, the return code from
   cendist is false to indicate failure to find any seed set.  when a unique
   seed set can be found, the return code is true. *)

var
      n              : integer;          (* index to cluster worked on *)
      attempts       : integer;          (* number of attempts to find seed set *
      i, j, k, m     : integer;
      bi             : integer;           (* index to best seed *)
      bestdist, dist: real;               (* syntastic distance scored *)
      ci             : pcomplex;          (* pointer to complex worked on *)
      emap           : evtmap;            (* event map for complex *)
      seedset        : evtmap;            (* a set of seeds *)
      sum            : real;
      levents      : array[0..maxevents] of pcomplex; (* local event list *)
      counter        : array[0..maxevents] of integer; (* value freq. counts *)
      sdist          : array[0..maxevents] of real; (* syntactic distances *)
      lincent        : array[1..maxvars] of real; (* lenear centers *)
      nomcent        : array[1..maxvars] of selmapunit; (* nominal centers *)
      goodseeds      : boolean;           (* seed set is good (unique) *)
      panic          : boolean;           (* trouble finding seed set *)
      sptr           : pseedmap;

begin
      with psstat^ do c[qcd] := c[qcd]+1;
      goodseeds := false;        attempts:=0;        panic := false;

      (* repeat until good seed set or paternimal panic *)
      repeat
         attempts := attempts+1;

         (* clear seed set *)
         mop(seedset,clr,seedset,seedset);

         (* increment cluster index used to select cluster to work on
            when trying to correct a duplicate seed set problem *)
         psstat^.clusadj := psstat^.clusadj + 1;
         if psstat^.clusadj>clstrs then psstat^.clusadj := 1;

         if qcd in dbug then writeln(' cendist: clusadj=',psstat^.clusadj:1);

         (* select best seed from all complexes *)
         n := psstat^.clusadj;
         repeat
            n := n+1;
            if n>clstrs then n:=1;

            (* be sure complex has an event map *)
            if mcard(cv^.pcluster[n]^.map)=0 then setmap(cv^.pcluster[n]);
            emap := cv^.pcluster[n]^.map;        ci := events;        m := -1;

            (* scan through all events, placing pointers to events covered
               by this complex into a local events list.  m is list length *)
            while (ci<>nil) do begin
```



```
               if mcomp(ci^.map,le,emap) then begin
                  m := m+1;        levents[m] := ci;
                  end;
               ci := ci^.next;
               end;

            (* process each variable according to its type *)
            for i := 1 to nv do case pvariables^[i]^.dtype of

(* for nominal or structured variables, build value histograms and find the
   most common (for central) or least common (for distant) value.  the
   values become the nominal center of mass, stored in nomcent. *)
nominal,structured:
               begin
                  for j := 0 to maxevents do counter[j]:=0;
                  for j := 0 to m do begin
                     k := reflow(levents[j]^.selectors[i]);
                     counter[k] := counter[k]+1;
                     end;
                  case rep of
central:             begin
                        k := counter[0];
                        for j := 1 to pvariables^[i]^.maxvalue do
                           if counter[j]>k then k:=counter[j];
                        nomcent[i] := [];
                        for j := 0 to pvariables^[i]^.maxvalue do
                           if counter[j]=k then nomcent[i] := nomcent[i]+[j];
                        end;

distant:             begin
                        k := counter[0];
                        for j := 1 to pvariables^[i]^.maxvalue do
                           if counter[j]<k then k:=counter[j];
                        nomcent[i] := [0..pvariables^[i]^.maxvalue];
                        for j := 0 to pvariables^[i]^.maxvalue do
                           if counter[j]=k then nomcent[i] := nomcent[i] - [j];
                        end;
                     end;
                  end;

(* for linear and cyclic variables, the average value is computed and stored
   in lincent *)
linear,cyclic: begin
                  sum := 0;
                  for j := 0 to m do begin
                     sum := sum + reflow(levents[j]^.selectors[i]);
                     end;
                  lincent[i] := sum / (m+1);
                  end;
               end;(* for case *)

            (* measure all distances relative to center of mass *)
            bestdist := 1e99;
            for i := 0 to m do begin
               ci := levents[i];        dist := 0;
               for j := 1 to nv do case pvariables^[j]^.dtype of
```



```
nominal,structured:
                  case rep of
central:             if (nomcent[j]*ci^.selectors[j])=[] then dist:=dist+1;
distant:             if (nomcent[j]*ci^.selectors[j])<>[] then dist:=dist+1;
                     end;

linear,cyclic:    begin
                     dist := dist + abs(lincent[j]-reflow(ci^.selectors[j]))
                        / pvariables^[j]^.maxvalue;
                     end;
                  end;

               (* keep the best choice *)
               case rep of
central:          sdist[i] := dist;
distant:          sdist[i] := -dist;
                  end;
               if sdist[i]<bestdist then begin
                  bestdist := sdist[i];        bi := i;
                  end;
               end;

            if qcd in dbug then for i:=0 to m do begin
               write(' cendist: score of ');        prtmap(levents[i]^.map);
               writeln(' is ',sdist[i]:8);
               end;

(* choose the best seed *)
            if n<>psstat^.clusadj then begin
               (* if not the last seed to select, just save seed choice
                  and maintain seed set map *)
               cv^.pseed[n] := levents[bi];
               mop(seedset,union,seedset,levents[bi]^.map);
               end
            else begin
               (* when last seed is selected, check to make sure seed set
                  was never used previously *)
               j := 0;
               repeat
                  (* find best seed (again) *)
                  j := j+1;        bestdist := 1e99;        bi:=0;
                  for k:=0 to m do if sdist[k]<bestdist then begin
                     bestdist:=sdist[k];        bi:=k;
                     end;
                  (* make a temporary seed set map *)
                  emap := seedset;        mop(emap,union,emap,levents[bi]^.map);

                  (* search list of past-used seed set maps for duplication *)
                  sptr := psstat^.usedseeds;
                  goodseeds := true;
                  while goodseeds and (sptr<>nil) do begin
                     if mcomp(sptr^.seeds,eq,emap) then begin
                        (* mark seed with a bad score to eliminate it *)
                        goodseeds := false;        sdist[bi] := 2e99;
                        end;
                     sptr := sptr^.next;
```

```
            until goodseeds or (j>m);
         end;
      until n=pastat^.clusadj;

   if qcd in dbug then begin
      write(' cendist: ');
      if goodseeds then write('good seeds are ')
                   else write('duplicated seeds are ');
      prtmap(emap);       writeln;
      end;

   (* at this point, seeds may or may not be "goodseeds".
      if not goodseeds, then there is no seed in the 'adjust'
      cluster that can be selected to yield a good seed set
      (all such seeds were just tried above and failed) *)
   if goodseeds then begin
      seedset := emap;       cv^.pseed[n] := levents[bi];
      end
   else begin

      (* here we face the problem of how to find a unique set of seeds.
         if fewer than k (clstrs) attempts have been made, then just
         repeat the above which will try to make adjustments using a
         different cluster.  if k attempts have been made, then switch
         rep type from distant to centeral or vice-a-versa.  this is
         good for another k attempts.  if 2k attempts have been attempted
         then panic and try to find any event from any cluster that will
         create a unique seed set (i.e., abandon central or distant
         qualities entirely *)
      if attempts=clstrs then begin
         if qcd in dbug then writeln(' cendist: rep changed');
         if rep=central then rep:=distant else rep:=central;
         end;
      if attempts=2*clstrs then begin
         writeln;       writeln(' panic finding new seeds');
         ci := events;  (* ignore cluster boundaries; just find a seed *)
         while (ci<>nil) and not goodseeds do begin
            emap := seedset;       mop(emap,union,emap,ci^.map);
            if mcomp(emap,ne,seedset) then begin
               sptr := pastat^.usedseeds;       goodseeds := true;
               while goodseeds and (sptr<>nil) do begin
                  if mcomp(sptr^.seeds,eq,emap) then goodseeds:=false;
                  sptr := sptr^.next;
                  end;
               end;
            ci:=ci^.next;
            end;
         if goodseeds then begin
            seedset := emap;       cv^.pseed[n] := levents[bi];
            end
         else begin
            (* if this last-ditch effort fails, then give up *)
            panic := true;       goodseeds := true;
            writeln(' premature end of clustering: seeds exhausted');
            end;
         end;
      end;
```



```
         until goodseeds;

      (* record seed set that was generated so it will not be duplicated *)
      if not panic then begin
         new(sptr);       sptr^.next := pastat^.usedseeds;
         pastat^.usedseeds := sptr;
         sptr^.seeds := seedset;

         if qcd in dbug then begin
            write(' cendist: new ');
            case rep of
central:       write('central');
distant:       write('distant');
               end;
            write(' seeds are ');
            for i:=1 to clstrs do write(maplow(cv^.pseed[i]^.map):4);
            writeln;
            end;
         end;

      (* report success or failure in finding a new unique seed set *)
      cendist := not panic;
      end;

procedure critval;              (* evaluate complexes according to the lef *)
(*     (c              : pcomplex);
*)

var
      cn, ci, cj     : integer;  (* criterion indices *)
      i, j, k        : integer;
      ri             : real;     (* score on the criterion *)
      p              : pcomplex;
      mt             : evtmap;
      s, ss          : selmapunit;
      ok             : boolean;
      reuse          : boolean;
      maxv           : integer;

begin
      with pastat^ do c[qcl] := c[qcl]+1;

      (* if complex has changed, recalculate its 'area' *)
      if cfchanged in c^.cflags then begin
         c^.cflags := c^.cflags - [cfchanged];       setmap(c);       ri := 1;
         (* area is product of all reference list lengths *)
         for i:=1 to nv do begin
            j := reflev(c^.selectors[i]);
            if j>1 then ri := ri * j;
            end;
         c^.area := ri;

         if qcl in dbug then writeln(' critval: area=',ri);
         end;

      (* evaluate all criteria in the lef *)
      for cn := 0 to pastat^.crit.ccnt do begin
```



```
         (* ci is the criterion id number *)
         ci := pastat^.crit.clist[cn];
         if ci<0 then ci:=-ci;

         (* check if this same criterion has already been calculated *)
         reuse := false;       j := 0;
         while (j<cn) and (not reuse) do begin
            k := pastat^.crit.clist[j];
            if k<0 then k:=-k;
            if k=ci then reuse := true else j := j+1;
            end;

         if reuse then begin
            (* just retrieve previously calculated value *)
            ri := c^.critvals[j];
            if pastat^.crit.clist[j]<0 then ri:=-ri;
            end
         else begin
            (* calculate criterion value *)
            case ci of
1:             begin (* sparseness *)
                  ri := c^.area - wmcard(c^.map,0);
                  end;

2:             begin (* disjointness *)
                  ri := 0;       p := context;
                  while (p<>nil) do begin
                     if p<>c then ri := ri + degisct(c,p);
                     p := p^.next;
                     end;
                  end;

3:             begin (* # events covered more than once *)
                  p := context;       ri := 0;
                  while (p<>nil) do begin
                     mop(mt,inter,c^.map,p^.map);
                     if p<>c then ri := ri + wmcard(mt,0);
                     p := p^.next;
                     end;
                  end;

4:             begin (* balance *)
                  if level <= clstrs then begin
                     setlevelmap(context);       mop(mt,union,levelmap,c^.map);
                     ri := iabs(((numevents*level) div clstrs) - wmcard(mt,0));
                     end
                  else ri := iabs((numevents div clstrs)-wmcard(c^.map,0));
                  end;

5:             begin (* commonality *)
                  ri := -numsel(c);
                  end;

6:             begin (* dimensionality *)
                  ri:=0;
                  for i := 1 to nv do
                     if clstrs <= pvariables^[i]^.maxvalue+1 then begin
```



```
                        p := context;       ok := true;       j := 1;
                        s := c^.selectors[i];
                        while (j<=clstrs-level) and ok do begin
                           ss := cvcontext^.pseed[j]^.selectors[i];
                           ok := [] = s * ss;
                           s := s + ss;
                           j := j + 1;
                           end;
                        while (p<>nil) and ok do begin
                           if c<>p then ok := [] = s * p^.selectors[i];
                           s := s + p^.selectors[i];       p := p^.next;
                           end;
                        if ok then ri := ri - 1;
                        end;
                  end;

7:             begin (* simplicity *)
                  ri := 0;
                  for i:=1 to nv do begin
                     j := refnum(c^.selectors[i],i);
                     if j>0 then ri := ri + j - 1 + pvariables^[i]^.cost;
                     end;
                  end;

8:             begin (* projected sparseness *)
                  ri := 1;
                  for i:=1 to nv do begin
                     p := context;       j:=reflev(c^.selectors[i]);
                     maxv := pvariables^[i]^.maxvalue;
                     cj := j;
                     while (j>maxv) and (p<>nil) do begin
                        j:=reflev(p^.selectors[i]);       p := p^.next;
                        end;
                     if j<=maxv then ri := ri * cj;
                     end;
                  ri := ri - wmcard(c^.map,0);
                  end;

9:             begin (* number of exceptional events *)
                  ri := 0;  (* always zero at complex level *)
                  end;

               end;(*case*)
            end;

         if pastat^.crit.clist[cn]<0 then ri := -ri;
         c^.critvals[cn] := ri;

         if qcl in dbug then writeln(' critval: ',cn:2,ri);
         end;
      end;

(* rltb segment *)
(*$1'cluster/2 relational table input' *)
```

```
function stringmatch                         (* compare two strings *)
     (s1, s2          : alfafld;
      l               : integer): boolean;

var
      i               : integer;
      r               : boolean;

begin
      r := true;      i := 1;
      while r and (i<=l) do begin
         r := s1[i] = s2[i];        i := i+1;
         end;
      stringmatch := r;
      end;

procedure setup;

(* this procedure reads the relational table input to the program *)
(* and sets up all necessary data structures *)
(* algorithm properly *)

(* lexical characteristics:

   a relational table consists of a number of informational units
   separated by "white space" consisting of blanks and the characters
   , _ ! and " or any combination of these characters

   the input file contains a series of relational tables, each beginning
   with the table name alone on one line, followed by a line of column
   heading identifiers.  as many lines of data as necessary follow,
   containing values ordered according to the column headings
    *)

const
      numcoldfn     = 60;  (* total number of column definitions *)
      numtbldfn     = 15;  (* number of defined table types *)
      cecho         = false;  (*echo input character data*)
      blankkonst    = '          ';
      seqkonst      = '#         ';
      costkonst     = 'wt        ';
      intkonst      = ' int      ';
      charkonst     = ' char     ';

type
      colheader     = packed record
         name          : alfa;            (* header name *)
         table         : 0..63;           (* table id number *)
         action        : 0..511;          (* semantic action code *)
         convert       : conversion;      (* input conversion *)
                      end;

      tblheader     = record
         name          : alfa;            (* name of table *)
```



```
         id            : integer;  (* table id number *)
                      end;

var
      chartype      : char;                (* type of input: f,q,9,' ' *)
      inreal        : real;                (* input value, if real *)
      inint         : integer;             (* input value, if integer *)
      inchar        : record               (* input value, if character *)
         length        : integer;          (* number of chars read *)
                      case boolean of
false:  (chars         : lalfa);
true:   (alfas         : packed array[1..8] of alfa);
                      end;

      typecodes     : packed array[1..64] of char;  (* char types *)

      colord        : array[1..maxvars] of integer;  (* order of columns *)
                                          (* integers indicate defn numbers*)
      varord        : array[1..maxvars] of integer;  (* gives var in data table

      coldfn        : array[1..numcoldfn] of
                   colheader;(* definition of columns *)
      tbldfn        : array[1..numtbldfn] of tblheader;(* definition of table
      rwork         : packed array[1..maxlevels] of refval;

      colno, nxtcol: integer;              (* column management *)
      column        : integer;             (* current column number *)
      tidref        : integer;             (* current table *)
      i, j          : integer;
      ih, it        : integer;
      achold        : integer;             (* saved semantic action code *)
      cv            : conversion;          (* input conversion required *)
      eot           : boolean;             (* true if end-of-table *)
      mv, match     : boolean;             (* true if missing value *)
      dref, dend    : pdomain;             (* pointers for domain defns *)
      cref          : pcrit;               (* pointers for criteria defns *)
      tref, tend    : ptitle;              (* pointers to title text *)
      pref, pend    : ppara;               (* pointers to parameter defns *)
      nref          : pname;               (* pointers to name defns *)
      sref, send    : pstruct;             (* pointers to structures *)
      eref, eend    : pcomplex;            (* pointers to events *)
      lnexpectd     : integer;             (* line number expected next *)
      tid           : integer;             (* current table id *)
      sname         : alfa;                (* specific table name *)
      vname         : alfa;                (* name of value *)
      inlineno      : integer;             (* input line number *)

      blankconst    : alfa;
      seqconst      : alfa;
      costconst     : alfa;
      charconst     : alfa;
      intconst      : alfa;

function typcod
     (c             : char): char;
```



```
begin
   (* typcod := typecodes[1+ord(c)]; *)
      if ord(c)<96 then typcod := typecodes[ord(c)-31]
      else typcod := typecodes[ord(c)-63];
      end;

procedure getinput;

(* get next input character and echo it *)

begin
      get(input);
      if cecho then
         if eoln(input) then writeln else write(input^);
      end;

function eolninput: boolean;

(* this function detects eoln on the input file *)

begin
      eolninput := false;
      while (not eof(input)) and eoln(input) do begin
         inlineno := inlineno + 1;      getinput;
         chartype := typcod(input^);    eolninput := true;
         end;
      if eof(input) then eolninput := true;
      end;

function skipfill: boolean;

(* skip 'fill' chars, return true if eoln crossed *)

var
      eln           : boolean;           (* true if eoln found *)

begin
      eln := eolninput;      chartype := typcod(input^);
      while (chartype='f') and (not eof(input)) do begin
         getinput;      eln := eln or eolninput;
         chartype := typcod(input^);
         end;
      skipfill := eln;
      end;

procedure errormsg
     (errval         : integer;
      numval         : integer;
      alfaval        : alfafld);

(* write setup error messages                                          *)

begin
      write(' on line ',inlineno:1,': ');
      case errval of
1:       writeln('unmatched quotation marks:  ',alfaval);
```



```
3:       writeln('integer has a fractional part');
4:       writeln('item(s) missing');
5:       writeln('domain not defined:  ',alfaval);
6:       writeln('criteria-table name not defined:  ',alfaval);
7:       writeln('incorrect relational table-name:  ',alfaval);
8:       writeln('row out of sequence:  ',numval);
9:       writeln('invalid numeric item:  ',alfaval);
10:      writeln('invalid cyclic variable values:  ',alfaval);
11:      writeln('invalid variable value:  ',numval);
12:      writeln('invalid value for type:  ',alfaval);
13:      writeln('too many criteria (must be <= ',maxcrit,')');
14:      writeln('unknown variable value:  ',alfaval);
15:      writeln('incorrect column header:  ',alfaval);
16:      writeln('invalid mode option:  ',alfaval);
17:      writeln('invalid trace option:  ',alfaval);
23:      writeln('invalid linear variable values:  ',alfaval);
33:      writeln('invalid value for initmethod: ',alfaval);
34:      writeln('invalid value for midspeed: ',alfaval);
35:      writeln('invalid value for covertype: ',alfaval);
36:      writeln('insufficient memory to store events');
37:      writeln('criterion name ',alfaval,' is unknown');
38:      writeln('node value of ',numval:1,' conflicts with a leaf value');
39:      begin
            write('invalid debug request: ');
            if numval=0 then writeln('"',alfaval,'"') else writeln(numval:1);
            end;
         end;
      halt;                              (* no recovery is attempted     *)
      end;

function readcolumn
     (conv           : conversion;
      var eot        : boolean;
      var missing    : boolean;
      var nextcol    : integer): integer;

(* read a relational table column according to conv and return column no. *)
(* eot means 'end of table'; missing means 'no value found' *)

(* for "charval" input: the data format can be *)
(* (1) a string of non-fill chars surrounded by fill *)
(* (2) a string without internal ' enclosed by '-marks *)
(* (3) a string without internal" enclosed by "-marks *)

(* for "realval" input: the data format can be *)
(* an optional + or -, followed by any format acceptable to pascal *)

(* for "intval" input: the data format can be *)
(* anything described above for "realval" with a zero fractional part *)


var
      id             : boolean;          (* true if possible tid found *)
      m              : boolean;
      delim          : char;             (* string delimiter char *)
      sign, r        : real;             (* used to read numerals *)
```

```
     n              : integer;

begin
     cno := nextcol;        chartype := typcod(input^);
     if chartype='f' then begin
        if skipfill then cno := 1;
        end;
     if eof(input) then id := true else case conv of

charval: begin
           inchar.length := 0;       strasgn(inchar.alfas[1],'            ');
           if chartype = 'q' then begin
              delim := input^;      getinput;
              while(input^ <> delim) and (not eoln(input)) do begin
                 inchar.length := inchar.length+1;
                 inchar.chars[inchar.length] := input^;
                 getinput;
                 end;
              if eoln(input) then errormsg(1,0,inchar.alfas[1]) else getinput;
              end
           else while (chartype <> 'f') and (not eoln(input)) do begin
              inchar.length := inchar.length+1;
              inchar.chars[inchar.length] := input^;
              getinput;
              chartype := typcod(input^);
              end;
           missing := typcod(inchar.chars[1])='m';       id := skipfill;
           end;

intval,realval:
         begin
           sign := 1.0;       id := false;       missing:=typcod(input^)='m';
           if input^ = '+' then getinput
           else if input^ = '-' then begin
              sign := -1.0;       getinput
              end;
           chartype := typcod(input^);       id := skipfill;
           if chartype <> '9' then begin
              m := nextcol;       cno := readcolumn(charval,id,m,n);
              if not (id or missing) then errormsg(9,0,inchar.alfas[1]);
              end
           else begin
              read(input,inreal);
              if cecho then begin
                 write('(',inreal:0:0,') ');
                 if eoln(input) then writeln;
                 end;
              inreal := inreal * sign;
              if conv = intval then begin
                 inint := trunc(inreal);       r := inint;
                 if r <> inreal then errormsg(3,0,blankconst);
                 end;
              chartype := typcod(input^);
              if chartype <> 'f' then errormsg(2,0,blankconst)
                                 else id := skipfill;
              end;
           end;
```



```
        end;
     if cno <> 1 then eot := false else eot := id;
     nextcol := nextcol + 1;
     if id then nextcol := 1;
     readcolumn := cno;
     end;

procedure tablesetup;

(* identify relational table and assign columns                          *)

var
     i, j, k, l     : integer;
     glen           : integer;          (* length of general table name   *)
     slen           : integer;          (* length of specific table name  *)
     gname          : alfafld;          (* general table name             *)
     match          : boolean;          (* true if item found             *)
     id, eoh        : boolean;          (* true if eoln reached           *)


begin
     with pastat^ do c[qtp] := c[qtp]+1;
     if qtp in dbug then begin
        write(' tablesetup--inchar=');
        for i := 1 to inchar.length do write(inchar.chars[i]);
        writeln;
        end;

     glen := 0;       slen := 0;       gname := blankconst;
     sname := blankconst;

(* table name of the form: specific-general                              *)
(* separate at hyphen and leave in sname and gname                       *)

     i := 1;
     while (inchar.chars[i]<>'-') and (i<=inchar.length) do begin
        slen := slen+1;
        if slen <= alfasize then sname[slen] := inchar.chars[i];
        i := i+1;
        end;
     if inchar.chars[i]='-' then repeat
        i := i+1;       glen := glen+1;
        if glen <= alfasize then gname[glen] := inchar.chars[i];
        until i >= inchar.length;
     if glen=0 then begin
        gname := sname;       glen := slen;
        end;

(* identify general table type                                           *)

     tid:=1;
     while (not stringmatch(gname,tbldfn[tid].name,glen)) and
        (tbldfn[tid].id<>0) do tid:=tid+1;
     if tbldfn[tid].id=0 then errormsg(7,0,gname);
     tid := tbldfn[tid].id;                        (* incorrect table name    *)
     case tid of
```



```
1:        begin
             titles := nil;       tend := nil;
             end;

2:        begin
             parameters := nil;       pend := nil;
             end;

3:        begin
             domains := nil;       dend := nil;
             end;

4:        if pvariables=nil then begin
             new(pvariables);
             for i:=1 to maxvars do pvariables^[i] := nil;
             end;

5,6,9:    begin                         (* names, onames, structure *)
             dref := domof(sname);
             if dref=nil then errormsg(5,0,sname); (* domain name unknown *)
             if (tid=5) or (tid=9) then begin       (* initialize names lists   *)
                new(nref);       dref^.onames := nref;
                if (tid=5) then dref^.inames := nref;
                for i:=1 to maxlevels do begin
                   nref^.names[i] := blankconst;       nref^.names[i][1] := 'x';
                   if i<10 then nref^.names[i][2] := chr(ord('0')+i) else begin
                      nref^.names[i][2] := chr(ord('0')+(i div 10));
                      nref^.names[i][3] := chr(ord('0')+(i mod 10));
                      end;
                   end;
                end;
             end;

7:        begin
             cref := criteria;       match := false;
             if cref<>nil then repeat
                match := sname=cref^.name;
                if not match then cref := cref^.next;
                until match or (cref=nil);
             if cref=nil then errormsg(6,0,sname); (* table name not defined *)
             end;

8:        begin                        (* event-tables    *)
             eref := nil;
             end;

10:       dbug := dbug * [q00];       (* clear dbug codes *)



          end;

(* process column headings                                               *)

     if (tid > 0) then begin
          eoh := false;
```



```
(* "noval" pseudo-column   *)
        i := 1;       inchar.alfas[i] := blankconst;       k := 1;       l := 0;
        inchar.length := alfasize;
        repeat

(* following special treatment applies to event data only               *)

           if (tid=8) and (i>1) and (inchar.alfas[i]<>seqconst) and
              (inchar.alfas[i]<>costconst) then begin
              j := 1;       dref := nil;
              while (dref=nil) and (j<=maxvars) do begin
                 dend := pvariables^[j];
                 if dend<>nil then
                    if dend^.name=inchar.alfas[i] then dref:=dend;
                 j := j+1;
                 end;
              if dref=nil then errormsg(15,0,inchar.alfas[i]);
              j := j-1;       varord[i-1] := j;       inchar.length := alfasize
              if dref^.inames<>nil then inchar.alfas[i] := charconst
                                   else inchar.alfas[i] := intconst;
              end;

           j := 1;
           repeat
              match := (tid=coldfn[j].table) and stringmatch(inchar.alfas[i],
                 coldfn[j].name,inchar.length);
              j := j+1;
              until match or (coldfn[j].table=0);
           j := j-1;

(* column order indicated by values in the array colord                  *)

           if match then colord[i] := j else errormsg(16,0,inchar.alfas[i]);
           if k>1 then l:=readcolumn(charval,id,mv,k) else eoh := true;
           i := i+1;
           until id or eoh;

        colord[i] := 0;                 (* set circular flag             *)
        lnexpectd := 0;                 (* the next line seq. no is 1    *)

        end;
     end;

(*-----------------------------------*)

function dtypeval: domaintype;

(* decode inchar and return domain type indicated                        *)

begin
     if inchar.chars[1]='n' then dtypeval := nominal
     else if inchar.chars[1]='l' then dtypeval := linear
     else if inchar.chars[1]='c' then dtypeval := cyclic
     else if inchar.chars[1]='s' then dtypeval := structured
                                 else errormsg(12,0,inchar.alfas[1]);
     end;                              (* invalid value for type    *)
```

```
procedure semantics
     (act            : integer);

(* perform action as per 'act' to accept input data                    *)

var
      i, j, k, l     : integer;
      si, sj         : stattyp;
      a1, a2         : alfa;
      c              : char;


procedure critdef
     (cn             : alfafld);

(* to define a criteria table                                           *)

var
      cp             : pcrit;
      match          : boolean;

begin
      cp := criteria;       match := false;
      while (not match) and (cp<>nil) do begin
         match := cn=cp^.name;       cp := cp^.next;
         end;
      if not match then begin
         new(cp);       cp^.next := criteria;       cp^.name := cn;
         criteria := cp;                            cp^.clist[0]  := 1;
         cp^.clist[1] := 1;                         cp^.tlist[0]  := 0.0;
         cp^.tlist[1] := 0.0;                       cp^.cent := 1;
         end;
      end;


begin (* semantics *)
      with psstat^ do c[qss] := c[qss]+1;
      if qss in dbug then begin
         write(' semantics: act = ',act:1,' column = ',column:1);
         case cv of
noval:      writeln(' noval');
intval:     writeln(' intval=',inint:1);
realval:    writeln(' realval=',inreal:1:2);
charval:    begin
               write(' charval=');
               for i:=1 to inchar.length do write(inchar.chars[i]);
               writeln;
               end;
            end;
         end;

      case act of

1:       begin                              (* prep title *)
            lnexpectd := lnexpectd+1;       new(tref);
```



```
            if tend=nil then titles := tref else tend^.next := tref;
            tend := tref;
            with tref^ do begin
               next := nil;      length := 0;
               end;
            end;

2:       begin                              (* check int expected *)
            if inint<>lnexpectd then errormsg(8,inint,blankconst);
            end;

3:       begin                              (* title text *)
            for i := 1 to inchar.length do tref^.text[i] := inchar.chars[i];
            tref^.length := inchar.length;
            end;

4:       begin                              (* prep parameters *)
            lnexpectd := lnexpectd+1;       new(pref);
            if pend=nil then parameters := pref else pend^.next := pref;
            pend := pref;
            with pref^ do begin
               next := nil;       mink:=2;       maxk:=4;       debug := [];
               h1 := 3;           h2 := 2;       h3 := 3;
               initmthd := random;               nidspeed := fast;
               covertp := disjoint;              strasgn(critname,' defatlt  ');
               critdef(critname);                base := 2;     probe := 2;
               beta := 3.0;       maxht := 99;   minsize := 4;
               end;
            end;

5:       begin                              (* set clstrs *)
            pref^.mink := inint;      pref^.maxk := inint;
            end;

6:       begin                              (* set trace *)
            vname := inchar.alfas[1];
            if (vname[1]='o') and (vname[2]='f') then pref^.debug:=dbug
            else if (vname[1]='o') and (vname[2]='n') then
               pref^.debug:=dbug + [q00]
            else errormsg(17,0,vname);
            end;


7:       pref^.h1 := inint;                 (* set heuristic 1 *)

8:       pref^.h2 := inint;                 (* set heuristic 2 *)

9:       begin                              (* set initmethod *)
            if inchar.chars[1]='r' then pref^.initmthd := random
            else if inchar.chars[1]='o' then pref^.initmthd := ormeth
                                       else errormsg(33,0,inchar.alfas[1]);
            end;

10:      begin                              (* set nidspe:d *)
            if inchar.chars[1]='s' then pref^.nidspeed := slow
            else if inchar.chars[1]='f' then pref^.nidspeed := fast
                                       else errormsg(34,0,inchar.alfas[1]);
```



```
            end;

11:      begin                              (* set covertp  *)
            if inchar.alfas[1][1]='d' then pref^.covertp := disjoint
            else if inchar.alfas[1][1]='i' then pref^.covertp := intersecting
            else if inchar.alfas[1][1]='h' then pref^.covertp := hierarchical
                                       else errormsg(35,0,inchar.alfas[1]);
            end;

12:      begin                              (* set base value *)
            pref^.base := inint;
            end;

13:      begin                              (* set crit name *)
            pref^.critname := inchar.alfas[1];      critdef(pref^.critname);
            end;

14:      begin                              (* set probe value *)
            pref^.probe := inint;
            end;

15:      begin                              (* prep domains *)
            new(dref);
            if dend=nil then domains := dref else dend^.domain := dref;
            dend := dref;
            with dref^ do begin
               strasgn(name,'          ');      cost := 1;
               dtype := nominal;                 maxvalue := setsize;
               onames := nil;                    nwidth := 2;
               domain := nil;                    inames := nil;
               end;
            end;

16:      dref^.name := inchar.alfas[1];   (* set domain name *)

17:      dref^.cost := inint;               (* set domain cost *)

18:      dref^.dtype := dtypeval;           (* set dtype *)

19:      dref^.maxvalue := inint-1;         (* set domain maxvalue *)

20:      begin                              (* prep variables *)
            lnexpectd := lnexpectd+1;       nv := lnexpectd;
            new(pvariables^[lnexpectd]);
            with pvariables^[lnexpectd]^ do begin
               strasgn(name,'x         ');      cost := 1;
               dtype := nominal;                 maxvalue := setsize;
               domain := nil;                    structlist := nil;
               onames := nil;                    inames:=nil;
               nwidth := 3;                      j := lnexpectd mod 10;
               i := lnexpectd div 10;
               if i=0 then name[2] := chr(ord('0')+j) else begin
                  name[2] := chr(ord('0')+i);      name[3] := chr(ord('0')+j)
                  end;
               end;
            end;
```



```
21:      begin                              (* set variable name *)
            i:=1;       strasgn(vname,'          ');
            while (inchar.chars[i]<>'-') and (i<=inchar.length) do begin
               if i<=alfasize then vname[i] := inchar.chars[i];
               i := i+1;
               end;
            pvariables^[lnexpectd]^.name := vname;      j:=alfasize;
            while (j>1) and (vname[j]=' ') do j:=j-1;
            pvariables^[lnexpectd]^.nwidth := j;
            if inchar.chars[i]='-' then begin
               j := 0;      strasgn(vname,'          ');
               repeat
                  i := i+1;      j := j+1;
                  if j<=alfasize then vname[j]:=inchar.chars[i];
                  until (i>=inchar.length);
               dref := domains;      dend := nil;
               while (dref<>nil) and (dend=nil) do begin
                  if dref^.name=vname then dend := dref;
                  dref := dref^.domain;
                  end;
               if dend<>nil then pvariables^[lnexpectd]^.domain := dend
                         else errormsg(5,0,vname);
               end;
            end;

22:      pvariables^[lnexpectd]^.cost := inint; (* set variable cost*)

23:      pvariables^[lnexpectd]^.dtype := dtypeval;(* set domain type *)

24:      pvariables^[lnexpectd]^.maxvalue := inint-1;(* set maxvalue *)

25:      begin                              (* prep names *)
            nref := nil;      strasgn(vname,'          ');      lnexpectd := 0;
            end;

26:      begin                              (* set value *)
            lnexpectd := inint+1;
            if (vname[1]<>' ') then begin
               inchar.alfas[1] := vname;      cv:=charval; semantics(27);
               end;
            end;

27:      begin                              (* set name *)
            vname := inchar.alfas[1];       nref := dref^.onames;
            if (nref<>nil) and (lnexpectd>0) then begin
               i:=2;
               while (i<alfasize) do begin
                  if (vname[i]='.') and (vname[i+1]='.') then begin
                     for j:=i-1 downto 1 do vname[j+1]:=vname[j];
                     vname[1]:='!';      vname[i+1]:='!';      i:=alfasize;
                     end;
                  i := i+1;
                  end;
               nref^.names[lnexpectd] := vname;
               if inchar.length > alfasize then inchar.length := alfasize;
               if inchar.length>dref^.nwidth then dref^.nwidth := inchar.leng
               end;
```

```
        end;

28:     begin                           (* prep structure *)
           sref := nil;       lnexpectd := 0;       strasgn(vname,'          ');
           end;

29:     begin                           (* struct name *)
           vname := inchar.alfas[1];       sref := dref^.structlist;
           inint := dref^.maxvalue+1;
           while (sref<>nil) do begin
              if (vname = sref^.name) then begin
                 inint := sref^.valu;       sref := nil;
                 end
              else begin
                 if (sref^.valu+1) > inint then inint := sref^.valu+1;
                 sref := sref^.next;
                 end;
              end;
           cv:=intval; semantics(30);
           end;

30:     begin                           (* struct value *)
           if inint <= dref^.maxvalue then errormsg(38,inint,blankconst);
           sref := dref^.structlist;       match := false;
           while (sref<>nil) and (not match) do begin
              match := sref^.valu=inint;
              if not match then sref := sref^.next;
              end;
           if sref=nil then begin
              new(sref);      sref^.valu := inint;      sref^.name := vname;
              sref^.next := nil;                        sref^.nodedefn := [];
              send := dref^.structlist;
              if send=nil then dref^.structlist := sref else begin
                 while (send^.next<>nil) do send := send^.next;
                 send^.next := sref;
                 end;
              end;
           for i:=1 to lnexpectd do begin
              inint := rwork[i];       cv:=intval; semantics(32);
              end;
           lnexpectd := 0;
           end;

31:     begin                           (* struct subname *)
           vname := inchar.alfas[1];       send := dref^.structlist;
           inint := -1;
           while (send<>nil) and (inint<0) do begin
              if vname = send^.name then inint := send^.valu
                                    else send := send^.next;
              end;
           nref := dref^.inames;
           if (inint<0) and (nref<>nil) then begin
              i := 0;
              while (inint<0) and (i<=dref^.maxvalue) do begin
                 if vname = nref^.names[i+1] then inint := i;
                 i := i+1;
                 end;
```



```
              end;
           if inint < 0 then errormsg(14,0,vname);
           cv:=intval; semantics(32);
           end;

32:     begin                           (* struct subvalue *)
           if sref=nil then begin
              lnexpectd := lnexpectd+1;       rwork[lnexpectd] := inint;
              end
           else begin
              if inint<=dref^.maxvalue then sref^.nodedefn :=
                 sref^.nodedefn + [inint]
              else begin
                 send := dref^.structlist;
                 while (send<>nil) do begin
                    if (send^.valu=inint) then sref^.nodedefn :=
                       sref^.nodedefn + send^.nodedefn;
                    send := send^.next;
                    end;
                 end;
              end;
           end;

33:     begin                           (* prep crit *)
           lnexpectd := lnexpectd + 1;
           if lnexpectd>maxcrit then errormsg(13,0,blankconst);
           end;

34:     begin                           (* set crit number *)
           cref^.clist[lnexpectd] := inint;      cref^.ccnt := lnexpectd;
           end;

35:     begin                           (* set tolerance *)
           cref^.tlist[lnexpectd] := inreal;
           end;

36:     begin                           (* prep event data *)
           lnexpectd := lnexpectd+1;
           if eref<>nil then eref := eref^.next else eref := events;
           if eref=nil then begin
              eref := newcplx;        numevents := numevents+1;
              if eref=nil then errormsg(36,0,blankconst);
              for i:=1 to nv do eref^.selectors[i] := [];
              mop(eref^.map,clr,eref^.map,eref^.map);      eref^.cost := 1;
              masgn(eref^.map,lnexpectd-1);
              if eend=nil then events := eref else eend^.next := eref;
              eend := eref;
              end;
           end;

37:     begin                           (* event int value*)
           i := varord[column];
           if inint>pvariables^[i]^.maxvalue then errormsg(11,inint,blankcons
              ;
           eref^.selectors[i] := [inint];
           end;
```



```
38:     begin                           (* event name value*)
           i := varord[column];        nref := pvariables^[i]^.inames;
           j := 1;                     k := 0;
           while (nref<>nil) and (k=0) and (j<=maxlevels) do begin
              if nref^.names[j]=inchar.alfas[1] then k:=j;
              j := j+1;
              end;
           if k=0 then errormsg(14,0,inchar.alfas[1]);
           eref^.selectors[i] := [k-1];
           end;

39:     begin                           (* debug name *)
           sj := q00;      si := q00;      vname := inchar.alfas[1];
           while (sj=q00) and (si<qlast) do begin
              statnam(si,ai);
              if vname = ai then sj := si;
              si := succ(si);
              end;
           if sj=q00 then errormsg(39,0,vname);
           dbug := dbug + [sj];
           end;

40:     pref^.h3 := inint;              (* set h3 *)

41:     pref^.mink := inint;            (* min k *)

42:     pref^.maxk := inint;            (* max k *)

43:     pref^.beta := inreal;           (* beta *)

44:     pref^.maxht := inint;           (* maxheight *)

45:     pref^.minsize := inint;         (* minsize *)

46:     eref^.cost := inint;            (* event weight *)

47:     begin                           (* criterion name *)
           i := 1;       a1 := inchar.alfas[1];
           repeat
              l := 0;
              for j := 1 to maxcrit do begin
                 a2[1] := ' ';      prtcrit(j,a2);      k := 1;
                 while (k>1) and (a1[k]=a2[k]) do k:=k-1;
                 if (k=1) and (a1[1]=a2[1]) then begin
                    inint := j;       l := l+1;
                    end;
                 end;
              i := i+1;
              until (l=1) or (i>5);
           cv:=intval;
           if l=1 then semantics(34) else errormsg(37,0,a1);
           end;

48:     begin                           (* prep debug *)
           lnexpectd := lnexpectd + 1;
           end;
```



```
49:     begin                           (* print parameter *)
           for i:=1 to inchar.length do begin
              c := inchar.chars[i];
              if (c>='a') and (c<='z') then prtflags := prtflags + [c];
              (* if i option in print spec, turn on trace *)
              if 'i' in prtflags then pref^.debug := dbug + [q00];
              if 't' in prtflags then prtflags := [];
              if 'a' in prtflags then prtflags := ['a'..'z']-['t'];
              end;
           end;

        end;
     end; (* semantics *)

procedure dh
     (keywrd        : alfa10;
      act           : integer;
      conv          : conversion);

(* define header keywords *)

(* the format is: dh('word      ',tid,act,conv)  *)
(* where tid is a table id number, act is a semantic action number *)
(* and conv is an input conversion code, or "noval" if nothing is read *)


begin
     ih := ih+1;
     with coldfn[ih] do begin
        strasgn(name,keywrd);      table := tidref;      action := act;
        convert := conv;
        end;
     end;

procedure dt
     (tname         : alfa10;
      tid           : integer);

(* define relational table *)

(* format: dt('tablename ',tid); *)

begin
     it:=it+1;      strasgn(tbldfn[it].name,tname);      tbldfn[it].id:=tid;
     tidref := tid;
     end;


begin (* setup *)
     strasgn(blankconst,blankkonst);       strasgn(seqconst,seqkonst);
     strasgn(costconst,costkonst);         strasgn(intconst,intkonst);
     strasgn(charconst,charkonst);         events := nil;
     eend := nil;                          nref := nil;
     cref := nil;                          inlineno := 1;

(* typecodes below denotes character classes: *)
(* 9=>numeric,  f=>fill,  q=>qote,  m=>missing data *)
```

```
(* blank denotes ordinary characters *)

     typecodes :=
    (*:abcdefghijklmnopqrstuvwxyz0123456789+-*/()$= ,.#[]%"_!&'?<>@;*)
   (* '                            9999999999  m   m f 9    q f qm      '; *)
    (* !"#$%&'()*+,-./0123456789:;<=>?@abcdefghijklmnopqrstuvwxyz[]^_*)
     'ffq m  q  m      9999999999     m                                  ';

     ih := 0;       it := 0;            (* column headings definitions *)

     dt('title       ',1);                          (* title *)
     dh('            ',1,noval);                    (* *)
     dh('type        ',1,charval);                  (* *)
     dh('text        ',3,charval);                  (* *)

     dt('parameters',2);                            (* parameters *)
     dh('            ',4,noval);                    (* *)
     dh('k           ',5,intval);                   (* *)
     dh('mink        ',41,intval);                  (* *)
     dh('maxk        ',42,intval);                  (* *)
     dh('trace       ',6,charval);                  (* *)
     dh('h1          ',7,intval);                   (* *)
     dh('h2          ',8,intval);                   (* *)
     dh('h3          ',40,intval);                  (* *)
     dh('initmethod',9,charval);                    (* *)
     dh('midspeed    ',10,charval);                 (* *)
     dh('covertype ',11,charval);                   (* *)
     dh('base        ',12,intval);                  (* *)
     dh('criterion ',13,charval);                   (* *)
     dh('probe       ',14,intval);                  (* *)
     dh('beta        ',43,realval);                 (* *)
     dh('maxheight ',44,intval);                    (* *)
     dh('minsize     ',45,intval);                  (* *)
     dh('print       ',49,charval);                 (* *)

     dt('domains     ',3);                          (* domains *)
     dh('            ',15,noval);                   (* *)
     dh('name        ',16,charval);                 (* *)
     dh('cost        ',17,intval);                  (* *)
     dh('type        ',18,charval);                 (* *)
     dh('levels      ',19,intval);                  (* *)

     dt('variables ',4);                            (* variables *)
     dh('            ',20,noval);                   (* *)
     dh('#           ',2,intval);                   (* *)
     dh('name        ',21,charval);                 (* *)
     dh('cost        ',22,intval);                  (* *)
     dh('type        ',23,charval);                 (* *)
     dh('levels      ',24,intval);                  (* *)

     dt('names       ',5);                          (* names *)
     dh('            ',25,noval);                   (* *)
     dh('value       ',26,intval);                  (* *)
     dh('name        ',27,charval);                 (* *)

     dt('structure ',6);                            (* structure *)
     dh('            ',28,noval);                   (* *)
```



```
     dh('name        ',29,charval);                 (* *)
     dh('value       ',30,intval);                  (* *)
     dh('subname     ',31,charval);                 (* *)
     dh('subvalue    ',32,intval);                  (* *)

     dt('criterion ',7);                            (* criterion *)
     dh('            ',33,noval);                   (* *)
     dh('#           ',2,intval);                   (* *)
     dh('crit#       ',34,intval);                  (* *)
     dh('crit        ',34,intval);                  (* *)
     dh('criterion ',47,charval);                   (* *)
     dh('tol         ',35,realval);                 (* *)
     dh('tolerance ',35,realval);                   (* *)

     dt('events      ',8);                          (* events *)
     dh('            ',36,noval);                   (* *)
     dh(seqkonst,2,intval);                         (* *)
     dh(intkonst,37,intval);                        (* *)
     dh(charkonst,38,charval);                      (* *)
     dh(costkonst,46,intval);                       (* *)

     dt('onames      ',9);                          (* output-only names *)
     dh('            ',25,noval);                   (* *)
     dh('value       ',26,intval);                  (* *)
     dh('name        ',27,charval);                 (* *)

     dt('debug       ',10);                         (* debug control *)
     dh('            ',48,noval);                   (* *)
     dh('#           ',2,intval);                   (* *)
     dh('name        ',39,charval);                 (* *)

     dt(' the end   ',0);                           (* *)
     dh(' the end   ',0,noval);                     (* *)
(* the last entry must have a act of zero (end of list mark) *)


(* we start by assuming that title data will be first *)
(* i.e. as if we had already seen   title    followed by *)
(*                                  type  text             *)

     for i := 1 to 3 do colord[i] := i;
     colord[4] := 0;      tid := -1;      tend := nil;      lnexpectd := 0;


(* read all relational table input *)
     repeat
        column := 0;      i := 0;      eot := false;      achold := 0;
        nxtcol := 1;
        repeat
           i := i+1;      j := colord[i];
           if j=0 then begin
              i := 0;      column := 0;      nxtcol := 1;
              end
           else begin
              cv := coldfn[j].convert;
              if cv=noval then achold := coldfn[j].action else begin
                 colno := readcolumn(cv,eot,av,nxtcol);
```



```
                 column := column + 1;
                 if column<>colno then errormsg(4,0,blankconst);
                 if (not eot) then begin
                    if achold<>0 then semantics(achold);
                    achold := 0;
                    if not av then semantics(coldfn[j].action);
                    end;
                 end;
              end;
           until eot;

        if tid=4 then begin
           for i:=1 to nv do if pvariables^[i]^.domain<>nil then begin
              dref := pvariables^[i];      dend := pvariables^[i]^.domain;
              dref^.cost := dend^.cost;    dref^.dtype := dend^.dtype;
              dref^.maxvalue := dend^.maxvalue;
              if dend^.onames<>nil then dref^.onames := dend^.onames;
              if dend^.inames<>nil then dref^.inames := dend^.inames;
              if dend^.structlist<>nil then dref^.structlist :=
                 dend^.structlist;
              if dend^.nwidth>dref^.nwidth then dref^.nwidth:=dend^.nwidth;
              end;
           end;

        if (tid>0) then case tid of
1:         prtrlt(tid,sname);
2:         if 'p' in prtflags then prtrlt(tid,sname);
3,4:       if 'v' in prtflags then prtrlt(tid,sname);
5,6,9:     if 'n' in prtflags then prtrlt(tid,sname);
7:         if 'c' in prtflags then prtrlt(tid,sname);
8,10:      if 'a' in prtflags then prtrlt(tid,sname);
           end;
                                  (* write back what was read *)

        if not eof(input) then tablesetup;
        until eof(input);

     (* if there are events, display them *)
     if (events<>nil) then
     if ('e' in prtflags) and not ('a' in prtflags) then prtrlt(8,sname);

     end;                                (* we stop and end of file *)

(* prim segment *)
(*$l'cluster/2 conceptual primitives' *)




function tcover(*     (c1, c2      : pcomplex): boolean;
*)
     ;

(* if c1 is completely covered by c2, then true *)

var
```



```
     i               : integer;

begin
     with pastat^ do c[qtr] := c[qtr]+1;
     i := 0;
     repeat
        i := i+1;
        if not (c1^.selectors[i]<=c2^.selectors[i]) then i := nv+1;
        until (i>=nv);
     if qtr in dbug then begin
        writeln(' tcover: ',i:1);      prt(c1,1);      prt(c2,2);
        end;
     tcover := i<=nv;
     end;

function cmpcov(*     (c1, c2      : pcomplex): ccode;
*)
     ;

(* c1 is compared to c2, the result being one of: 'c1c2' if c2 covers c1 *)
(* 'c2c1' if c1 covers c2, or 'ind' otherwise *)

var
     i               : integer;
     cd              : ccode;

begin
     with pastat^ do c[qcv] := c[qcv]+1;
     i := 0;
     repeat i := i+1 until(c1^.selectors[i]<>c2^.selectors[i]) or (i=nv);
     if c1^.selectors[i]<=c2^.selectors[i] then cd := c1c2 else cd := c2c1;
     if i<nv then case cd of

c1c2:   repeat
           i := i+1;
           if not (c1^.selectors[i]<=c2^.selectors[i]) then i := nv+1;
           until (i>=nv);

c2c1:   repeat
           i := i+1;
           if not (c2^.selectors[i]<=c1^.selectors[i]) then i := nv+1;
           until (i>=nv);

        end;
     if i<=nv then cmpcov := cd else cmpcov := ind;
     end;

function reflow(*     (s             : selmapunit): integer;
*)
     ;

(* returns the bit number of the lowest bit in s *)

var
     i               : integer;

begin
```

```
      with pastat^ do c[qrv] := c[qrv]+1;
      i := -1;
      repeat i := i+1 until (i=setsize) or (i in s);
      reflow := i;
      end;

function refhigh(*     (s            : selmapunit): integer;
*)
      ;

(* returns the bit number of the highest bit in s *)

var
      i              : integer;

begin
      with pastat^ do c[qrh] := c[qrh]+1;
      if ([10..setsize]*s) = [] then i := 10
      else if ([20..setsize]*s) = [] then i := 20 else i := setsize+1;
      repeat i := i-1 until (i=0) or (i in s);
      refhigh := i;
      end;

function reflev(*     (s            : selmapunit): integer;
*)
      ;

(* returns the number of bits in s *)

begin
      with pastat^ do c[qrv] := c[qrv]+1;
      reflev := card(s);
      end;

function refnum(*     (s            : selmapunit;
      v           : varidx): integer;
*)
      ;

(* returns the number of syntactic referents in s *)

var
      i              : integer;

begin
      with pastat^ do c[qrm] := c[qrm]+1;
      i := card(s);
      if i> pvariables^[v]^.maxvalue then i := 0
      else if i>1 then case pvariables^[v]^.dtype of

nominal: ;

linear,cyclic:
         i := 2;

structured:
         i := 1;
```



```
         end;
      refnum := i;
      end;

procedure genrliz(*     (c            : pcomplex);
*)
      ;

(* make all selectors expressable in vl1 by appropriate generalization *)

var
      i,ii,j,l,r,m,mv,g: integer;
      ws              : selmapunit;
      sp              : pstruct;
      os, ns          : selmapunit;
      ds              : selmapunit;

procedure genbest
     (hs              : selmapunit);

(* this procedure saves the best-fitting generalization *)

begin
      if os<=hs then
         if card(hs-os)<ii then begin
            ws:=hs;       ii:=card(hs-os);
            end;
      end;

begin
      with pastat^ do c[qgz] := c[qgz]+1;
      if qgz in dbug then begin
         writeln(' genrliz: before(1) and after(2)');       prt(c,1);
         end;
      for i := 1 to nv do if pvariables^[i]^.dtype <> nominal then begin
         os := c^.selectors[i];       m := reflev(os);
         mv := pvariables^[i]^.maxvalue;
         if (m>1) and (m<=mv) then begin
            case pvariables^[i]^.dtype of

nominal:          ;

linear:           begin
                     ns := [reflow(os)..refhigh(os)];
                     end;

cyclic:           begin
                     l := refhigh(os);       r := reflow(os);       m := 1;
                     g := mv-l+r+1;          ii := r;
                     while (ii<m) do begin
                        j := ii;
                        repeat j := j+1 until j in os;
                        if j-ii>g then begin
                           l := ii;       r := j;       g := j-ii;
                           end;
                        ii := j;
```



```
                        end;
                     if l>r then ns := [r..l] else ns := [0..l,r..mv];
                     end;

structured:       begin
                     sp := pvariables^[i]^.structlist;       ws := [];
                     ds := [0..pvariables^[i]^.maxvalue];   ii := setsize;
                     while (sp<>nil) do begin
                        genbest(sp^.nodedefn);       genbest(ds-sp^.nodedefn);
                        sp := sp^.next;
                        end;
                     for j:=0 to pvariables^[i]^.maxvalue do genbest(ds-[j]);
                     if ws=[] then ns := [0..mv] else ns := ws;
                     end;

                  end;
            c^.selectors[i] := ns;
            if os<>ns then c^.cflags := c^.cflags + [cfchanged];
            sop(c^.map,clr,c^.map,c^.map);
            end;
         end;

      if qgz in dbug then prt(c,2);
      end;

procedure refunion(*     (c1, c2, c3 : pcomplex);
*)
      ;

(* form the reference union of complexes c2 and c3 *)

var
      i              : integer;

begin
      with pastat^ do c[qrn] := c[qrn]+1;
      for i := 1 to nv do
         c1^.selectors[i] := c2^.selectors[i] + c3^.selectors[i];
      sop(c1^.map,union,c2^.map,c3^.map);
      if qrn in dbug then begin
         write(' refunion: c2');        prtmap(c2^.map);       write(' c3');
         prtmap(c3^.map);               writeln;
         end;
      c1^.cflags := c1^.cflags + [cfchanged];
      if c1<>c2 then genrliz(c1);
      end;

function intrsct(*     (c1, c2      : pcomplex): boolean;
*)
      ;

(* if c1 intersects c2 then true *)

var
      i              : integer;

begin
```



```
      with pastat^ do c[qit] := c[qit]+1;
      i := 0;
      repeat
         i := i+1;
         if (c1^.selectors[i]*c2^.selectors[i])=[] then i := nv+1;
         until (i>=nv);
      if qit in dbug then begin
         writeln(' intrsct: ',i:1);       prt(c1,1);       prt(c2,2);
         end;
      intrsct := i<=nv;
      end;

procedure extend(*     (var rslt   : selmapunit;
      s1,s2       : selmapunit;
      v           : integer);
*)
      ;

(* extend selector s1 against s2, both for variable v *)

var
      l              : integer;
      sl             : pstruct;

begin
      with pastat^ do c[qed] := c[qed]+1;
      case pvariables^[v]^.dtype of

nominal,cyclic:
         rslt := [0..pvariables^[v]^.maxvalue]-s2;

linear:  begin
            l := reflow(s2);
            if reflow(s1)<l then rslt := [0..l-1] else begin
               s2 := s2+[0..l];       rslt := [0..pvariables^[v]^.maxvalue]-s2;
               end;
            end;

structured:
         begin
            rslt := s1;       sl := pvariables^[v]^.structlist;
            while sl<>nil do begin
               if (s1<=sl^.nodedefn) and ([]=s2*sl^.nodedefn) then rslt :=
                  sl^.nodedefn;
               sl := sl^.next;
               end;
            end;
         end;
      if qed in dbug then begin
         write(' extend: ',v:1,' from');       prtset(s1,'=');
         write(' to');                          prtset(rslt,'=');
         write(' against');                     prtset(s2,'=');
         writeln;
         end;
      end;

function degisct(*     (c1, c2      : pcomplex): integer;
```

```
*)
      ;

(* returns the number of selectors which intersect *)
(* selectors in c1 which are dropped are not used  *)

var
      i, j          : integer;

begin
      with pastat^ do c[qdt] := c[qdt]+1;
      j := 0;
      for i := 1 to nv do
 (*      if card(c1^.selectors[i])<=pvariables^[i]^.maxvalue then  *)
         if (c1^.selectors[i]*c2^.selectors[i])<>[] then j := j+1;
      degisct := j;
      if qdt in dbug then writeln(' degisct: ',j:1);
      end;

function bestc(*      (var head    : pcomplex;
      critl       : critlist;
      cn          : integer): pcomplex;
*)
      ;

(* this function scans down the list of complexes from head and *)
(* picks out the one which is best, according to the cost *)
(* the selected complex is removed from the list and a pointer *)
(* to it is returned *)

var
      p,sp,psp       : pcomplex;
      bestcost       : real;
      worscost       : real;
      cst            : real;
      tol            : real;
      limit          : real;
      best           : pcomplex;
      tail           : pcomplex;
      newhead        : pcomplex;

function cselect: pcomplex;

(* pick complexes with cost less than limit from the list, dechain *)
(* them, and present their pointer to caller                       *)

label
      99;

var
      wp             : pcomplex;

begin
      while (sp<>nil) do begin
         if sp^.critvals[cn]<=limit then goto 99;
         psp := sp;       sp := sp^.next;
         end;
```



```
99:   if (sp=nil) then begin
         tail := psp;       cselect := nil;
         end
      else begin
         wp := sp^.next;
         if (psp=nil) then head := wp else psp^.next := wp;
         cselect := sp;       sp^.next := nil;       sp := wp;
         end;
      end;

begin
      with pastat^ do c[qbc] := c[qbc]+1;
      if (head=nil) or (cn>critl.ccnt) then begin
         best := head;            (* give out first complex *)
         if (head<>nil) then begin
            head:=head^.next;       best^.next := nil;
            end;
         end
      else begin

(* find best and worst costs *)

         p := head^.next;       bestcost := head^.critvals[cn];
         worscost := bestcost;
         while (p<>nil) do begin
            cst := p^.critvals[cn];
            if (cst<bestcost) then bestcost := cst;
            if (cst>worscost) then worscost := cst;
            p := p^.next;
            end;

         tol := critl.tlist[cn];
         if tol<0 then tol := -tol*(worscost-bestcost);
         limit := bestcost + tol;

(* move all complexes with costs below limit to new list *)

         sp := head;       psp := nil;       newhead := cselect;
         p := newhead;
         while (p<>nil) do begin
            p^.next := cselect;       p := p^.next;
            end;

(* if only one complex, just return it *)

         if (newhead^.next=nil) then begin
            best := newhead;
            end

(* else invoke bestc to select from among several *)

         else begin
            best := bestc(newhead,critl,cn+1);
            if (tail=nil) then head := newhead else tail^.next := newhead;
            end;
         end;
```



```
      if qbc in dbug then writeln(' bestc: ',ord(best):1,' cn=',cn:1);
      bestc := best;
      end;

procedure trim(*      (q             : integer;
      var clist   : pcomplex;
      critl       : critlist);
*)
      ;

(* trim complexes to q best according to specified criteria *)

var
      p, pp          : pcomplex;          (* pointers to complexes *)
      c              : pcomplex;
      i              : integer;

begin
      with pastat^ do c[qtm] := c[qtm]+1;
      if qtm in dbug then writeln(' trim: ',q:1);

(* make sure crit values are present *)

      p := clist;
      while (p<>nil) do begin
         if cfchanged in p^.cflags then critval(p);
         p := p^.next;
         end;

      i := 0;       c := clist;       pp := nil;
      repeat
         i := i+1;       p := bestc(c,critl,1);
         if (pp=nil) then begin
            pp := p;       clist := p;
            end
         else begin
            pp^.next := p;       pp := p;
            end;
         until (i>=q) or (pp=nil);

(* everything left on c list can be freed *)

      while (c<>nil) do c := freecplx(c);
      end;

function syndist(*      (c1 , c2     : pcomplex ) : real;
*)
      ;

(*computes syntactic distance between two complexes*)

var
      d, dd          : real;              (* sum of syntactic distance *)
      i              : integer;           (* selector incrementor *)

begin
```



```
      with pastat^ do c[qst] := c[qst]+1;
      d := 0.0;
      for i := 1 to nv do case pvariables^[i]^.dtype of

nominal: if c1^.selectors[i] <> c2^.selectors[i] then d:=d+1;

linear:  d:=d+(abs(reflow(c1^.selectors[i])-reflow(c2^.selectors[i]))
   / pvariables^[i]^.maxvalue);

cyclic:  begin
            dd := abs(reflow(c1^.selectors[i])-reflow(c2^.selectors[i]));
            if dd>(pvariables^[i]^.maxvalue/2.0) then dd:=
               pvariables^[i]^.maxvalue - dd;
            d:=d+(2.0 * dd) / pvariables^[i]^.maxvalue;
            end;

structured:
         if c1^.selectors[i] <> c2^.selectors[i] then d:=d+1;

         end;

      if qst in dbug then writeln(' syndist: ',d:1);
      syndist := d;
      end;


(* oper segment *)
(*$I'cluster/2 operational primitives' *)



function newcplx;                    (* get and initialize a complex *)
(*     : pcomplex;
*)

(* returns pointer to new complex or nil *)

var
      i              : integer;
      p              : pcomplex;

begin
      with pastat^ do c[qnx] := c[qnx]+1;
      if postat=nil then begin
         new(postat);       postat^.numfree := 0;       postat^.numgotten := 1;
         postat^.freeclst := nil;
         if nv>maxvars then begin
            writeln('newcplx: nv too large');       halt;
            end;
         new(postat^.cwork);
         end;

      p := postat^.freeclst;
      if (p=nil) then begin
         new(p);       postat^.freeclst := p;
         postat^.numgotten := postat^.numgotten+1;
```

```
            postat^.numfree := postat^.numfree+1;
            p^.next := nil;
            end;
        postat^.freeclst := p^.next;        postat^.numfree := postat^.numfree-1;
        p^.cflags := [cfchanged];           p^.area := 0;
        for i:=0 to numevsets do p^.map[i] := [];
        p^.next := nil;

        if qnx in dbug then writeln(' newcplx: ',ord(p):1);

        newcplx := p;
        end;

function freecplx;                          (* free a complex *)
(*    (c                : pcomplex): pcomplex;
*)

(* places complex c on free list *)

var
        p               : pcomplex;

begin
        with pastat^ do c[qfx] := c[qfx]+1;
        if qfx in dbug then writeln(' freecplx: ',ord(c):1,' ',ord(c^.next):1);

        c^.cflags := c^.cflags + [cffree];
        p := postat^.freeclst;
        while (p<>nil) do if c=p then halt else p:=p^.next;
        freecplx := c^.next;        c^.next := postat^.freeclst;
        postat^.freeclst := c;      postat^.numfree := postat^.numfree+1;
        end;

function numcplx(*          : integer;
*)
        ;

(* returns the number of remaining free complexes *)

begin
        numcplx := postat^.numfree;
        end;

function domof(*        (id: alfa): pdomain;
*)
        ;

(* find domain defn of variable defn of given name *)

var
        d1, d2          : pdomain;
        i               : integer;

begin
        d1 := domains;          d2 := nil;
        while (d1<>nil) and (d2=nil) do begin
            if (d1^.name=id) then d2 := d1;
```



```
            d1 := d1^.domain;
            end;
        if d2=nil then begin
            i := 1;
            while (i<=nv) and (d2=nil) do begin
                d1 := pvariables^[i];
                if (d1^.name=id) then d2:=d1;
                i := i+1;
                end;
            end;
        domof := d2;
        end;

function prtalfa
        (a              : alfa): integer;

(* print only leading content of alfa *)

var
        i, en           : integer;
        c               : char;
        range           : boolean;

begin
        en := alfasize;
        while (en>1) and (a[en]=' ') do en:=en-1;
        range := a[1]='!';
        if range then i:=2 else i:=1;
        while (i<=en) do begin
            c:=a[i];
            if range and (c='!') then write(output,'..') else write(output,c);
            i := i+1;
            end;
        prtalfa:=en;
        end;

function prtval
        (val            : integer;
         v              : integer): integer;

(* print value of selector #v; return num chars written *)

var
        nref            : pname;
        w               : integer;

begin
        if (v>0) and (v<=nv) then nref := pvariables^[v]^.onames else nref := nil
        if nref=nil then begin
            if val>99 then begin
                w := prtval(val div 100, v);        val := val mod 100;
                end
            else w := 0;
            if val>9 then w := w+1;
            write(val:1);       w := w+1;
            end
```



```
        else w := prtalfa(nref^.names[val+1]);
        prtval := w;
        end;

procedure printreal
        (r              : real;
         w              : integer);

var
        i               : integer;

begin
        i := 0;
        if (abs(r)<10000.0) then i := trunc(r);
        if i=r then write(' ',i:(w-1)) else write(r:w);
        end;

procedure prtcrit;                          (* print crit name *)
(*    (n                : integer;
        var chn         : alfa);
*)

var
        i,j             : integer;
        p               : boolean;      (* print request *)
        r               : boolean;      (* right justified *)

begin
        i := n;         p := chn[1]='p';      r := chn[1]='r';
        if i<0 then i := -i;
        if (i>0) and (i<=maxcrit) then case i of
1:          strasgn(chn,'spar       ');
2:          strasgn(chn,'dis        ');
3:          strasgn(chn,'multcov    ');
4:          strasgn(chn,'bal        ');
5:          strasgn(chn,'com        ');
6:          strasgn(chn,'dim        ');
7:          strasgn(chn,'sim        ');
8:          strasgn(chn,'pspars     ');
9:          strasgn(chn,'except     ');
            end
        else strasgn(chn,'unknown    ');
        if p then begin
            if n<0 then write('-') else write(' ');
            write(chn:9);
            end;
        if r then begin
            i := alfasize;
            while (chn[i]=' ') do i:=i-1;
            j := 9-i;
            if j>0 then write(' ':j);
            if n<0 then write('-') else write(' ');
            for j:=1 to i do write(chn[j]);
            end;
        end;

procedure prtset(*      (s              : selmapunit;
```



```
        dlm             : char);
*)
        ;

var
        i               : integer;

begin
        for i := 0 to setsize do if i in s then begin
            write(dlm,i:1);         dlm := ',';
            end;
        end;

procedure prtmap(*      (var m          : evtmap);
*)
        ;

var
        dlm             : char;
        i               : integer;

begin
        dlm := '=';
        for i := 0 to numevsets do if m[i]<>[] then begin
            prtset(m[i],dlm);       dlm := ',';
            end;
        end;

procedure prt(*     (c              : pcomplex;
        n               : integer);
*)
        ;

(* procedure to print a complex *)

var
        i, j            : integer;

begin
        if n>=0 then write(' complex ',n:1,'(',ord(c):1, '-->',ord(c^.next):1,
            '):');
        j := 0;
        with c^ do begin
            write(' area=',area:1,' costs=');
            for i:=1 to maxcrit do begin
                printreal(critvals[i],2);
                end;
            write(' map');          prtmap(map);
            for i := 1 to nv do
                if reflev(selectors[i])<=pvariables^[i]^.maxvalue then begin
                    write(' [x',i:1);          prtset(selectors[i],'=');
                    write('] ');               j := j+1;
                    if 0=(j mod 8) then writeln;
                    end;
            end;
        writeln;
        end;
```

```
procedure prtrlt;               (* print a relational table *)
(*   (tid          : integer;
      tname        : alfa);
*)

(* print a relational table *)

var
      i, j, k, l,m  : integer;
      si            : stattyp;
      dc, nln       : integer;
      tref          : ptitle;
      dref          : pdomain;
      pref          : pparm;
      nref          : pname;
      sref          : pstruct;
      cref          : pcrit;
      eref          : pcomplex;
      f             : set of 0..1;
      c             : char;
      hier          : boolean;
      wted          : boolean;
      a             : alfa;

begin
      writeln;      writeln;      writeln;
      case tid of
1:       writeln('  title');
2:       writeln('  parameters');
3:       writeln('  domains');
4:       writeln('  variables');
5:       begin
            write('  ');      i := prtalfa(tname);      writeln('-names');
            end;
6:       begin
            write('  ');      i := prtalfa(tname);      writeln('-structure');
            end;
7:       begin
            write('  ');      i := prtalfa(tname);      writeln('-criterion');
            end;
8,-8: writeln('  events');
9:       begin
            write('  ');      i := prtalfa(tname);      writeln('-onames');
            end;

10:      writeln('  debug');

         end;

      case tid of

1:       begin                          (* title *)
            writeln('    #   title');      tref := titles;      i := 0;
            while (tref<>nil) do begin
               i := i+1;      write(i:5,' ':2);      f := [];
               for j := 1 to tref^.length do
```



```
                  if tref^.text[j]='''' then f := f+[0]
                  else if tref^.text[j]='"' then f := f+[1];
               if 1 in f then
                  if 0 in f then c:=' ' else c:='''' else c:='"';
               write(c);
               for j := 1 to tref^.length do write(tref^.text[j]);
               writeln(c);      tref := tref^.next;
               end;
            end;

2:       begin                          (* parameters *)
            pref := parameters;      hier := false;
            while (pref<>nil) do begin
               hier := hier or (pref^.covertp=hierarchical);
               pref := pref^.next;
               end;
            write('   mink maxk   trace   h1   h2   h3  initmethod  nidspeed
               'covertype  criterion  base  probe  beta');
            if hier then writeln('  maxheight  minsize') else writeln;
            pref := parameters;
            while (pref<>nil) do begin
               write(pref^.mink:6,pref^.maxk:5,' ':4);
               if pref^.debug=[] then write(' off ')
               else if pref^.debug=[q00] then write(' on  ') else write(' ***
               ;
               write(pref^.h1:5,pref^.h2:5,pref^.h3:5,' ':4);
               if pref^.initmthd=random then write('random',' ':6)
                                        else write('ormeth',' ':6);
               if pref^.nidspeed=slow then write('slow   ')
                                      else write('fast   ');
               case pref^.covertp of
disjoint:         write(' disjoint   ');
intersecting:     write('intersectng ');
hierarchical:     write('hierarchial ');
                  end;
               i := prtalfa(pref^.critname);      write(' ':(11-i));
               write(pref^.base:2,pref^.probe:7,pref^.beta:7:1);
               if hier then writeln(pref^.maxht:8,pref^.minsize:9) else write)
               pref := pref^.next;
               end;
            end;

3:       begin                          (* domains *)
            writeln('    name        type          levels   cost');
            dref := domains;
            while (dref<>nil) do begin
               write(' ':5);      i := prtalfa(dref^.name);
               write(' ':(12-i));
               case dref^.dtype of
nominal:          write('nominal   ');
linear:           write('linear    ');
cyclic:           write('cyclic    ');
structured:       write('structured');
                  end;
               writeln(dref^.maxvalue+1:7,dref^.cost:10);
               dref := dref^.domain;
               end;
```



```
            end;

4:       begin                          (* variables *)
            writeln('    #   type          levels    cost   name');
            for i := 1 to nv do begin
               write(i:5,' ':4);
               case pvariables^[i]^.dtype of
nominal:          write('nominal   ');
linear:           write('linear    ');
cyclic:           write('cyclic    ');
structured:       write('structured');
                  end;
               write(pvariables^[i]^.maxvalue+1:7,pvariables^[i]^.cost:9,
                  '      ');
               j := prtalfa(pvariables^[i]^.name);
               if pvariables^[i]^.domain<>nil then begin
                  write('-');
                  j := 1+j+prtalfa(pvariables^[i]^.domain^.name);
                  end;
               writeln;
               end;
            end;

5,9:     begin                          (*  names *)
            writeln(' value   name');      dref := domains;
            dref := domof(tname);
            if (dref<>nil) then begin
               case tid of
5:                nref := dref^.inames;
9:                nref := dref^.onames;
                  end;
               if (nref<>nil) then begin
                  for i:=1 to dref^.maxvalue+1 do begin
                     write(i-1:5,'      ');      j:=prtalfa(nref^.names[i]);
                     writeln;
                     end;
                  end;
               end;
            end;

6:       begin                          (* structure *)
            dref := domof(tname);
            if (dref<>nil) then begin
               k:=2;      l:=maxint;
               for j:=2 to 6 do begin
                  dc := 0;      sref := dref^.structlist;
                  while (sref<>nil) do begin
                     m := card(sref^.nodedefn);
                     if m<1 then m:=1;
                     nln := (m+j-1) div j;
                     dc := dc + (nln-1) + (nln*j - m);
                     sref := sref^.next;
                     end;
                  if (dc<l) then begin
                     l:=dc;      k:=j;
                     end;
                  end;
```



```
               write(' value  name        ');
               if dref^.onames=nil then for i:=1 to k do write('subvalue    ')
                                   else for i:=1 to k do write('subname     ')
               writeln;      sref := dref^.structlist;
               while (sref<>nil) do begin
                  i:=0;
                  for j:=0 to dref^.maxvalue do begin
                     if j in sref^.nodedefn then begin
                        if i=0 then begin
                           write(sref^.valu:5,'   ');
                           l := prtalfa(sref^.name);
                           write(' ':12-l);
                           end;
                        if dref^.onames=nil then l:=prtval(j,0)
                        else l:=prtalfa(dref^.onames^.names[j+1]);
                        write(' ':12-l);

                        i:=i+1;
                        if i>=k then begin
                           i:=0;      writeln;
                           end;
                        end;
                     end;
                  sref := sref^.next;
                  if i>0 then begin
                     for l:=i+1 to k do write('$',' ':11);
                     writeln;
                     end;
                  end;
               end;
            end;

7:       begin                          (* criterion*)
            writeln('    #   criterion  tolerance');      cref := criteria;
            while (cref<>nil) do begin
               if cref^.name=tname then begin
                  for i:=1 to cref^.ccnt do begin
                     write(i:5,'   ');      a[1]:='p';
                     prtcrit(cref^.clist[i],a);
                     writeln(cref^.tlist[i]:6:2);
                     end;
                  end;
               cref := cref^.next;
               end;
            end;

8,-8: begin                             (* events *)
            write('    #  ');      eref := events;      wted := false;
            while (eref<>nil) and not wted do begin
               wted := eref^.cost>1;      eref := eref^.next;
               end;
            if wted then write('  wt    ');
            if tid=8 then i:=0 else i:=postat^.eventcol;
            m := 0;      l := i+1;
            while (i<nv) and (m<115-alfasize) do begin
               i := i+1;      j := prtalfa(pvariables^[i]^.name);
               write(' ':(pvariables^[i]^.nwidth+2-j));
```

```
            m := m + pvariables^[i]^.awidth+2;
            end;
         postat^.eventcol := 1;       writeln;       j := 0;
         eref := events;
         while (eref<>nil) do begin
            j := j+1;        write(j:5,'   ');
            if wted then write(eref^.cost:3,' ':5);
            for i := 1 to postat^.eventcol do begin
               if eref^.selectors[i]<>[] then k :=
                  prtval(reflow(eref^.selectors[i]),i)
               else begin
                  k := 1;       write(output,'$');
                  end;
               write(' ':(pvariables^[i]^.awidth+2-k));
               end;
            eref := eref^.next;       writeln;
            end;
         if postat^.eventcol < nv then prtrlt(-@,tname);
         end;

10:      begin                         (* debug *)
            writeln('   #  name');        i := 0;
            for si:=succ(q00) to qlast do begin
               if si in dbug then begin
                  i := i+1;        statnam(si,a);        writeln(i:4,'  ',a);
                  end;
               end;
            end;

         end;

      writeln;
      end;

procedure mop;                (* set operations extended to an array of sets *)
(* five set operations are implemented on an array of sets.  the only reason
   for this is the cyber pascal limitation on set size.  a very large set
   is implemented as an array of small sets *)
(*   (var m1: evtmap;
      op          : mopcd;
      var m2, m3  : evtmap);
*)

var
      i              : integer;

begin
      case op of
clr:     for i :=0 to numevsets do m1[i]  := [];
all:     for i :=0 to numevsets do m1[i]  := [0..setsize];
union:   for i :=0 to numevsets do m1[i]  := m2[i] + m3[i];
diff:    for i :=0 to numevsets do m1[i]  := m2[i] - m3[i];
inter:   for i :=0 to numevsets do m1[i]  := m2[i] * m3[i];
         end;
      end;
```



```
procedure reduce;      (* reduce sparseness in a complex *)
(*    (c              : pcomplex);
*)

var
      p              : pcomplex;
      cw             : pcomplex;
      i              : integer;

begin
      with pastat^ do c[qre] := c[qre]+1;
      if qre in dbug then writeln(' reduce:');

      (* a complex is reduced by taking the refunion of all covered events *)
      p := events;       cw := postat^.cwork;

      (* start with an empty complex *)
      mop(cw^.map,clr,cw^.map,cw^.map);
      for i := 1 to nv do cw^.selectors[i] := [];

      while p<>nil do begin
         (* expand the complex to cover each covered event *)
         if tcover(p,c) then refunion(cw,cw,p);
         p := p^.next;
         end;

      (* replace old complex with new complex *)
      c^.map := cw^.map;
      for i := 1 to nv do c^.selectors[i]  = cw^.selectors[i];

      (* generalize complex minimally *)
      genrliz(c);
      end;

procedure setmap;                  (* build event map for a complex *)
(*    (c              : pcomplex);
*)

var
      p              : pcomplex;
      lmap           : evtmap;

begin
      with pastat^ do c[qsp] := c[qsp]+1;
      if qsp in dbug then writeln(' setmap:');

      (* clear event map of covered events *)
      mop(lmap,clr,lmap,lmap);

      (* check each event to see if covered *)
      p := events;
      while p<>nil do begin
         if tcover(p,c) then mop(lmap,union,lmap,p^.map);
         p := p^.next;
         end;

      (* save map of covered events in complex *)
```



```
      c^.map := lmap;

      (* if complex has changed, evaluate all criteria in lef *)
      if cfchanged in c^.cflags then critval(c);
      end;

function mcard;               (* calculate cardinality of an array of sets *)
(*    (var map       : evtmap): integer;
*)

var
      i, j           : integer;

begin
      with pastat^ do c[qmd] := c[qmd]+1;

      (* sum cardinalities of all sets into j *)
      j := 0;
      for i := 0 to numevsets do j := j+card(map[i]);
      mcard := j;

      if qmd in dbug then writeln(' mcard: ',j:1);
      end;

procedure setlevelmap; (* build event map of events covered by any cluster *)
(*    (p             : pcomplex);
*)

begin
      with pastat^ do c[qsq] := c[qsq]+1;
      if qsq in dbug then writeln(' setlevelmap:');

      (* clear levelmap *)
      mop(levelmap,clr,levelmap,levelmap);

      (* run down list of clusters and collect union of covered events *)
      while p<>nil do begin
         if mcard(p^.map)=0 then setmap(p);
         mop(levelmap,union,levelmap,p^.map);       p := p^.next;
         end;
      end;

function numsel;         (* compute number of selectors in a complex *)
(*    (c             : pcomplex): integer;
*)

var
      i, r           : integer;

begin
      with pastat^ do c[qnl] := c[qnl]+1;

      (* count non-dropped selectors in r *)
      r := 0;
      for i := 1 to nv do
         if card(c^.selectors[i]) <= pvariables^[i]^.maxvalue then r := r+1;
      numsel := r;
```



```
      end;

procedure massgn;     (* add an event to an event map *)
(*    (var m          : evtmap;
       b              : integer);
*)

var
      eset, ebit    : integer;

begin
      (* determine which set in the set array contains the event.
         then add that event to that set *)
      eset := b div (setsize+1);        ebit := b-(eset*(setsize+1));
      m[eset] := m[eset] + [ebit];
      end;

function min;              (* determine if an event is in an event set *)
(*    (e              : integer;
       var m          : evtmap): boolean;
*)

var
      eset, ebit    : integer;

begin
      (* calculate which set the event may be in; check if there *)
      eset := e div (setsize+1);
      if eset > numevsets then min := false else begin
         ebit := e-(eset*(setsize+1));        min := ebit in m[eset];
         end
      end;

function mcomp;     (* compare two event sets *)
(*    (var m1        : evtmap;
       t             : mcops;
       var m2        : evtmap): boolean;
*)

label
      888,
      999;

var
      i              : integer;
      b              : boolean;

begin
      b := false;
      case t of
ne:      begin
            for i := 0 to numevsets do if m1[i]<>m2[i] then goto 888;
            goto 999;
            end;
le:      for i := 0 to numevsets do if not (m1[i]<=m2[i]) then goto 999;
eq:      for i := 0 to numevsets do if not (m1[i] =m2[i]) then goto 999;
ge:      for i := 0 to numevsets do if not (m1[i]>=m2[i]) then goto 999;
```

```
	end;
888:	b := true;
999:	mcomp := b;
	end;

function maplow;	(* determine lowest event number in event set *)
(*	(var m		: evtmap): integer;
*)

label
	998;

var
	i		: integer;

begin
	(* find lowest array set that is not empty *)
	for i := 0 to numevsets do if m[i]<>[] then goto 998;
998:	maplow := reflow(m[i]) + ((setsize+1)*i);
	end;

function wmcard;	(* computed weighted event set cardinality *)
(*	(var map:  evtmap;
	 op:  integer): integer;
*)

var
	w		: integer;
	p		: pcomplex;

begin
	(* if op<>0 this means to note whether to use weighted calculations *)
	if op<>0 then begin
	   (* if op is positive, use weighted calculations *)
	   postat^.wted := op>1;
	   end
	(* else if unweighted, use plain mcard to get cardinality *)
	else if not postat^.wted then wmcard := mcard(map) else begin
	   (* else form sum of weights of covered events *)
	   w := 0;	p := events;
	   while (p<>nil) do begin
	      if mcomp(p^.map,le,map) then w := w + p^.cost;
	      p := p^.next;
	      end;
	   wmcard := w;
	   end;
	end;

function iabs;		(* integer absolute value *)
(*	(v: integer): integer;
*)

begin
	if v<0 then iabs := -v else iabs := v;
	end;

procedure statnam;	(* generate the name for a usage counter *)
```



```
begin
     case i of
q00:	strasgn(n,'         **');	(* trace on indicator *)
qcd:	strasgn(n,'cendist    ');	(* cendist *)
qcl:	strasgn(n,'critval    ');	(* critval *)
qcr:	strasgn(n,'cluster    ');	(* cluster *)
qcv:	strasgn(n,'capcov     ');	(* capcov *)
qed:	strasgn(n,'extend     ');	(* extend *)
qfx:	strasgn(n,'freecplx   ');	(* freecplx *)
qgz:	strasgn(n,'genrliz    ');	(* genrliz *)
qit:	strasgn(n,'intrsct    ');	(* intrsct *)
qnd:	strasgn(n,'nid        ');	(* nid *)
qnl:	strasgn(n,'numsel     ');	(* numsel *)
qnx:	strasgn(n,'newcplx    ');	(* newcplx *)
qre:	strasgn(n,'reduce     ');	(* reduce *)
qrh:	strasgn(n,'refhigh    ');	(* refhigh *)
qrm:	strasgn(n,'refnum     ');	(* refnum *)
qrn:	strasgn(n,'refunion   ');	(* refunion *)
qrv:	strasgn(n,'reflev     ');	(* reflev *)
qrw:	strasgn(n,'reflow     ');	(* reflow *)
qsp:	strasgn(n,'setmap     ');	(* setmap *)
qsq:	strasgn(n,'setlevelma');	(* setlevelmap *)
qsr:	strasgn(n,'star       ');	(* star *)
qss:	strasgn(n,'semantics  ');	(* semantics *)
qst:	strasgn(n,'syndist    ');	(* syndist *)
qtm:	strasgn(n,'trim       ');	(* trim *)
qtr:	strasgn(n,'tcover     ');	(* tcover *)
qdt:	strasgn(n,'degisct    ');	(* degisct *)
qcg:	strasgn(n,'clustering');	(* clustering *)
qtp:	strasgn(n,'tablesetup');	(* tablesetup *)
qbc:	strasgn(n,'bestc      ');	(* bestc *)
qmd:	strasgn(n,'mcard      ');	(* mcard *)
qgh:	strasgn(n,'genpath    ');	(* genpath *)
qcq:	strasgn(n,'clearcv    ');	(* clearcv *)
qsv:	strasgn(n,'savecv     ');	(* savecv *)
	end;
     end;

(* main segment *)
(*$1'cluster/2 main routines' *)


		(* include global defns *)


procedure nid	(* make non-disjoint complexes disjoint *)
     (cv		: pcover);

var
	i, j, cnt	: integer;
	nidcrit		: integer;
	m1		: evtmap;		(* work map *)
	m2		: evtmap;		(* events needing assignment *)
	m3		: evtmap;		(* union of covered events *)
	mt		: evtmap;
	wcplx		: array[1..maxk] of pcomplex;
```



```
	sparseness	: array[1..maxk] of real;
	nosel		: array[1..maxk] of real;
	c, p		: pcomplex;
	tempflgs	: comflgs;

function elim	(* eliminate less optimal event host complexes *)
     (cr		: integer): integer;

var
	i		: integer;
	mincost		: real;
	deltacost	: array[1..maxk] of real;
	c		: pcomplex;

begin
	mincost := 1e99;	elim := -1;

	(* compute delta costs for all complexes *)
	for i := 1 to clstrs do begin
	   c := wcplx[i];
	   (* if marknid is true, the complex has already been eliminated *)
	   if not (cfmarknid in c^.cflags) then with c^ do begin
	      (* measure complex on given nid criterion *)
	      case cr of
1:		begin		(* delta sparseness *)
		   deltacost[i] := area - mcard(map) - sparseness[i];
		   end;

2:		begin		(* # events placed *)
		   mop(mt,inter,map,m2);	deltacost[i] := - mcard(mt);
		   end;

3:		begin		(* delta number of selectors *)
		   deltacost[i] := numsel(c) - nosel[i];
		   end;

4:		cnt := 0;	(* cause end of judging *)

		end;

	      (* find minimum cost *)
	      if deltacost[i] < mincost then mincost := deltacost[i];
	      end;
	   end;

	(* eliminate all host complexes that score worse than the best *)
	for i := 1 to clstrs do if not (cfmarknid in wcplx[i]^.cflags) then
	   if deltacost[i] > mincost then begin
	      (* mark complex 'eliminated' *)
	      wcplx[i]^.cflags := wcplx[i]^.cflags + [cfmarknid];
	      cnt := cnt-1;
	      end
	   else elim := i; (* remember one of the best ones *)
	end;

begin (* nid *)
	with pastat^ do c[qnd] := c[qnd]+1;
```



```
	if qnd in dbug then begin
	   writeln(' nid: before');
	   for i := 1 to clstrs do prt(cv^.pcluster[i],1);
	   end;

	(* clear work event sets *)
	mop(m2,clr,m2,m2);	mop(m3,clr,m3,m3);	context := nil;

	(* process each cluster, collecting all events in set m3 and all
	   multiply-covered events in set m2 *)
	for i := 1 to clstrs do begin
	   m1 := cv^.pcluster[i]^.map;	mop(m3,union,m3,m1);
	   cv^.pcluster[i]^.next := context;
	   context := cv^.pcluster[i];
	   for j := i+1 to clstrs do begin
	      mop(mt,inter,m1,cv^.pcluster[j]^.map);	mop(m2,union,m2,mt);
	      end;
	   end;

	(* add to m2 those events not covered at all *)
	mt := allevents;	mop(mt,diff,mt,m3);	mop(m2,union,m2,mt);

	(* check each complex again.  form core complexes for those whose
	   events intersects the set of multiply-covered events *)
	for i := 1 to clstrs do begin
	   (* first make a copy of each complex *)
	   c := cv^.pcluster[i];	p := newcplx;	tempflgs := p^.cflags;
	   p^ := c^;	p^.cflags := tempflgs;	cv^.pcluster[i] := p;  c := p;

	   (* check if any bad events *)
	   mop(mt,inter,c^.map,m2);
	   if mcard(mt)<>0 then with c^ do begin
	      mop(map,diff,map,m2);
	      (* clear the complex *)
	      for j := 1 to nv do selectors[j] := [];

	      (* reduce the complex to just singly-covered events *)
	      p := events;
	      while p<>nil do begin
	         if mcomp(p^.map,le,map) then refunion(c,c,p);
	         p := p^.next;
	         end;
	      genrliz(c);
	      end;

	   (* evaluate the LEF criteria: recompute the sparseness... *)
	   critval(c);	sparseness[i] := c^.area - mcard(c^.map);
	   (* and number of selectors *)
	   nosel[i] := numsel(c);
	   end;

	if qnd in dbug then begin
	   write(' nid: bad events');	prtmap(m2);	writeln;
	   end;

	(* get k host complexes to manipulate *)
	for i := 1 to clstrs do wcplx[i] := newcplx;
```

```
      (* process each event in the multiply-covered set *)
      p := events;
      while (mcard(m2)>0) and (p<>nil) do begin
         (* check if in multiply-covered set *)
         if mcomp(p^.map,le,m2) then begin

            (* process a multiply-covered event *)
            cnt := 0;

            (* add the event to each complex to form potential hosts *)
            for i := 1 to clstrs do begin
               c := wcplx[i];      refunion(c,cv^.pcluster[i],p);
               context := nil;
               (* form a clustering with a particular host complex *)
               for j := 1 to clstrs do begin
                  if i=j then begin
                     c^.next := context;      context := c;
                     end
                  else begin
                     cv^.pcluster[j]^.next := context;
                     context := cv^.pcluster[j];
                     end;
                  end;
               critval(c);  (* evaluate LEF *)
               c^.cflags := c^.cflags - [cfmarknid];

   (* at this point there is a potential clustering containing one host
      complex.  the clustering is the list of complexes linked via context *)

               (* check to see of any complexes intersect *)
               j := 0;
               repeat
                  j := j+1;
                  if j<>i then
                     if intrsct(c,cv^.pcluster[j]) then j:= clstrs+1;
                  until j>=clstrs;

               (* if the host intersects another complex, eliminate it *)
               if j>clstrs then c^.cflags := c^.cflags + [cfmarknid];
               if not (cfmarknid in c^.cflags) then cnt:=cnt+1;
               end;

            (* if no host was ever satisfactory, move event to exceptions set *)
            if cnt=0 then begin
               if qnd in dbug then writeln(' nid failed to place event ',
                  (1+maplow(p^.map)):1);
               mop(m2,diff,m2,p^.map);
               mop(cv^.exception,union,cv^.exception,p^.map);
               end
            else begin
               (* apply 3 nid criteria until a single best host is identified *)
               nidcrit := 0;
               repeat nidcrit := nidcrit+1; j := elim(nidcrit); until cnt<2;

               (* exchange host complex for original complex *)
               c := wcplx[j];      wcplx[j] := cv^.pcluster[j];
```



```
               cv^.pcluster[j] := c;

               (* store complexes in cover *)
               context := nil;      level := clstrs;
               for i := 1 to clstrs do begin
                  cv^.pcluster[i]^.next := context;
                  context := cv^.pcluster[i];
                  end;

               (* reevaluate criteria on host complex *)
               c^.cflags := c^.cflags + [cfchanged];      critval(c);
               sparseness[j] := c^.area - mcard(c^.map);
               nosel[j] := numsel(c);

               (* remove event placed into host from multiply-covered set *)
               mop(m2,diff,m2,c^.map);
               end;
            end;
         p := p^.next;
         end;

      if mcard(m2)>0 then writeln(' nid: mcard error');

      (* since context has changed, reevaluate all criterion *)
      for i:=1 to clstrs do critval(cv^.pcluster[i]);

      if qnd in dbug then begin
         writeln(' nid: after');
         for i := 1 to clstrs do prt(cv^.pcluster[i],1);
         end;

      (* return to free pool the k work complexes *)
      for i := 1 to clstrs do p := freecplx(wcplx[i]);

      end;

procedure prtstate;

var
      si              : stattyp;
      a               : alfa;
      ll              : integer;

begin
      writeln;        writeln(' activity statistics');     ll := 0;
      for si := succ(q00) to pred(qlast) do begin
         ll := ll + 20;
         if ll>lnelength then begin
            ll := 0;  writeln;
            end;
         statnam(si,a);        write('  ',a:10,'=',pastat^.c[si]:6);
         pastat^.c[si] := 0;
         end;
      writeln;
      end;

procedure mane;         (* main program *)
```



```
var
      p, q, pp       : pcomplex;        (* pointers to complexes *)
      pref           : ppara;           (* pointer to parameters data *)
      cv, cv2        : pcover;          (* pointers to covers *)
      i              : integer;
      si             : stattyp;         (* statistics data index variable *)
      rt             : seedtype;        (* indicates central or distant seeds *)
      citer          : integer;         (* clustering iteration count *)
      btries, ptries: integer;          (* no. of remaining base and probe tries *)
      cptr           : pcrit;           (* pointer to a lef definition *)
      fcrit          : pcrit;           (* pointer to current lef definition *)
      tptr           : ptitle;          (* pointer to title data *)
      cvector        : array[1..maxbase] of pcover;  (* solution set *)
      numcv          : integer;         (* number of covers so far *)
      clk1, clk2     : integer;         (* clock values *)
      expnum         : integer;         (* experiment (parameter line) number *)
      msgtext        : record           (* text used by prtbmsg *)
                       case boolean of
false:  (la             : lalfa);
true:   (al             : alfa);
                       end;
      mwork          : alfa;            (* msgtext work variable *)
      cpxlist        : pcplxlist;       (* pointer to master event list *)
      cpxl           : pcplxlist;       (* pointer to element of mast. evt. list *)
      hhead          : pcover;          (* head of a hierarchical solution *)
      condensed      : boolean;         (* indicates event set was condensed *)

procedure cluster                       (* form a k-clustering *)
      (cv            : pcover);
      (* cv is the cover data area into which the complexes in the
         k-clustering are placed *)

label 88;    (* quick exit if genpath overflows tree storage *)

var
      i, j           : integer;
      w              : integer;         (* current pathsum value being explored *)
      d              : integer;
      pathsum        : integer;         (* sum of ranks along search path *)
      freenode       : integer;         (* index to free search tree node *)
      tries          : integer;         (* number of paths left to explore *)
      tol            : real;            (* lef tolerance value *)
      c              : pcomplex;        (* pointer to a complex *)
      path           : array[1..maxk] of integer; (* current search path *)
      nodelinks      : packed array[treeidx] of treeidx; (* search tree links *)
      nodecplx       : packed array[treeidx] of pcomplex;
                       (* the complex at each node *)
      lmap           : evtmap;
      cmin, cmax: array[0..maxcrit] of real;
                       (* criteria information *)
      ci             : integer;         (* criteria index *)
      best           : boolean;         (* indicates best cltstering found *)
      bestcv         : cover;           (* the best cover *)

function g:npath: boolean; (* fill in search tree along a given path *)
```



```
(* given a path vector (path) this procedure tries to trace that path
   through the search tree.  where nodes do not exist, star is called
   to create them.  on output, the search tree contains all stars needed
   to take the given path and the set of complexes along the path is
   noted in the linked list starting from 'context'.  if the space for
   the search tree is exhausted, false is returned, else true is returned *)

label 77;    (* immediate exit if tree overflow *)

var
      i, j           : integer;
      arity          : integer;         (* the branching factor at a node *)
      np             : integer;         (* tree node pointer *)
      pathsum        : integer;         (* path index sum *)
      c, cc          : pcomplex;        (* pointers to complexes *)

begin
      with pastat^ do c[qgh] := c[qgh]+1;

      (* start from top of tree... *)
      np:=0;       context := nil;       pathsum := 0;       cvcontext := cv;
      (* clear map of covered events *)
      mop(levelmap,clr,levelmap,levelmap);                      i := clstrs;

      (* process each level of the tree, starting with level 1 *)
      repeat
         level := i+clstrs-1;
         (* check if tree already developed this far *)
         if nodelinks[np]=0 then begin
            (* tree not developed, so add a node here.
               arity based on pathsum skews tree to best side *)
            arity := 1+pastat^.h2-pathsum;

            (* the links below this node represent complexes in a star
               of the seed event for this level of the tree (seed i)
               against all other seeds (given in cover cv).  the partial
               stars should be trimmed to h1 number of complexes.  the
               final star should report out a maximum of 'arity' number
               of complexes.  the output from star is a linked list of
               at most 'arity' number of complexes *)
            c := star(i,cv,pastat^.h1,arity);

            (* link search tree nodes *)
            nodelinks[np] := freenode;
            cc := c;
            for j := 1 to arity do begin
               (* place each complex into search tree *)
               nodelinks[freenode] := 0;       nodecplx[freenode] := cc;
               freenode := freenode + 1;
               if (freenode > treesize) then begin
                  writeln; writeln(' search tree exceeds internal storage');
                  pastat^.h2 := (pastat^.h2+1) div 2;
                  writeln(' parameter h2 is reset to ',pastat^.h2:1);
                  genpath := false;
                  goto 77;
                  end;
```

```
            (* mark each complex as gotten by genpath and advance to
               next complex in the star.  if the star contains fewer
               that arity number of complexs, the last complex pointers
               under this node in the tree are nil *)
            if cc<>nil then begin
               cc^.cflags := cc^.cflags + [cfintree];       cc := cc^.next;
               end;
            end;
         end;

      (* the tree node pointer is adjusted to the node that is next
         along the path whose indices are in the 'path' vector.
         complexes (one from each level) are linked together into
         a chain headed at 'context'.  if any nil complex is found
         in the tree, 'context' becomes nil too. *)
      np := nodelinks[np] + path[i];        cc := nodecplx[np];
      if cc<>nil then cc^.next := context;
      context := cc;       pathsum := pathsum + path[i];       i := i-1;
      until (context=nil) or (i=0);
   (* genpath ends with either context=nil (the indicated path cannot
      be completed) or with a list of k complexes in context, one
      complex from each level of the tree.  this list is a k-clustering *)

   if qgh in dbug then begin
      writeln(' genpath:');       cc := context;       i := 1;
      while (cc<>nil) do begin
         prt(cc,path[i]);       cc := cc^.next;       i := i+1;
         end;
      end;
   genpath := true;
77:   begin end;  (* jump here if tree overflow *)
   end;

begin (* cluster *)
   with pastat^ do c[qcr] := c[qcr]+1;
   best := true;   (* set for first time *)
   for i := 1 to clstrs do bestcv.pcluster[i] := nil;

   (* empty the search tree and exceptional event list *)
88:   freenode := 1;       nodelinks[0] := 0;       w := 0;
   sop(cv^.exception,clr,lmap,lmap); (* clear exception list *)

   (* the number of paths explored is set to twice h3.
      note: h3 gives the 'search probe' value.  here 'search base'
            is also taken from parameter h3 *)
   tries := 2*parameters^.h3;                         cv^.clstrs := clstrs;

   (* reset lef evaluation sums, max, min *)
   for i := 1 to fcrit^.ccnt do begin
      cmax[i]:=-1e99;       cmin[i]:=1e99;       cv^.critvals[i] := 1e99;
      end;

   (* go explore paths in the search tree in pathsum order *)
   repeat
      (* clear path indicators *)
      for i := 1 to clstrs do path[i] := 0;
      pathsum := 0;
```



```
      (* call genpath for all paths with sum equal w value *)
      while (pathsum<=w) and (tries>=0) do begin
         path[1] := w - pathsum;
         if not genpath then goto 88; (* jump back if overflow *)

         (* evaluate k-clustering if one was produced by genpath *)
         if context<>nil then begin
            (* copy cover to cv area *)
            c := context;   level := clstrs;
            for i := 1 to clstrs do begin
               c^.cflags := c^.cflags + [cfchanged];  critval(c);
               cv^.pcluster[i] := c;  c := c^.next;
               end;

            (* check best cover to see of it is disjoint and covers all
               events. if it does not cover all events, invoke nid *)
            sop(lmap,clr,lmap,lmap);       j := 0;
            for i := 1 to clstrs do with cv^.pcluster[i]^ do begin
               sop(lmap,union,lmap,map);       j := j + scard(map);
               end;
            if (scard(lmap)<>numevents) or (j<>numevents) then begin
               (* use nid to make disjoint *)
               nid(cv);
               context := nil;       level := clstrs;
               for i := 1 to clstrs do begin
                  c := cv^.pcluster[i];       c^.next := context;
                  context := c;
                  end;
               end;
            for i := 0 to fcrit^.ccnt do cv^.critvals[i] := 0;
            for j := 1 to clstrs do begin
               c := cv^.pcluster[j];
               for i := 0 to fcrit^.ccnt do
                  cv^.critvals[i] := cv^.critvals[i]+c^.critvals[i];
               end;
            (* special processing for counting no. of exceptional events *)
            for i := 1 to fcrit^.ccnt do begin
               if fcrit^.clist[i]=9 then
                  cv^.critvals[i] := scard(cv^.exception);
               end;
            (* find max and min score for each criterion *)
            for i := 1 to fcrit^.ccnt do begin
               if cv^.critvals[i]<cmin[i] then cmin[i]:=cv^.critvals[i];
               if cv^.critvals[i]>cmax[i] then cmax[i]:=cv^.critvals[i];
               end;

            (* check this clustering against a saved best one (in bestcv)
               and set 'best' if this one is better than the saved one *)
            ci := 1;
            while (not best) and (ci<=fcrit^.ccnt) do begin
               tol := fcrit^.tlist[ci];
               if tol<1 then tol := tol*(cmin[ci]-cmax[ci]);
               if cv^.critvals[ci] > bestcv.critvals[ci]-tol
                  then ci:=fcrit^.ccnt
               else best := cv^.critvals[ci] < bestcv.critvals[ci]-tol;
               ci := ci+1;
```



```
               end;

            (* if a best one is found, then set tries for h3 more
               iterations and copy the clustering to 'bestcv' *)
            if best then begin
               best := false;
               if qcr in dbug then writeln(' cluster: best');
               if tries < parameters^.h3 then tries := parameters^.h3;
               (* free old complexes *)
               for i := 1 to clstrs do begin
                  c := bestcv.pcluster[i];
                  if c<>nil then if not (cfintree in c^.cflags)
                     then c := freecplx(c);
                  end;
               bestcv := cv^;
               for i := 1 to clstrs do begin
                  if qcr in dbug then prt(bestcv.pcluster[i],i);
                  end;
               end;
            end;

         (* after one path is generated, the path is updated to another
            one with the same pathsum of 'w' *)
         d := 2;       path[d] := path[d] + 1;       pathsum := pathsum + 1;
         while (pathsum>w) and (d<clstrs) do begin
            pathsum := pathsum - path[d] + 1;       path[d] := 0;
            d := d+1;                               path[d] := path[d]+1;
            end;

         (* this completes one 'try' *)
         tries := tries-1;
         end; (* while *)

      w := w+1;  (* advance to paths with pathsum one higher *)
      until (tries<0) or (w>pastat^.h2);

   (* complexes from stars that were not used are freed.  because it
      is computationally inefficient to check each complex to see if
      it matches a complex in the resulting k-clustering (cv), a
      marking scheme is used.  the k final complexes are marked by
      setting markgnod to true.  then one pass over the entire tree
      is used to remove all unmarked complexes *)
   cv^ := bestcv;
   context := nil;
   for i := 1 to clstrs do with cv^.pcluster[i]^ do begin
      cflags := cflags - [cfintree];       next := context;
      context := cv^.pcluster[i];
      end;
   for i := 1 to freenode-1 do begin
      c := nodecplx[i];
      if c<>nil then
         if cfintree in c^.cflags then c := freecplx(c);
      end;
   end;


procedure dispruls                    (* display a clustering *)
```



```
      (cv            : pcover;
       cp            : pcrit;
       prtflag       : integer);

(* calculate values to active criteria *)

const
       bufsize       = 2600;

var
       cnum          : integer;
       vlbuf, mapbuf: packed array[1..bufsize] of char;
       vlidx, mapidx: integer;
       vlcols, mapcols: integer;
       i, j, k, l    : integer;
       p             : pcomplex;
       sref          : pstruct;
       dlm           : char;
       nwrk, a       : alfa;
       ds            : selmapunit;

procedure addchr
      (ch            : char);

(* place char into vlbuffer *)

begin
       vlidx := vlidx + 1;       vlbuf[vlidx] := ch;
       end;

procedure addnum
      (v             : integer);

(* place value into vlbuffer *)

begin
       if v>9 then begin
          addnum(v div 10);       v := v mod 10;
          end;
       addchr(chr(ord('0')+v));
       end;

procedure addval
      (v             : integer;
       x             : integer;
       t             : integer);

var
       nref          : pname;
       sref          : pstruct;
       i, en         : integer;
       nam           : alfa;

begin
       nref := pvariables^[x]^.onames;
       if nref=nil then addnum(v) else begin
          en := alfasize;
```

```
      if (pvariables^[x]^.dtype=structured) and (v>pvariables^[x]^.maxvalue)
         then begin
         sref := pvariables^[x]^.structlist;        strasgn(nam,'?          ');
         while (sref<>nil) do begin
            if sref^.valu=v then begin
               nam:=sref^.name;        sref:=nil;
               end
            else sref:=sref^.next;
            end;
         end
      else nam:=sref^.names[v+1];
      while (nam[en]=' ') and (en>1) do en:=en-1;
      if nam[1]<>'!' then for i:=1 to en do addchr(nam[i]) else begin
         i:=2;
         while (i<=en) and (nam[i]<>'!') do begin
            if (t<1) then addchr(nam[i]);
            i:=i+1;
            end;
         if (t=0) then begin
            addchr('.');        addchr('.');
            end;
         if (t>-1) then begin
            i:=i+1;
            while (i<=en) do begin
               addchr(nam[i]);        i:=i+1;
               end;
            end;
         end;
      end;
   end;

procedure disppart
      (list           : evtmap);

var
      j               : integer;
      dlm             : char;

begin
      vlidx := 0;
      for j := 1 to bufsize do vlbuf[j] := ' ';
      dlm := '=';
      for j := 0 to maxevents do if min(j,list) then begin
         if dlm=',' then addchr(dlm);
         dlm := ',';        addnum(j+1);
         end;
      mapbuf := vlbuf;        mapidx := vlidx;
      end;

procedure dprtbufs;

var
      j, k            : integer;

begin
      j:=mapidx+mapcols-1;
      while (mapbuf[j]<>' ') and (mapbuf[j]<>',') and (j>mapidx) do j:=j-1;
```



```
      for k := mapidx to j do write(mapbuf[k]);
      mapidx := j+1;        writeln;

(* write additional lines *)
      while (vlbuf[vlidx]<>' ') or (mapbuf[mapidx]<>' ') do begin
         write(' ':11);
         if vlbuf[vlidx]=' ' then write(' ':vlcols) else begin
            j:=vlidx + vlcols-1;
            while (vlbuf[j]<>',') and (vlbuf[j]<>']')
               and (vlbuf[j]<>' ') and (j>vlidx) do j:=j-1;
            if j<=vlidx then j := vlidx + vlcols-1;
            for k:=vlidx to j do write(vlbuf[k]);
            k := vlidx+vlcols-1-j;
            if k>0 then write(' ':k);
            vlidx := j+1;
            end;
         write(' ':(8+10*cnum));

         if mapbuf[mapidx]<>' ' then begin
            j:=mapidx+mapcols-1;
            while (mapbuf[j]<>' ') and (mapbuf[j]<>',') and (j>mapidx) do j:=;
            for k := mapidx to j do write(mapbuf[k]);
            mapidx := j+1;
            end;
         writeln;
         end;
      end;

begin (* disprule *)
      cnum := cp^.ccnt;        vlcols := (lnelength-22-10*cnum) div 2;
      mapcols := vlcols;
      if prtflag>1 then begin(* write header *)
         write(' iter/cplx#  vl-rule',' ':(vlcols-8),' seed ');
         j := (10*cnum-6) div 2;
         for i := 1 to j do write('-');
         write('costs');     j := 10*cnum -6 - j;
         for i := 1 to j do write('-');
         writeln(' events covered');        write(' ':(vlcols+18));
         for i := 1 to cnum do begin
            a[i]:='r';                    prtcrit(cp^.clist[i],a);
            end;
         writeln;
         end;(* end of header *)

      for i := 1 to cv^.clstrs do begin
         p := cv^.pcluster[i];        disppart(p^.map);
         for j := 1 to bufsize do vlbuf[j] := ' ';
         vlidx := 0;

         for j := 1 to nv do
            if reflev(p^.selectors[j])<=pvariables^[j]^.maxvalue then begin
               addchr('[');        k:=alfasize;
               nwrk := pvariables^[j]^.name;
               while (k>1) and (nwrk[k]=' ') do k:=k-1;
               for l:=1 to k do addchr(nwrk[l]);
               dlm:='=';
               case pvariables^[j]^.dtype of
```



```
nominal,cyclic:    begin
                      for k := 0 to pvariables^[j]^.maxvalue do
                         if k in p^.selectors[j] then begin
                            addchr(dlm);        dlm:=',';        addval(k,j,0);
                            end;
                      end;

linear:            begin
                      addchr(dlm);
                      if reflev(p^.selectors[j])=1 then begin
                         addval(reflow(p^.selectors[j]),j,0);
                         end
                      else begin
                         addval(reflow(p^.selectors[j]),j,-1);        addchr('.');
                         addchr('.');
                         addval(refhigh(p^.selectors[j]),j,1);
                         end;
                      end;

structured:        begin
                      if reflev(p^.selectors[j])>1 then begin
                         sref := pvariables^[j]^.structlist;
                         ds := [0..pvariables^[j]^.maxvalue];
                         k := maxint;
                         while (sref<>nil) and (k=maxint) do begin
                            if (sref^.nodedefn=p^.selectors[j]) then k :=
                               sref^.valu;
                            if ((ds-sref^.nodedefn)=p^.selectors[j]) then begin
                               k := sref^.valu;        dlm := '<';
                               end;
                            sref := sref^.next;
                            end;
                         l := 0;
                         while (l<=pvariables^[j]^.maxvalue) and (k=maxint) do
                            if ((ds-[l])=p^.selectors[j]) then begin
                               k:=l;        dlm := '<';
                               end
                            else l:=l+1;
                         addchr(dlm);
                         if dlm='<' then addchr('>');
                         if k=maxint then addchr('?') else addval(k,j,0);
                         end
                      else begin
                         addchr(dlm);        addval(reflow(p^.selectors[j]),j,0);
                         end;
                      end;

                  end;

               addchr(']');
               end;

         vlidx := 1;        mapidx := 1;    (* write first line *)
         write(cv^.iteration:4,i:5,' ':2);
         j:=vlidx + vlcols-1;
         while (vlbuf[j]<>' ') and (vlbuf[j]<>']')
            and (vlbuf[j]<>',') and (j>vlidx) do j:=j-1;
```



```
         if j<=vlidx then j := vlidx + vlcols-1;
         for k:=vlidx to j do write(vlbuf[k]);
         k := vlidx+vlcols-1-j;
         if k>0 then write(' ':k);
         vlidx := j+1;

         write((maplow(cv^.pseed[i]^.map)+1):5,'   ');
         for j := 1 to cnum do begin
            printreal(p^.critvals[j],10);
            end;
         write(' ');

         dprtbufs;
         end;

(* check for exceptional events *)
      if mcard(cv^.exception)>0 then begin
         disppart(cv^.exception);
         write(cv^.iteration:4,' ':12,'exceptional events:',' ':(vlcols-12));
         for j:=1 to cnum do write(' ':10);
         mapidx := 1;        vlidx := 1;        vlbuf[1] := ' ';        dprtbufs;
         end;

(* write summary *)
      if prtflag>0 then begin
         write(cv^.iteration:4,'   totals',' ':(vlcols+6));
         for j := 1 to cnum do begin
            printreal(cv^.critvals[j],10);
            end;
         writeln;
         end;
      end;

procedure clearcv;                      (* clear cover storage area *)

var
      i, j            : integer;
      cv              : pcover;

begin
      with psstat^ do c[qcq] := c[qcq]+1;
      for i:=1 to numcv do begin
         cv := cvector[i];
         if cv<>nil then begin
            if qcq in dbug then writeln(' clearcv: k=',cv^.clstrs:1);
            for j:=1 to cv^.clstrs do p:=freecplx(cv^.pcluster[j]);
            dispose(cv);
            end;
         end;
      numcv := 0;
      for i:=1 to maxbase do cvector[i] := nil;
      end;

function savecv                         (* save a cover keeping 'base' best ones *)
      (cv             : pcover): pcover;

var
```

```
     ci            : integer;
     top           : integer;
     i, j, k, l    : integer;
     tol           : real;
     done          : boolean;
     p             : pcover;
     c1, c2        : pcomplex;
     base          : integer;
     unique        : boolean;

begin
     with pastat^ do c[qsv] := c[qsv]+1;

     (* savecv first checks if the cover is unique.  if not it is not
        saved.  this is indicated by the boolean 'unique' *)
     base := parameters^.base;      i:=1;      unique:=true;
     while (i<=numcv) and unique do begin
        p := cvector[i];      j:=1;      unique:=false;
        while (j<=clstrs) and not unique do begin
           k:=1;      c1:=cv^.pcluster[j];      unique:=true;
           while (k<=clstrs) and unique do begin
              l:=1;      c2:=p^.pcluster[k];      unique:=false;
              while (l<=nv) and not unique do begin
                 unique := c1^.selectors[l]<>c2^.selectors[l];      l:=l+1;
                 end;
              k:=k+1;
              end;
           j:=j+1;
           end;
        i:=i+1;
        end;

     if qsv in dbug then begin
        writeln(' savecv: in=',ord(cv):1,'  unique=',ord(unique):1);
        end;

     (* if a cover is unique it is saved if fewer than 'base' covers are
        currently saved.  else it is exchanged for a saved cover if it
        is better than that cover *)
     if unique then begin
        cvector[numcv+1] := cv;      ci := 1;      top := 1;
        repeat
           tol:=fcrit^.tlist[ci];
           for i := top to numcv+1 do
              for j := i+1 to numcv+1 do
                 if cvector[i]^.critvals[ci]>cvector[j]^.critvals[ci] then
                    begin
                    p:=cvector[j];      cvector[j]:=cvector[i];
                    cvector[i] := p;
                    end;
           if tol<0 then tol :=
              tol*(cvector[top]^.critvals[ci] -cvector[numcv+1]^.critvals[ci]);
           if numcv>=base then done :=
              cvector[base+1]^.critvals[ci]>cvector[base]^.critvals[ci]+tol
           else done := false;
           if not done then while (cvector[top]^.critvals[ci] < cvector[numcv+1
              ]^. critvals[ci]-tol) do top := top+1;
```



```
           ci := ci+1;
           until done or (top>numcv) or (ci>fcrit^.ccnt);
        p := cvector[base+1];
        if numcv<base then numcv:=numcv+1;
        end
     else p := cv;
     (* at this point, p contains:  cv if cv was not unique,
                                    cv if cv was not worth keeping,
                                    nil if fewer than 'base' covers saved,
                                 or pointer to complex replaced by cv. *)

     if qsv in dbug then writeln(' savecv: out=',ord(p):1);

     (* the complexes on the saved clustering are copied to the rejected
        clustering.  this is done because this info. is need for seed
        selection on the next iteration *)
     if p<>cv then begin
        if p=nil then begin
           new(p);
           for i:=1 to cv^.clstrs do p^.pcluster[i] := newcplx;
           end;
        for i:=1 to cv^.clstrs do p^.pcluster[i]^ := cv^.pcluster[i]^;
        end;

     (* a 'rejected' cover is always returned.  it is possibly the same one
        as input, or may have been built above *)
     savecv := p;
     end;

function clstring : pcover;    (* find best clustering over range of k values *

(* this function performs clustering for values of k from mink
   to maxk, evaluating the measure s=sparseness + k**beta as it goes.
   the clustering with the lowest s value is output, by returning
   a pointer to the "cover" record for it    *)

var
     i, j          : integer;
     bestofk       : pcover;
     bests         : real;
     tmpcv         : pcover;
     x1, x2        : real;
     pc, p         : pcomplex;
     hf            : integer;

begin
     with pastat^ do c[qcg] := c[qcg]+1;

     (* call wmcard to set weighted or unweighted cardinality calculation *)
     i := wmcard(events^.map,1);      (* assume unweighted (faster) method *)
     (* check if any weighted events, if so weighted method required *)
     pc := events;
     while (pc<>nil) do begin
        if pc^.cost>1 then begin
           i := wmcard(pc^.map,2); (* set weighted method *)
           pc := nil;
           end
```



```
        else pc := pc^.next;
        end;

     (* initialize variables used to determine best k value *)
     bestofk := nil;      bests := 0;
     (* start with mink number of clusters *)
     clstrs := parameters^.mink;

     if qcg in dbug then begin
        writeln(' clstring: ',numevents:1,' events');      p := events;
        for i := 1 to numevents do begin
           prt(p,1);      p := p^.next;
           end;
        end;

     (* build clusterings for a range of number of clusters *)
     repeat
        write('1');
        for i := 1 to (lnelength div 2) do write(' *');
        writeln;      writeln;

        (* prepare by selecting the first k events as seeds *)
        new(cv);      p := events;      i := 0;
        while (i<clstrs) and (p<>nil) do begin
           i := i+1;      cv^.pcluster[i] := p;      p := p^.next;
           end;
        if i<clstrs then begin        (* insufficient number of events *)
           clstrs := i;      parameters^.maxk := i;
           writeln(' maximum number of clusters set to the number of events (',
              clstrs:1,')');
           end;
        if not cendist(cv,central) then begin
           writeln;      writeln(' cant get initial seeds');      halt;
           end;

        (* the first clustering will use central seeds.  the base number
           and the probe number of clusterings are set to user-specified
           starting values *)
        rt := central;      citer := 0;      clearcv;
        btries := parameters^.base;      ptries := parameters^.probe;
        clk1:=clock; (* record start-of-clusterng time *)

        if q00 in dbug then hf:=2 else hf:=0;

        writeln(' experiment ',expnum:1,': k=',clstrs:1, ', criterion=',
           parameters^.critname);
        writeln;

        (* make successive clusterings until they fail to improve *)
        repeat
           (* make another clustering *)
           clk2:=clock;      cluster(cv);      citer := citer+1;
           cv^.iteration := citer;

           (* update 'bmsg' (cyber only) *)
           msgtext.la[29] := chr(ord('0')+(citer div 10));
           msgtext.la[30] := chr(ord('0')+(citer mod 10));
```



```
           msgtext.la[17] := chr(ord('0')+ clstrs);
           prtbmsg(msgtext.al,1,1);

           (* if trace is on, display each clustering *)
           if hf>1 then writeln(' intermediate results...');
           if q00 in dbug then begin
              disprule(cv,fcrit,hf);      hf:=1;      j:=clock;
              writeln('      (',(j-clk2):1,' ms)');   writeln;
              end;

           (* save base best covers *)
           cv2 := savecv(cv);
           (* if pointer returned different than input, cover was saved *)
           if (cv2<>cv) then begin
              (* set for central seeds; reset probe count *)
              rt:=central;      ptries := parameters^.probe;      cv := cv2;
              end
           (* if cover not better, try distant seeds *)
           else rt:=distant;

           (* choose next seeds.  if seeds cannot be found, stop by
              setting tries counts negative *)
           if not cendist(cv,rt) then begin
              btries := -1;      ptries := -1;
              end;

           (* free old complexes *)
           for j:=1 to clstrs do pc:=freecplx(cv^.pcluster[j]);

           (* count clusterings against base count and probe count *)
           btries := btries-1;
           if btries<1 then ptries := ptries-1;
           until (btries<0) and (ptries<0);

        (* display results *)
        writeln;      writeln;      j:=clock;
        writeln(' the ',numcv:1, ' best clusterings follow...   (', (j-clk1)
           :1,' ms)');
        hf:=2;
        for j := 1 to numcv do begin
           cv := cvector[j];      writeln;      disprule(cv,fcrit,hf);
           hf := 1;
           end;
        writeln;
        writeln('  (',numstar:1,' stars built)');

        (* calculate the 's' score for the best clustering *)
        tmpcv := cvector[1];
        x1 := wmcard(allevents,0) - ecard(allevents);
        x2 := (tmpcv^.critvals[0]+x1)*exp(ln(tmpcv^.clstrs)*parameters^.beta);
        writeln;      writeln(' for the best solution above, s=',x2:9);
        if x1>0.1 then writeln(
           ' in this calculation, sparseness was increased by ',x1:9,
           ' to account for weighted events');
        writeln;

        (* set 'bestofk' to clustering having best 's' score *)
```

```
        if (bests>x2) or (bestofk=nil) then begin
           cvector[1] := bestofk;       bestofk := tmpcv;       bests := x2;
           end;

        (* prepare for next higher value of k *)
        clearcv;        clstrs := clstrs + 1;
        until clstrs>parameters^.maxk;

      (* report the k-clustering that has the best 's' score over all
         values of k that were considered *)
      if parameters^.mink < parameters^.maxk then begin
         writeln;
         writeln(' with beta=',parameters^.beta:8:2,
            ' the best clustering at this level is for k=', bestofk^.clstrs:1);
         writeln;       disprule(bestofk,fcrit,2);
         end;
      clstring := bestofk;
      end;

procedure hlevel
     (hp           : pcover);(* pointer to parent clustering *)

(* this procedure creates subclusterings under a given parent.
   the procedure then recursively descends to the clusterings
   so created to created lower levels of the hierarchy, stopping
   when either the hierarchy height exceeds the limit, or the
   cluster size drops to low.

   a later addition would be to monitor s scores and stop when
   their improvement tapers off.              *)

var
     i            : integer;
     pl           : pcplxlist;
     ncv          : pcover;
     p,c          : pcomplex;
     h            : pcover;

begin
     for i:=1 to maxk do hp^.subcovers[i] := nil;

     (* check hierarchy height *)
     if hp^.rank < parameters^.maxht then begin

        for i:=1 to hp^.clstrs do begin
           c := hp^.pcluster[i];

           (* check size of cluster *)
           if mcard(c^.map) > parameters^.minsize then begin

              writeln;
              write(' beginning classification below hierarchy path ');
              h := hp;
              write(i:1);
              while (h<>nil) do begin
                 write('-',h^.pcnum:1);      h := h^.parent;
                 end;
```



```
              writeln;      writeln;

              (* build new event list *)
              pl := cpxlist;       events := nil;       numevents := 0;
              sop(allevents,clr,allevents,allevents);
              while (pl<>nil) do begin
                 p := pl^.cplx;
                 if scomp(p^.map,le,c^.map) then begin
                    p^.next := events;       events := p;
                    numevents := numevents + 1;
                    sop(allevents,union,allevents,p^.map);
                    end;
                 pl := pl^.next;
                 end;
              if events=nil then begin
                 writeln;       writeln('events inconsistency');       halt;
                 end;

              (* generate clustering *)
              ncv := clstring;

              (* attach clustering to hierarchy structure *)
              hp^.subcovers[i] := ncv;       ncv^.parent := hp;
              ncv^.rank := hp^.rank+1;       ncv^.pcnum := i;

              (* recursively generate lower clusters *)
              hlevel(ncv);
              end;
           end;
        end;
     end;

begin (* mane *)
     writeln('1  conjunctive conceptual clustering program cluster/2',
        '   last upgrade: ',verdate);

     (* create and set to zero the procedure frequency counters *)
     new(pastat);
     for si := q00 to qlast do pastat^.c[si] := 0;

     (* empty the set of alternative clusterings *)
     for i := 1 to maxbase do cvector[i] := nil;

     (* start with a 'bmsg' of 'setup' *)
     for i := 1 to lalfasize do msgtext.la[i] := ' ';
     strasgn(swork,'setup     ');
     for i := 1 to alfasize do msgtext.la[i] := swork[i];
     prtbmsg(msgtext.al,1,1);

     (* read input data as relational tables *)
     setup;

(* after completing input of all relational tables, the events
   table is condensed by removing all duplicate events and adjusting
   the weights (costs) to match.  the map of all events is also built *)
     sop(allevents,clr,allevents,allevents);       p := events;
```



```
     condensed := false;                              i := 0;
     while (p<>nil) do begin
        pp := p;       q := p^.next;
        while (q<>nil) do begin
           if intrsct(p,q) then begin
              condensed := true;       p^.cost := p^.cost + q^.cost;
              q := freecplx(q);        pp^.next := q;
              end
           else begin
              pp := q;       q := q^.next;
              end;
           end;
        sop(p^.map,clr,p^.map,p^.map);       masgn(p^.map,i);       i := i+1;
        sop(allevents,union,allevents,p^.map);
        p := p^.next;
        end;

     (* if some duplicate events were found, then the new event table
        is printed *)
     if condensed then begin
        i := wmcard(allevents,2);  (* tell system there are weighted events *)
        writeln;
        writeln(' duplicate events were combined');
        writeln(' the resulting event set follows');
        writeln;                          prtrlt(8,swork);
        end;

     (* scan all parameters lines for a hierarchical cover type *)
     pref := parameters;       cpxlist := nil;       hhead := nil;
     while (pref<>nil) and (cpxlist=nil) do begin
        if pref^.covertp=hierarchical then begin
           (* if a hierarchical cover is required, then save the
              original events on a master events list.  this frees the
              regular events list for holding events subsets *)
           p := events;
           while (p<>nil) do begin
              new(cpxl);       cpxl^.next := cpxlist;       cpxl^.cplx := p;
              cpxlist := cpxl;                              p := p^.next;
              end;
           end;
        pref := pref^.next;
        end;

     (* write title lines *)
     tptr := titles;
     while (tptr<>nil) do begin
        write(' ':(121-tptr^.length) div 2);
        for i:=1 to tptr^.length do write(tptr^.text[i]);
        writeln;       tptr := tptr^.next;
        end;

     (* process each parameters table line, each is an 'experiment' *)
     new(pmstat);       expnum := 0;
     while (parameters<>nil) do begin
        (* set experiment number and starting number of clusters *)
        expnum := expnum + 1;       clstrs := parameters^.mink;
```



```
        (* generate 'bmsg' text (for cyber only) *)
        strasgn(swork,'experiment');
        for i:=1 to alfasize do msgtext.la[i] := swork[i];
        msgtext.la[15] := 'k';       msgtext.la[16] := '=';
        strasgn(swork,'iteration ');
        for i:=19 to 28 do msgtext.la[i] := swork[i-18];
        msgtext.la[29] := '0';       msgtext.la[30] := '0';
        msgtext.la[12] := chr(ord('0')+(expnum div 10));
        msgtext.la[13] := chr(ord('0')+(expnum mod 10));
        msgtext.la[17] := chr(ord('0')+clstrs);
        prtbmsg(msgtext.al,1,1);

        (* set other parameters for this experiment *)
        dbug := parameters^.debug;       pmstat^.h1 := parameters^.h1;
        pmstat^.h2 := parameters^.h2;   cptr := criteria;

        (* get indicated lef for this experiment *)
        fcrit := nil;
        while (cptr<>nil) do begin
           if cptr^.name=parameters^.critname then fcrit:=cptr;
           cptr := cptr^.next;
           end;
        starcr(fcrit); (* tell star production about this lef *)

        (* produce hierarchy head clustering *)
        hhead := clstring;       hhead^.rank := 1;       hhead^.parent := nil;
        hhead^.pcnum := 0;
        (* if hierarchical cover, then recursively produce lower levels
           of the clustering hierarchy *)
        if parameters^.covertp=hierarchical then hlevel(hhead);

        parameters := parameters^.next;     (* do next unit of work *)
        end;

     (* print procedure frequency counts *)
     prtstats;
     end;

begin                              (* this is the main program *)
(* gblinit segment *)

(* initialization of global data *)

     linelimit(output,-1);
     postat := nil;       pestat := nil;
     pmstat := nil;       pastat := nil;
     titles := nil;       criteria := nil;     parameters := nil;
     events := nil;       numevents := 0;      domains := nil;
     dbug := [];          nv := 0;             clstrs := 1;
     pvariables := nil;   context := nil;      level := 0;
     prtflags := ['c','p'];
     sop(levelmap,clr,levelmap,levelmap);

     mane;  (* call 'main' procedure.  this was done to permit separate
               compilation in certain pascal environments *)
     end.
```

REFERENCES

[1] Michalski, R. S., Stepp, R. E., Diday, E., A RECENT ADVANCE IN DATA ANALYSIS: Clustering Objects into Classes Characterized by Conjunctive Concepts, Invited chapter in *Progress in Pattern Recognition,* Vol. 1, L. Kanal and A. Rosenfeld, Eds., pp. 33-55, 1981.

[2] Michalski, R. S., Stepp, R., Revealing conceptual structure in data by inductive inference, in Machine Intelligence 10, eds. J. E. Hayes, D. Michie, Y.-H. Pao, Ellis Horwood, Chichester, Halsted Press (John Wiley), New York, 1981.

[3] Michalski, R. S., Stepp, R., An application of AI techniques to structuring objects into an optimal conceptual hierarchy, Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, Canada, August 24-28, pp. 460-465, 1981.

[4] Michalski, R. S., Stepp, R. E., AUTOMATED CONSTRUCTION OF CLASSIFICATIONS: Conceptual Clustering versus Numerical Taxonomy, to appear in IEEE Pattern Analysis and Machine Intelligence, 1983.

[5] Michalski, R. S., KNOWLEDGE ACQUISITION THROUGH CONCEPTUAL CLUSTERING: A theoretical framework and an algorithm for partitioning data into conjunctive concepts, A Special Issue on Knowledge Acquisition And Induction, International Journal of Policy Analysis and Information Systems, Vol. 4, No. 3, pp. 219-244, 1980.

REFERENCES

[1] Michalski, R. S., Stepp, R. E., Diday, E., A RECENT ADVANCE IN DATA ANALYSIS: Clustering Objects into Classes Characterized by Conjunctive Concepts, Invited chapter in *Progress in Pattern Recognition,* Vol. 1, L. Kanal and A. Rosenfeld, Eds., pp. 33-55, 1981.

[2] Michalski, R. S., Stepp, R., Revealing conceptual structure in data by inductive inference, in Machine Intelligence 10, eds. J. E. Hayes, D. Michie, Y.-H. Pao, Ellis Horwood, Chichester, Halsted Press (John Wiley), New York, 1981.

[3] Michalski, R. S., Stepp, R., An application of AI techniques to structuring objects into an optimal conceptual hierarchy, Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, Canada, August 24-28, pp. 460-465, 1981.

[4] Michalski, R. S., Stepp, R. E., AUTOMATED CONSTRUCTION OF CLASSIFICATIONS: Conceptual Clustering versus Numerical Taxonomy, to appear in IEEE Pattern Analysis and Machine Intelligence, 1983.

[5] Michalski, R. S., KNOWLEDGE ACQUISITION THROUGH CONCEPTUAL CLUSTERING: A theoretical framework and an algorithm for partitioning data into conjunctive concepts, A Special Issue on Knowledge Acquisition And Induction, International Journal of Policy Analysis and Information Systems, Vol. 4, No. 3, pp. 219-244, 1980.