14. Basic Features of the TRANSFORM Step

14.1 Introduction

This introduction to the TRANSFORM step was primarily written in 1993. It is included here to provide partial guidance while the syntax for the step is being improved. The references in the text were originally to an earlier version of the documentation. There has been an attempt to revise these but some may not be correct.

Chapter 2 illustrated the TRANSFORM step through simple examples, using DIVIDE to produce a ratio and RPRINT or REPPRINT to print the values for the full sample and each replicate.

The TRANSFORM step begins with:

TRANSFORM [IN= *fname1*] OUT = *fname2*

If the input file is not specified, the previously created or referenced VPLX file will be used. As with other VPLX steps that employ input and output files simultaneously, these files must be different, that is, have different names and not be equated to each other.

Summary of this chapter:

- Section 14.2 presents an example, showing how ratio estimation may be implemented in the TRANSFORM step.
- Section 14.3 discusses CLASS and REMOVE BLOCK, which are auxiliary statements used during the TRANSFORM step.
- Section 14.4 describes how variables are passed to subroutines, which are the primary work unit of the TRANSFORM step.
- Section 14.5 describes ADD, SUBTRACT, MULTIPLY, and DIVIDE.
- Section 14.6 and 14.7 provide a brief outline of these incomplete sections, listing other subroutines.

8.2 An Example of the TRANSFORM Step for Ratio Estimation

14.2

This section will show the computation of variance for a post-stratified estimator. For example, suppose that there are independent estimates, perhaps based on the previous census, for the numbers of owners and renters in an area. If the sampling frame provided information on tenure for all units in the population, then this information might be incorporated into the stratification of the initial sample. If the information is not or cannot be used for stratification, however, it is still possible to use the estimated totals by forming poststrata. The next example illustrates ratio estimation applied to produce a poststratified estimate based on tenure.

```
comment  EXAM18

comment  This example begins from EXAM11 but employs an extended
         second TRANSFORM step to poststratify the data by renters and
         owner/others, as if there were separately available control
         totals, such as from a recent census.

create  in = exampl11.dat  out = exampl11.vpl

input    rooms persons cluster tenure

      4 variables are specified

format   (4f2.0)

comment  The input data set contains
 5 7 1 2
 6 8 2 2
 5 2 3 1
 4 1 4 2
 8 4 5 1
 8 2 6 1

class  tenure (2/1,3) 'Renter' 'Owner/other'

labels  rooms 'Number of rooms' persons 'Persons'
        tenure 'Tenure'

cat  rooms into rooms_c (1-5/6-high) '5 or fewer' '6 or more'

cat  persons into persons_c (1-2/3-4/5-high) '1-2' '3-4' '5 or more'

cross rooms_c by persons_c into rooms_cross

labels  rooms_c 'Rooms' persons_c 'Persons'
        rooms_cross  'Rooms by size of household'

    (Simple) jackknife replication assumed

    Size of block   1  =            28

    Total size of tally matrix =    28

    Unnamed scratch file opened on unit 13

    Unnamed scratch file opened on unit 14
```

```
    End of primary input file after obs #       6

transform  in = exampl11.vpl out=exam11a.vpl

divide

old  rooms persons / class tenure (0-2)                                       #1

derived   proom  / class tenure (0-2)

    (assigned to block   2)

labels  proom  'Rooms per person'


display

option   ndecimal=2

list     n(1) rooms  persons total (rooms persons ) proom
          rooms_c persons_c rooms_cross
          / class total /
         n(1) rooms persons  total (rooms persons ) proom / class tenure

cov      n (1)    / class tenure

cov      total (rooms ) / class tenure


                                                  Estimate      Standard error

Sample N (wtd) for block   1                          6.00                 .00         #2

Number of rooms         :  MEAN                       6.00                 .68

Persons                 :  MEAN                       4.00                1.18

Number of rooms         :  TOTAL                     36.00                4.10

Persons                 :  TOTAL                     24.00                7.10

Rooms per person        :  VALUE                      1.50                 .52

Rooms                   :  PERCENTS
   5 or fewer                                        50.00               22.36
   6 or more                                         50.00               22.36

Persons                 :  PERCENTS
   1-2                                               50.00               22.36
   3-4                                               16.67               16.67
   5 or more                                         33.33               21.08

Crossed values of Rooms                  : PERCENTS
   Persons                 :  1-2
      5 or fewer                                     66.67               37.27
      6 or more                                      33.33               37.27
   Persons                 :  3-4
      5 or fewer                                       .00                 .00*
      6 or more                                     100.00                 .00*
```

Persons	: 5 or more			
5 or fewer		50.00	64.55	
6 or more		50.00	64.55	

Tenure : Renter

		Estimate	Standard error	
Sample N (wtd) for block	1	3.00	1.34	#3
Number of rooms	: MEAN	5.00	.65	
Persons	: MEAN	5.33	2.44	
Number of rooms	: TOTAL	15.00	6.88	
Persons	: TOTAL	16.00	9.25	
Rooms per person	: VALUE	.94	.30	

Tenure : Owner/other

		Estimate	Standard error	
Sample N (wtd) for block	1	3.00	1.34	#4
Number of rooms	: MEAN	7.00	1.12	
Persons	: MEAN	2.67	.75	
Number of rooms	: TOTAL	21.00	9.77	
Persons	: TOTAL	8.00	4.00	
Rooms per person	: VALUE	2.63	.71	

Covariances of the Sample Estimates

	Estimate	1	2	
Tenure : Renter				
1: Sample N for block 1	3.0000	.18000000D+01		
Tenure : Owner/other				
2: Sample N for block 1	3.0000	-.18000000D+01	.18000000D+01	#5

```
                    Covariances of the Sample Estimates

                                Estimate              1               2

    Tenure                  :   Renter


    Number of rooms         :  TOTAL
   1                                 15.0000   .47400000D+02


    Tenure                  :   Owner/other


    Number of rooms         :  TOTAL
   2                                 21.0000  -.63000000D+02   .95400000D+02         #6

transform    out = exam11b.vpl

comment   first use the reciprocal subroutine, with constants, as
          if the universe count for renters and owners/others were 3
          each

reciprocal                                                                          #7

old       n(1)  / class tenure

constants  2 * 3

derived   ratio_factor / class tenure
```

(assigned to block   3)

```
comment  now use multiply to do a mass change on the variables, except
          for proom

multiply                                                                            #8

modify     n(1) rooms persons rooms_c persons_c rooms_cross
                  / class tenure

old       ratio_factor / class tenure

comment   recompute  proom, writing over the previous values

divide

old  rooms persons / class tenure (0-2)

modify     proom  / class tenure (0-2)

display

option   ndecimal=2

list     n(1) rooms  persons total (rooms persons ) proom
          rooms_c persons_c rooms_cross
          / class total /
         n(1) rooms persons  total (rooms persons ) proom / class tenure
```

14.6

```
cov      n (1)    / class tenure

cov      total (rooms ) / class tenure

                                                   Estimate      Standard error

Sample N (wtd) for block   1                           6.00                 .00

Number of rooms         :  MEAN                        6.00                 .65

Persons                 :  MEAN                        4.00                1.28

Number of rooms         :  TOTAL                      36.00                3.87

Persons                 :  TOTAL                      24.00                7.66

Rooms per person        :  VALUE                       1.50                 .38

Rooms                   :  PERCENTS
   5 or fewer                                         50.00               26.35
   6 or more                                          50.00               26.35

Persons                 :  PERCENTS
   1-2                                                50.00               26.35
   3-4                                                16.67               18.63
   5 or more                                          33.33               18.63

Crossed values of Rooms                 : PERCENTS
   Persons                 :  1-2
      5 or fewer                                      66.67               42.45
      6 or more                                       33.33               42.45
   Persons                 :  3-4
      5 or fewer                                        .00                 .00*
      6 or more                                      100.00                 .00*
   Persons                 :  5 or more
      5 or fewer                                      50.00               64.55
      6 or more                                       50.00               64.55

    Tenure                  :   Renter
                                                   Estimate      Standard error

Sample N (wtd) for block   1                           3.00                 .00         #9

Number of rooms         :  MEAN                        5.00                 .65

Persons                 :  MEAN                        5.33                2.44

Number of rooms         :  TOTAL                      15.00                1.94

Persons                 :  TOTAL                      16.00                7.33

Rooms per person        :  VALUE                        .94                 .30

    Tenure                  :   Owner/other
                                                   Estimate      Standard error

Sample N (wtd) for block   1                           3.00                 .00        #10

Number of rooms         :  MEAN                        7.00                1.12
```

```
Persons                 :  MEAN                    2.67                 .75

Number of rooms         :  TOTAL                  21.00                3.35

Persons                 :  TOTAL                   8.00                2.24

Rooms per person        :  VALUE                   2.63                 .71

                      Covariances of the Sample Estimates

                                      Estimate              1               2


    Tenure                  :   Renter


   1: Sample N for block   1            3.0000   .00000000D+00


    Tenure                  :   Owner/other


   2: Sample N for block   1            3.0000   .00000000D+00   .00000000D+00

                      Covariances of the Sample Estimates

                                      Estimate              1               2


    Tenure                  :   Renter


    Number of rooms         :  TOTAL
   1                                   15.0000   .37500000D+01


    Tenure                  :   Owner/other


    Number of rooms         :  TOTAL
   2                                   21.0000   .48317730D-29   .11250000D+02        #11
```

Exhibit 14.1 Ratio estimation to fixed totals by tenure

The first transform step computes `proom` as before. Similar to its function in the DISPLAY step, the /CLASS specification at #1 requests that the values of `rooms` and `persons` for the total and then for each level of `tenure` are to be operated on by DIVIDE. The next statement specifies that 3 separate results be stored in a new variable, `proom`. Furthermore, the statement declares this variable to be of the type derived, a type that the CREATE step does not produce.

Because each of the 6 replicates for the simple jackknife omits a single observation, the estimated variance of N(1) for the total number of cases is zero, at #2. (Two of the standard error estimates following #2 are marked by asterisks; this indicates that the value was undefined for one or more of the replicate samples.) The jackknife replicate samples vary the observed number of cases for the 2 poststrata, renters and owners, however, so that both have the same estimated (non-zero) standard error at #3 and #4. (The fact that the 2 estimates are equal is not coincidental, since this will be true whenever the sum of 2 random variables is a constant.) Note #5 calls attention to the covariance matrix for N(1) classified by tenure, especially to the negative covariance between the 2 estimates, equal to -1 times the variance of each. A second covariance matrix, for estimated number of rooms by tenure, follows, with again a substantial negative covariance at #6 between the 2 estimates. Intuitively, this negative covariance occurs because each sampled renter, which adds to the total rooms for renters, does not contribute to the total for owners, and similarly each sampled owner does not contribute to the total for renters, so that, in effect, the 2 estimated totals are in competition.

The second TRANSFORM step implements the ratio estimation. A new subroutine, RECIPROCAL, beginning at #7, computes the ratio factor to be used and illustrates several points about the TRANSFORM step. A CONSTANTS statement provides constants, in this case the fixed totals for the 2 poststrata, to the subroutine. In general, several, but not all, of the subroutines of the TRANSFORM step include the provision for constants as arguments. (The COPY subroutine in the TRANSFORM step also allows constants to be copied into a VPLX variable, similar to the function of the CONSTANTS statement in the CREATE step, described in Section 3.4.) A second feature of this subroutine is that N(1) appears here as an argument in the TRANSFORM step, using the same syntax as the DISPLAY step. In other words, the weighted N is available for calculation and even modification in the TRANSFORM step. In addition, unlike the previous TRANSFORM step, in which a marginal total for `proom` was desired, only 2 values of the ratio factor, for each of the poststrata, are computed here, without a marginal total.

The MULTIPLY subroutine at #8 applies the ratio factor to several variables at once. This subroutine also serves the purpose of illustrating a number of additional features of the TRANSFORM step. This is the first illustration that several of the available subroutines in the TRANSFORM step, including MULTIPLY, can operate on an arbitrary number of variables. Secondly, MODIFY appears instead of OLD, indicating that the series of existing variables may be modified by the subroutine. Furthermore, `ratio_factor` is now classified as an OLD variable, even though it was just created by the previous subroutine. In other words, OLD may be used to reference any existing variable, including those just created by previous subroutines. The list of MODIFY variables includes both 1) real variables that of the same size as `ratio_factor,` as well as 2) categorical and crossed categorical variables that each represent matrices of a larger size, suggesting that MULTIPLY has rules for operating on matrices of different sizes in an orderly manner.

A final step computes `proom` as before. Because this variable is itself a ratio instead of an estimated total, it would have been inappropriate to attempt to adjust it through the previous MULTIPLY subroutine, since the effect of ratio estimation on both the numerator and denominator must be reflected. Instead, a new value is computed from the adjusted values of the numerator and denominator, and the new value replaces the old one. This subroutine illustrates another general principle about TRANSFORM - that subroutines in some circumstances may simply replace the contents of an existing MODIFY variable rather than use the current information in some way, as was done to the MODIFY variables in the preceding MULTIPLY statement.

The DISPLAY provides the same statistics as before, but now after the effect of ratio estimation. Since the control totals by tenure agreed exactly with the estimated totals before ratio estimation, none of the estimates change; in practice, such agreement would rarely occur and most estimates would be different. Most standard errors have changed, however, reflecting the effect of the ratio estimation on the replicate totals. To begin, the estimated standard errors at #9 and #10 for the estimated population totals by tenure are now 0, showing that each replicate sample has been weighted to the control totals. A comparison of results before ratio estimation shows that, within renters or owners, the estimated standard errors for totals have been greatly reduced, whereas the estimated standard errors for the means and for the ratio, `proom`, have remained the same. The latter agreement reflects that the effect of the ratio factor cancels in the divisions involved in each case (since the means involve division by the estimated sample n). Ratio estimation generally changes the estimated standard errors of estimated characteristics of the overall population, however, including the means and `proom`.

The covariance matrices that follow again show the effects of ratio estimation. All elements of the covariance matrix for the estimated n's are now 0, and the covariance between the estimated totals for rooms by tenure is shown to be a number that is effectively 0. (The specific value here is a result of roundoff error on the PC on which it was run. The exact value could vary from one type of computer to another but in each case should be a similarly small number that is effectively 0.)

The remaining parts of this chapter explain virtually each aspect of the preceding example. For simplicity, however, the scope of the chapter is limited to the rules for the introduction of new DERIVED and REAL variables, and the modification of DERIVED, REAL and estimated N's for blocks during the TRANSFORM step. Aside from the effects of class variables, each of these variables occupies a single cell, and the rules for their treatment in the TRANSFORM step are consequently simpler. Rules for the remaining variable types are deferred to later chapters, since these rules become more complex as they consider both the effect of class variables and the number of cells of the variable. In practice, however, this chapter provides an adequate accounting of all features of the TRANSFORM step presumed by the next chapter.

14.3 Auxiliary Statements

14.3.1 CLASS in the TRANSFORM Step

All class variables included on the incoming VPLX file may be referenced during the TRANSFORM step. In addition, however, new class variables may be introduced. New class variables can only apply to other newly created variables, since there is no provision in the TRANSFORM step to alter the manner in which existing variables are cross-classified without creating new variables.

Chapter 5 described the CLASS statement for the CREATE step. The syntax during the TRANSFORM step is:

```
CLASS  vlist (nlevel) ['label1' ['label2'[ ...]]]
```

The variable or variables in *vlist*, which must not appear on the incoming VPLX file or have been referenced previously in the TRANSFORM step, become newly defined class variables, each with *nlevel* levels. Labels are recommended, since the labels will be blank otherwise.

As an example, suppose `income` is a real variable that has been cross-classified by sex and age in six age groups, 18-24, 25-34, 35-44, 45-54, 55-64, and 65+, then

```
class  broad_age (3) '18-34' '35-54' '55+'
copy
old    income / class  sex * age (1+2, 3+4, 5+6)
real   income_br / class sex * broad_age
```

would establish a three-level age grouping and then create a new variable cross-classified by sex and the broad age groups.

14.3.2 REMOVE BLOCK

The function of the REMOVE BLOCK statement is to control the contents of the output VPLX file. It has no effect on the availability of variables for calculation during the TRANSFORM step; in general, all variables from the input file may be used for calculation, regardless of whether they will be written to the output file. Specifically, the statement is used to identify a block on the input file to be omitted from the output file, including the estimated N for the block, if present. The form of the statement is:

```
REMOVE BLOCK   nblock
```

where `nblock` is a number of a block on the input file. The statement identifies a single block, but more than one such statement may be included in the TRANSFORM step. It is possible, for example, to use this statement to write only newly created variables to the output file, if desired.

The primary purpose of this statement is to save computer storage. It also may be useful if an application reaches the limit on the number of allowed blocks or some other VPLX resource.

14.3.3 Auxiliary Statements Similar to Their Use in the CREATE Step

A single KEEP or DROP statement (Section 3.14) may be included in the TRANSFORM step to define which variables to include on the output file. Unlike the CREATE step (Chapter 3), KEEP in the TRANSFORM step is not used to achieve assignment of variables to blocks. The estimated N of each block is automatically retained on the outgoing file, unless dropped by a REMOVE BLOCK command; consequently, it is neither necessary nor allowed to reference the N of a block on a KEEP list. Similarly, N of a block may not be included on a DROP list; the only way to drop the N of a block is to remove the entire block through REMOVE BLOCK.

RENAME follows the same rules as in the CREATE step, that is, it takes effect immediately, and subsequent statements must reference the new name. LABEL (Section 3.15) and LEVEL as in the CREATE step, may be applied both to new and incoming variables. For example, one may use the TRANSFORM step to correct labels from the CREATE step without rerunning the CREATE step.

14.4 Specifying Values to Subroutines

14.4.1 Introduction

The following types of information may be passed to subroutines in the TRANSFORM step:

1) OLD variables, which are existing variables. Their values are not changed by the subroutine.

2) MODIFY variables, also existing variables, which may be modified or overwritten by the subroutine, updating the information.

3) New variables, which are declared by type. This chapter discusses the declarations REAL and DERIVED.

4) CONSTANTS, which are constant values. Some subroutines, including ADD, SUBTRACT, MULTIPLY, DIVIDE, and RECIPROCAL, do not alter the values of the constants, while others do.

5) INTEGER CONSTANTS, which are used by some subroutines not discussed in this chapter.

6) STRING and LONGSTRING, which may provide character strings to subroutines. These declarations are again outside the scope of this chapter.

7) OPTIONS for some built-in subroutines, beginning with Version 93.03. Options will not be available to user-supplied routines.

14.4.1.1 Order of Specification A separate list is made for each of type of argument and each list is passed, in the order of reference, to the subroutine. Consequently, the following 2 statements usually have different effects, unless the specific operation is cummutative in these two variables:

```
old   a b

old   b a
```

On the other hand, VPLX does not maintain the order of arguments across lists. For example, the 2 pairs of statements:

```
old   a  b
real  c  d
```

and

```
real  c  d
old   a  b
```

have identical consequences for all subroutines, since VPLX does not retain the information about whether the old or new variables were listed first. Consequently, either order is acceptable and in general VPLX imposes no restriction on which type of list precedes another. Furthermore, statements building up different lists may be interspersed:

```
old   a
real  c
old   b
```

```
real  d
```

again with identical consequences as the preceding two examples.

Here and elsewhere, however, some users may prefer a programming style that imposes restrictions not exacted by VPLX. For example, it may be helpful for clarity to place all OLD statements first, followed by MODIFY, and then, finally, the declarations for new variables.

14.4.2 Introducing New Variables: REAL and DERIVED

14.4.2.1 REAL. One or more new real variables may be created by a subroutine through the declaration:

REAL   *vlist*

or

REAL   *vlist* / CLASS   *clist*

where *vlist* contains variable names not appearing either on the input file or in previous specifications in the step. In the first case, each real variable occupies a single cell in a block without class variables. In the second case, *clist* should either contain a single class variable or a series of class variables shown as a product separated by "*", as in the DISPLAY step. In the second case, each new real variable will be placed in a block with the specified class variables, creating a new block if necessary.

14.4.2.2 DERIVED One or more new derived variables may be created through the declaration:

DERIVED   *vlist*

or

DERIVED   *vlist* / CLASS   *clist*

where *vlist* contains variable names not appearing either on the input file or in previous specifications in the step. In the first case, the derived variable occupies a single cell in a block without class variables. In the second case, *clist* should either contain a single class variable or a series of class variables shown as a product separated by "*", as in the DISPLAY step. In the second case, each new derived variable will be placed in a block with the specified class variables, creating a new block if necessary. In addition, in each class variable may be stated in a form that implies a margin, e.g., (0-n), (0,1,2,3) (where 3 is the upper limit of the class), etc.

Placement into blocks will be according to an appropriate match on whether each class variable has a specified margin.

Examples of the use of the DERIVED specification appear in Exhibit 14.1 at #1 and at #7.

14.4.2.3 Summable vs. Non-Summable Variables VPLX variables may be distinguished according to their treatment for margins of class variables. Summable variables, including REAL, REAL with missing, CAT, CROSSED REAL, CROSSED CAT, and N of a block, are stored only as inner cells of the cross-classification of their corresponding class variables. On the basis of the inner cells of the matrix, VPLX constructs requested margins for summable variables in DISPLAY and for summable variables declared as OLD in the TRANSFORM step (Section 14.4.3).

DERIVED is an example of a non-summable type. The TRANSFORM step can create this variable, although the CREATE step does not. Margins are optional, but, if they are desired, they must be constructed at the same time as the variable is created. Margins of non-summable variables occupy separate cells on a VPLX file.

The example in Exhibit 14.1 illustrates both aspects. At #1, the summable variables `rooms` and `persons` are matched with a class statement for `tenure (0-2),` which requests the construction of a marginal total. The resulting non-summable DERIVED variable, `proom,` is stored with a marginal total occupying a separate cell. At #7, however, no separate margin is created for `ratio_factor,` which becomes defined only for the 2 separate levels of `tenure.`

14.4.3 Using Values of Existing Variables: OLD

The purpose of the OLD declaration is to identify one or more existing variables whose values are to be used in the calculation. When declared as OLD, the current contents of the variables will not be permanently modified by the subroutine, in contrast to MODIFY. This rule applies not only to any built-in subroutines in VPLX, but also to any user-supplied subroutines, such as USER1, USER2, etc.

For example, OLD is the required declaration for the subroutine REPPRINT, since the subroutine merely prints the values without altering them.

The statements take the form:

```
OLD   vlist
```

or

```
OLD  vlist / CLASS  clist
```

In two respects, the syntax of the OLD specification is similar to specifications for LIST or other parts of the DISPLAY step:

1) Generally, the rules governing *clist* are as described in Section 7.3 in the DISPLAY step, that is, one may take advantage of combining classes or other forms of collapsing, features not allowed for new or MODIFY variables. Only one rectangular matrix may be specified for each variable, however, in contrast to the DISPLAY step, where an arbitrary list of classes may be specified.

2) Functions such as MEAN, PERCENT1, etc., may be used, as described in Section 4.4.

There are some important differences between OLD and LIST as well:

1) At most one *clist* may appear in each statement, unlike the DISPLAY step which permitted alternation of variable lists and CLASS statements. If more that one list is wanted, then multiple OLD statements are required.

2) If no function is specified for a variable, then the default is to pass the specific matrix to the subroutine without altering it. In particular, if means are desired, then MEAN( ) must be included in the specification.

In Exhibit 14.1, OLD appeared at #1 for two variables used to compute `proom`, since these variables were not themselves altered by the calculation. In the second TRANSFORM step, it again appeared at #7, used with N(1), and at #8, where it was used with `ratio_factor`.

14.4.3.1 Examples of OLD If `occupation` is a categorical variable that has been cross-classified by sex and age in six age groups, then,

```
old  occupation
old  occupation / class sex
old  occupation / class sex (0-2)
old  occupation / class age (3-6) * sex
old  occupation / class age (4) * sex (1)
```

specify, respectively,

the marginal distribution of occupation
the marginal distribution of occupation by sex, presenting the subroutine with the totals for

occupation for sex(1), followed by the data for sex(2)
the marginal distribution of occupation for the whole table followed by the cross-classification by sex
a matrix extracted from the full cross-classification, giving occupation for age(3)*sex(1), age(4)*sex(1), age(5)*sex(1), age(6)*sex(1), age(3)*sex(2),..., age(6)*sex(2)
occupation for age(4)*sex(1) only.

Note that the order of specification of the class variables is important. For example, age(3-6)*sex varies age first, while sex*age(3-6) would produce the order sex(1)*age(3), sex(2)*age(3), etc. In general, matrices will be arrayed by varying the first class variable first, the second class variable second, etc., in "FORTRAN" order.

Typically, specifications that select a single level of a class can appear anywhere:

```
old occupation  / class age(3-6) * sex (1)
old occupation  / class sex(1) * age(3-6)
```

produce the same matrix. Generally, the two specifications would yield the same result, although some subroutines may accept only one of these. Most of the built-in subroutines, such as MULTIPLY, ADD, REPPRINT, etc., accept either. In a few instances, however, the order of dimensions of the matrix may be important. For example, MMULTIPLY would view the two specifications differently - the first specifies a 4 by 1 matrix whereas the second specifies 1 by 4. User-supplied routines may also impose restrictions on the statement of dimensions. For example, a specific routine may, because of /class specifications for other variables, check that the final dimension is 4, making only the second version acceptable.

If `occupation` is a summable variable, then several forms of collapsing are available,

```
old  occupation
old  occupation / class sex (1, 2, 0) * age (1+2)
old  occupation / class age (1, 2, 1+2, 3, 4, 3+4, 5, 6, 5+6)
old  occupation / class age (1, 2, 1+2, 3, 4, 3+4, 5, 6, 5+6)
                                                * sex (1)
```

are all valid expressions, giving `occupation` crossed by, respectively,

the total
sex at levels 1 and 2 followed by total, for ages 1 and 2, summed
age 1, age 2, ages 1 and 2 summed, age 3, age 4, etc.
age 1, age 2, ages 1 and 2 summed, age 3, etc., for sex 1 only.

If `earnings` were cross-classified by `race`, `sex`, and `age`, and the current or previous TRANSFORM step created `earn_ratio` ,

```
divide
old   mean(earnings) / class  age (0-6) * sex * race (1)
old   mean(earnings) / class  age (0-6) * sex * race (2)
derived  earn_ratio  / class  age (0-6) * sex
```

then any of the following may be used subsequently,

```
old   earn_ratio    / class age (0-6) * sex
old   earn_ratio    /  class sex
old   earn_ratio    / class  age (1,2) * sex (1)
old   earn_ratio    / class  age ( 1, 1, 2) * sex
```

but these may not

```
old   earn_ratio
old   earn_ratio   / class age * sex (1,2,0)
old   earn_ratio   / class age(1,2)
```

since each of the latter examples requires a margin summed across sex. If, instead, `earn_ratio` , was created through

```
divide
old   mean(earnings) / class  age (0-6) * sex (0-2) * race (1)
old   mean(earnings) / class  age (0-6) * sex (0-2) * race (2)
derived  earn_ratio  / class  age (0-6) * sex (0-2)
```

then the margin summed across sex would be available, and any of the previous references to `earn_ratio` would be correct.

Note also that the + operator in the class specification is not allowed for derived variables.

14.4.4 Modifying Values of Existing Variables: MODIFY

Unlike the treatment of OLD variables, VPLX allows subroutines to alter the values of MODIFY variables. As noted in Section 14.2, the previous values may be used in the calculation or they may be ignored and simply overwritten, depending upon the subroutine.

There are more restrictions on CLASS specifications for MODIFY variables than OLD, essentially because VPLX has to be able to store the results in an unambiguous manner. No marginal totals of summable variables may be specified; instead, the specification must refer to an individual cell, a rectangular submatrix of the inner cells, or the entire inner matrix.

Margins may be referenced for non-summable variables, but only if the margins had been previously created.

Exhibit 14.1 includes two instances of MODIFY, following #8. In the first, MULTIPLY is used to apply the ratio adjustment to a series of variables, modifying their contents. As noted in the comment, the next subroutine, DIVIDE, simply writes over the values of `proom`. Rules concerning these two uses of MODIFY follow in the discussion of individual subroutines, beginning in Section 14.5.

14.4.4.1 Examples of MODIFY Using the variables previously defined in Section 14.4.3.1, the following are acceptable forms for a summable variable:

```
modify  occupation / class sex * age
modify  occupation / class age * sex
modify  occupation / class sex (2, 1) * age (1)
modify  occupation / class age (1, 2, 3) * sex
modify  occupation / class age (1, 2,  5, 6) * sex (1)
```

The following are unacceptable, since each involves summation over classes

```
modify  occupation
modify  occupation / class age
modify  occupation / class sex (1, 2, 0) * age (1+2)
modify  occupation / class age (1, 2, 1+2, 3, 4, 3+4, 5, 6,
          5+6)
```

Note that each of these would be acceptable as OLD specifications.

For non-summable variables, margins may be modified only if they were previously created. To continue the example for non-summable variables from Section 14.4.3.1, the following are acceptable:

```
modify   earn_ratio    / class age (0-6) * sex
modify   earn_ratio    /  class sex
modify   earn_ratio    / class  age (1,2) * sex (1)
```

If `earn_ratio` , was created in the first of the two ways, which did not establish margins added across sex, the following are not acceptable uses of MODIFY:

```
modify   earn_ratio
modify   earn_ratio   / class age * sex (1,2,0)
modify   earn_ratio   / class age(1,2)
modify   earn_ratio    / class  age ( 1, 1, 2) * sex
```

since each of the first 3 requires a marginal total across sex that was not initially created, and the last redundantly specifies cells, which is allowed with OLD but not with MODIFY. If `earn_ratio` is instead created in the second of the two ways, then each of the first 3 would be acceptable, but not the last.

14.4.5 Specifying Constants to Subroutines: CONSTANTS

Exhibit 14.1 included a use of a CONSTANTS statement, similar in syntax to the version in the CREATE step (section 3.4), but without an associated assignment to one or more variables. In its simplest application, the function of this statement is to provide a series of constants to a subroutine. The constants occupy separate storage, reserved for the specific subroutine.

Some subroutines, including ADD, MULTIPLY, SUBTRACT, and DIVIDE, preserve the values of the constants as initially stated. Some subroutines take advantage of the fact that the storage for the constants is reserved for the given subroutine, and modify the initial values. For example, the subroutine SAVEFULL uses the space reserved for the constants to store the values of a group of variables for the full sample.

Only one CONSTANTS statement is allowed for each subroutine, although the number of specified constants is arbitrary, subject only to the availability of computer storage. As with the comments on order in Section 14.4.1.1, the placement of the CONSTANTS statement before or after other OLD, MODIFY, or new variable declarations has no effect on the outcome.

8.5 ADD, MULTIPLY, SUBTRACT, DIVIDE

The subroutines that perform basic arithmetic operations are generally called on the most in VPLX applications. Several forms are available, accommodating the optional use of constants and choices between modifying variables or creating new ones.

The section is divided into 2 primary subsections. Section 8.5.1 describes the operation of the subroutines when one or more constants are specified. Each application with constants has an exact counterpart without constants, which will be covered by Section 8.5.2.

8.5.1 ADD, MULTIPLY, SUBTRACT, DIVIDE with CONSTANTS

8.5.1.1 Basic Operations with Single Values ADD operates on a single real variable *a* with a constant *b* producing a new variable *c*:

```
ADD
OLD         a
CONSTANT    b
REAL        c
```

by adding *a* to *b* and placing the result in *c*. The other subroutines are similar:

```
MULTIPLY
OLD         a
CONSTANT    b
REAL        c
```

multiplies *a* and *b* and places the result in *c*;

```
SUBTRACT
OLD         a
CONSTANT    b
REAL        c
```

subtracts *b* from *a*, (i.e. *a* - *b*), and places the result in *c*;

```
DIVIDE
OLD         a
CONSTANT    b
REAL        c
```

divides *a* by *b*, (i.e. *a* / *b*), and places the result in *c*.

As noted in Section 8.4.5, the placement of the CONSTANT statement relative to either the OLD or REAL statements has no effect on the outcome. For example,

```
DIVIDE
CONSTANT    b
OLD         a
REAL        c
```

also divides *a* by *b*, (i.e. *a* / *b*), and places the result in *c*.

8.5.1.2 Rules for Matrices, with a List of Constants of Matching Length ADD operates on a real variable *a* cross-classified by one or more classes in *clist1* (which uses the * notation for product in the case of more than one class) with the same number of constants *blist* producing a real variable *c* cross-classified by *clist2*, which must have the same number of implied elements:

```
ADD
OLD          a / CLASS  clist1
CONSTANTS    blist
REAL         c / CLASS  clist2
```

by adding *a* to the corresponding element in *blist*, element by element, and placing the result in *c*. The order of elements in *a* is determined by applying FORTRAN order to the specification in *clist1*, i.e., varying the first class the most quickly, the next class the next most quickly, etc. Similarly, the elements of *c* are also varied in FORTRAN order.

As an example, suppose num_jobs is a real variable cross-classified by the class variables sex and age in three broad age groups. If the totals in num_jobs are 1, 2, 3, 4, 5, and 6 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells, respectively, then,

```
add
old         num_jobs / class sex * broad_age
constants   0,1,2, 3*0
real        num_jobs_mod / class  sex * broad_age
```

creates a new variable num_jobs_mod with totals 1, 3, 5, 4, 5, and 6 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells. If, instead,

```
add
old         num_jobs / class broad_age * sex
constants   0,1,2, 3*0
real        num_jobs_mod / class  broad_age * sex
```

then, num_jobs_mod will contain totals 1, 2, 4, 4, 7, and 6 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells.

As in Section 8.5.1.1, the constants are always the second operator. For example,

```
DIVIDE
OLD          a / CLASS  clist1
CONSTANTS    blist
REAL         c / CLASS  clist2
```

divides each of the elements of *a* by the corresponding element in *blist*. Again, changing the placement of the CONSTANT statement

```
DIVIDE
CONSTANT   blist
OLD        a / CLASS  clist1
REAL       c / CLASS  clist2
```

does not change the outcome.

8.5.1.3 Rules for Matrices and a Single Constant If a single constant is given, then that constant is used repeatedly as the second operator with each cell of the matrix, in turn. For example

```
DIVIDE
OLD        a / CLASS  clist1
CONSTANT   b
REAL       c / CLASS  clist2
```

divides each cell of *a* by *b* and places the result in the corresponding cell of *c*. The number of elements of *a* and *c* must agree.

8.5.1.4 Rules for Matrices with a List of Constants of Different Length The length of *blist* may be less than the number of elements of *a* if it is an integer multiple, *m*. In that case, the first *m* elements of *a* are operated on by the first element of *blist*, etc. The sizes for *a* and *c* must still agree.

As an example, consider `num_jobs` from Section 8.5.1.2. Then,

```
subtract
old        num_jobs / class sex * broad_age
constants  0,1,2
real       num_jobs_mod / class  sex * broad_age
```

creates a new variable `num_jobs_mod` with totals 1, 2, 2, 3, 3, and 4 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells. If, instead,

```
subtract
old        num_jobs / class broad_age * sex
constants  0,1,2
real       num_jobs_mod / class  broad_age * sex
```

then, `num_jobs_mod` will contain totals 1, 1, 3, 2, 4, and 4 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells. If,

```
subtract
old        num_jobs / class broad_age * sex
constants  0,2
real       num_jobs_mod / class  broad_age * sex
```

then, `num_jobs_mod` will contain totals 1, 0, 3, 2, 5, and 4 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells.

Note that these rules imply the special case in 8.5.1.3: when the CONSTANT list contains a single value, that value is used as the second operator for each cell of the matrix.

8.5.1.5 Joint Use of OLD and MODIFY with CONSTANTS

In the previous examples, the operations are on an OLD variable and constants to produce an newly defined variable. OLD and MODIFY may appear together instead, provided that no new variables are simultaneously introduced. The MODIFY variables are entirely overwritten without regard to their previous contents. In the simplest example

```
ADD
OLD         a
CONSTANT    b
MODIFY      c
```

by adding *a* to *b* and placing the result in *c* without regard to the previous contents of *c*.

All the extensions to cross-classifications apply similarly. For example,

```
add
old        num_jobs / class sex * broad_age
constants  0,1,2
modify     num_jobs_mod / class  sex * broad_age
```

revises a previously existing `num_jobs_mod` to contain totals 1, 2, 4, 5, 7, and 8 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells. The calculation uses the values of `num_jobs` without changing them, and overwrites without taking note of the previous contents of `num_jobs_mod` .

8.5.1.6 Use of MODIFY Alone with CONSTANTS

If no OLD or new variables appear, the results will operate on the information in the MODIFY variable.

```
ADD
MODIFY      a
CONSTANT    b
```

by adding *a* to *b* and placing the result back into *a*. Again, previous rules for matrices and lists of constants of different lengths apply. For example,

```
add
modify     num_jobs / class sex * broad_age
constants  0,1,2
```

revises `num_jobs` to contain totals 1, 2, 4, 5, 7, and 8 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells.

8.5.1.7 Lists of Variables

The previous rules apply to lists of variables, such as

```
DIVIDE
OLD         alist / CLASS  clist1
CONSTANTS   blist
REAL        clist / CLASS  clist2
```

where *alist* and *clist* are lists with equal numbers of variables. The rules are applied for each *alist*/*clist* pair of variables, reusing the constants in *blist* for each pair separately. The above summary shows a single OLD statement and a single statement defining new variables, but multiple statements may be used for either or both, as long as the combined number of old variables matches the combined number of new variables. For example, for

```
SUBTRACT
OLD         alist1 / CLASS  clist1
OLD         alist2 / CLASS  clist2
OLD         alist3 / CLASS  clist3
CONSTANTS   blist
REAL        clist1 / CLASS  clist4
REAL        clist2 / CLASS  clist5
```

the combined number of variables in *alist1*, *alist2*, and *alist3* must match the combined number of variables in *clist1* and *clist2*.

Again, previous rules for matrices apply. The rules are similarly extended to joint use of OLD and MODIFY or MODIFY alone.

As before, MODIFY, and new variable declarations cannot appear together. The allowed combinations with CONSTANTS are OLD and new, OLD and MODIFY, or MODIFY alone.

As an example,

```
add
old        num_jobs / class sex * broad_age
old        num_jobs / class broad_age
constants  0,1,2
real       num_jobs_mod / class  sex * broad_age
real       num_jobs_age / class broad_age
```

creates `num_jobs_mod` to contain totals 1, 2, 4, 5, 7, and 8 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells and `num_jobs_age` to contain 3, 8, and 13 for the three levels of age.

8.5.2 ADD, MULTIPLY, SUBTRACT, DIVIDE without CONSTANTS

Without CONSTANTS, at least one OLD variable must be declared. The last OLD variable is used in the same manner as the data would have been used after CONSTANTS. In all other respects, the same rules apply. Consequently:

> If new variables are created, then there must be one fewer new variable than OLD variables.
>
> If variables are modified, then there must either be:
>
> > a single OLD variable (in which case the information in the MODIFY variables is used in the calculation), or
> >
> > one more OLD variable than new one (in which case the information in the MODIFY variables is overwritten).

Again, MODIFY cannot appear with the creation of new variables.

In principle, all the remaining rules of this Section 8.5.2 derive from these rules. Nonetheless, the balance of the section will work through the application of these rules to clarify them and to increase the utility of this documentation for reference.

8.5.2.1 Basic Operations with Single Values ADD operates on 2 real variables *a* and *b* producing a new variable *c*:

```
ADD
OLD         a, b
REAL        c
```

by adding *a* to *b* and placing the result in *c*. The other subroutines are similar:

```
MULTIPLY
OLD         a, b
REAL        c
```

multiplies *a* and *b* and places the result in *c*;

```
SUBTRACT
OLD         a, b
REAL        c
```

subtracts *b* from *a*, (i.e., *a* - *b*), and places the result in *c*;

```
DIVIDE
OLD         a, b
REAL        c
```

divides *a* by *b*, (i.e., *a* / *b*), and places the result in *c*.

8.5.2.2 Rules for Matrices of Matching Length ADD operates on a real variable *a* cross-classified by one or more classes in *clist1* (which uses the * notation for product in the case of more than one class) with the same number of elements of *b* producing a real variable *c* cross-classified by *clist2*, which must have the same number of implied elements:

```
ADD
OLD         a, b / CLASS clist1
REAL        c / CLASS clist2
```

by adding *a* to the corresponding element in *b*, element by element, and placing the result in *c*. The order of elements in *a* and *b* is determined by applying FORTRAN order to the specification in *clist1*, i.e., varying the first class the most quickly, the next class the next most quickly, etc. Similarly, the elements of *c* are also varied in FORTRAN order.

These rules also include the case that *a* and *b* are given different /class specifications but still have the same number of specified elements:

```
ADD
OLD        a / CLASS  clist1
OLD        b / CLASS  clist2
REAL       c / CLASS  clist3
```

As an example, suppose `num_jobs` and `jobs_suppl` are real variables cross-classified by the class variables `sex` and `broad_age` in three broad age groups. If the totals in `num_jobs` are 1, 2, 3, 4, 5, and 6 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells, respectively, and the totals in `jobs_suppl` are 0, 1, 2, 0, 0, and 0 for the same cells, then

```
add
old        num_jobs jobs_suppl / class sex * broad_age
real       num_jobs_mod / class  sex * broad_age
```

creates a new variable `num_jobs_mod` with totals 1, 3, 5, 4, 5, and 6 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells. If, instead,

```
add
old        num_jobs / class broad_age * sex
old        jobs_suppl / class sex * broad_age
real       num_jobs_mod / class  broad_age * sex
```

then, `num_jobs_mod` will contain totals 1, 2, 4, 4, 7, and 6 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells. Note that the TRANSFORM step tolerates the reversal in the order of class variables here, as long as the lengths of the resulting matrices are the same. This feature affords both programming flexibility and the opportunity to carry out inappropriate calculations!

In Exhibit 8.1, the DIVIDE subroutine at #1 employed element by element division to define `proom`, thus applying the rules of this subsection.

8.5.2.3 Rules for Matrices with Lists of Different Lengths The length of *b* may be less than the number of elements of *a* if it is an integer multiple, *m*. In that case, the first *m* elements of *a* are operated on by the first element of *b*, etc. The sizes for *a* and *c* must still agree.

As an example, consider num_jobs from Section 8.5.1.2 or 8.5.2.2. Consider jobs_suppl2 to contain 0, 1, and 2 for the 3 levels of age. Then,

```
add
old        num_jobs / class sex * broad_age
old        jobs_suppl2 / class broad_age
real       num_jobs_mod / class  sex * broad_age
```

creates a new variable num_jobs_mod with totals 1, 2, 4, 5, 7, and 8 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells. Note that in this first example, the sums involve consistent age groups, for example, the value for the first age group in jobs_suppl2 are added to matching age cells in num_jobs , etc. If, instead,

```
add
old        num_jobs / class broad_age * sex
old        jobs_suppl2 / class broad_age
real       num_jobs_mod / class  broad_age * sex
```

then, num_jobs_mod will contain totals 1, 3, 3, 6, 6, and 8 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells. In this case, the summation incorporates mismatches across age groups in forming the sums. If,

```
add
old        num_jobs / class broad_age * sex
old        jobs_suppl / class broad_age(1,3)
real       num_jobs_mod / class  broad_age * sex
```

then, num_jobs_mod will contain totals 1, 4, 3, 6, 5, and 8 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells.

As a general remark, when combining matrices of different sizes, it is usually the case that one expects the class variables at the end of each class list to agree in order to produce sensible results, although exceptions to this remark can occur.

These rules imply the special case that when the last OLD variable contains a single value, that value is used as an operator for each operation.

8.5.2.4 Joint Use of OLD and MODIFY Without CONSTANTS

In the previous examples, the operations are on OLD variables to produce an newly defined variable. OLD and MODIFY may appear together instead, provided that no new variables are simultaneously introduced. If the number of OLD variables is one more than the number of

MODIFY variables, the MODIFY variables are entirely overwritten without regard to their previous contents. In the simplest example

```
DIVIDE
OLD          a, b
MODIFY       c
```

by dividing *a* by *b* and placing the result in *c* without regard to the previous contents of *c*.

All the extensions to cross-classifications apply similarly. For example,

```
add
old         num_jobs / class sex * broad_age
old         jobs_suppl2 / class broad_age
modify      num_jobs_mod / class  sex * broad_age
```

revises a previously existing `num_jobs_mod` to contain totals 1, 2, 4, 5, 7, and 8 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells. The calculation uses the values of `num_jobs` and `jobs_suppl2` without changing them, and overwrites without taking note of the previous contents of `num_jobs_mod` .

8.5.2.5 Use of MODIFY Alone Without CONSTANTS

If a single OLD variable appears, the results will operate on the information in the MODIFY variable.

```
ADD
MODIFY        a
OLD           b
```

by adding *a* to *b* and placing the result back into *a*.

The OLD variable is always the second operator, regardless of the order in which the OLD and MODIFY statements appear. Thus, both

```
DIVIDE
MODIFY        a
OLD           b
```

and

```
DIVIDE
```

14.30

```
OLD          b
MODIFY       a
```

divide *a* by *b*.

Again, previous rules for matrices and lists of constants of different lengths apply. For example,

```
add
modify     num_jobs / class sex * broad_age
old        jobs_suppl2 / class broad_age
```

revises num_jobs to contain totals 1, 2, 4, 5, 7, and 8 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells.

8.5.2.6 Lists of Variables

The previous rules apply to lists of variables, such as

```
ADD
OLD         alist / CLASS  clist1
REAL        clist / CLASS  clist2
```

where *alist* contains one more variable than *clist*. Except for the last variable in *alist*, the rules are applied for each *alist*/*clist* pair of variables, reusing the last variable in *alist* for each pair separately. The above summary shows a single OLD statement and a single statement defining new variables, but multiple statements may be used for either or both, as long as the combined number of old variables exceeds by 1 the combined number of new variables. For example,

```
DIVIDE
OLD         a, b
OLD         c
REAL        e, f
```

divides *a* by *c* and places the result into e, and it divides *b* by *c* and places the result into *f*. As a programming practice, some users may prefer to set off the last OLD variable in a separate OLD statement of its own in order to call attention to its special role in the calculation.

Again, previous rules for matrices apply. The rules are similarly extended to joint use of OLD and MODIFY or MODIFY alone.

As before, MODIFY, and new variable declarations cannot appear together. The allowed combinations without CONSTANTS are OLD and new, or OLD and MODIFY.

As an example,

```
add
old         num_jobs / class sex * broad_age
old         num_jobs jobs_suppl2 / class broad_age
real        num_jobs_mod / class  sex * broad_age
real        num_jobs_age / class broad_age
```

creates num_jobs_mod to contain totals 1, 2, 4, 5, 7, and 8 for the (1,1), (2,1), (1,2), (2,2), (1,3), and (2,3) sex by age cells and num_jobs_age to contain 3, 8, and 13 for the three levels of age.

8.5.3 A Comparison of ADD, SUBTRACT, MULTIPLY, and DIVIDE in the CREATE and TRANSFORM Steps

Because of their ability to work with matrices, ADD, SUBTRACT, MULTIPLY, and DIVIDE have far more complex roles in the TRANSFORM step than the CREATE step. Nonetheless, the following comparison helps to illustrate the roles of OLD, MODIFY, and new variables in the TRANSFORM step:

	CREATE Step	*TRANSFORM Step*
1.	constant 2 into b divide a by b into c	divide old a constant 2 modify c divide old a constant 2 derived c
2.	constant 2 into b divide a by b	divide modify a constant 2
3.	divide a by b into c	divide old a b modify c

```
                                        divide
                                        old a b
                                        derived c

4.   divide a by b                      divide
                                        modify a
                                        old b
```

Exhibit 8.2 A comparison of DIVIDE in the CREATE and TRANSFORM steps

The forms 1 and 3 in the CREATE step each have two counterparts in the TRANSFORM step, depending on whether the target variable has been previously defined.

8.5.4 Treatment of Missing Values in ADD, SUBTRACT, MULTIPLY, and DIVIDE

As noted previously in Section 4.7, the value -98765.432109 is a missing value indicator on VPLX files. This value is similarly used during the TRANSFORM step. Generally, if either operator in an addition, subtraction, multiplication or division is missing, then the result becomes missing - the single exception to this rule is that the product of a missing value and 0 is defined to be 0 for MULTIPLY.

If the divisor in DIVIDE is 0, the outcome is missing.

8.5.5 An Example

The following example incorporates some of the available features of the arithmetic subroutines:

```
comment  EXAM19


comment  This example also begins from EXAM11 but uses both tenure
         and persons_cl as class variables to illustrate features of
         ADD, SUBTRACT, MULTIPLY, and DIVIDE in the TRANSFORM step.

create  in = exampl11.dat  out = exampl11.vpl

input    rooms persons cluster tenure

      4 variables are specified

format   (4f2.0)

comment  The input data set contains
 5 7 1 2
```

```
 6 8 2 2
 5 2 3 1
 4 1 4 2
 8 4 5 1
 8 2 6 1

class  tenure (2/1,3) 'Renter' 'Owner/other'

labels  rooms 'Number of rooms' persons 'Persons'
        tenure 'Tenure'

class  persons into persons_cl (1-2/3-4/5-high) '1-2' '3-4' '5 or more'

    (Simple) jackknife replication assumed

    Size of block   1  =             18

    Total size of tally matrix =     18

    Unnamed scratch file opened on unit 13

    Unnamed scratch file opened on unit 14

    End of primary input file after obs #       6

transform  in = exampl11.vpl out=exam11a.vpl

comment  Make renter/owner comparisons for rooms expressed both
         as differences of means and as ratios of means

subtract                                                                                      #1

old    mean ( rooms   ) / class tenure (1) * persons_cl (0-n)

old    mean ( rooms   ) / class tenure (2) * persons_cl (0-n)

derived  mean_diff_r    / class persons_cl (0-n)

    (assigned to block   2)

divide

old    mean ( rooms   ) / class tenure (1) * persons_cl (0-n)

old    mean ( rooms   ) / class tenure (2) * persons_cl (0-n)

derived  ratio_df_r    / class persons_cl (0-n)

    (assigned to block   2)

comment  Cell by cell, multiply total rooms for owners by .5 and those
         by renters by 2

multiply                                                                                      #2

old   rooms / class persons_cl * tenure

constants   2 .5
```

```
real  rooms_adj / class persons_cl * tenure

    (assigned to block   3)

comment  Add 10 to the total for tenure (2) * persons_cl (2) only

copy                                                                                          #3

old   rooms  / class persons_cl * tenure

real  rooms_mod / class persons_cl * tenure

    (assigned to block   3)

****  BEGINNING OF SUBROUTINE   5

add

modify  rooms_mod / class tenure (2) * persons_cl (2)

constant   10


display

option   ndecimal=2

    Unnamed scratch file opened on unit 13

list     n(1) total (rooms rooms_adj rooms_mod )
                   / class persons_cl (1-3,0) * tenure /
         total (rooms rooms_adj rooms_mod mean_diff_r ratio_df_r  )
                   / class persons_cl (1-3,0)



    persons_cl             :   1-2
    Tenure                 :   Renter

                                                     Estimate      Standard error

Sample N (wtd) for block   1                             1.00                1.00

Number of rooms         :  TOTAL                         4.00                4.00

rooms_adj               :  TOTAL                         8.00                8.00

rooms_mod               :  TOTAL                         4.00                4.00


    persons_cl             :   3-4
    Tenure                 :   Renter

                                                     Estimate      Standard error

Sample N (wtd) for block   1                              .00                 .00

Number of rooms         :  TOTAL                          .00                 .00
```

rooms_adj	:	TOTAL	.00	.00
rooms_mod	:	TOTAL	.00	.00

persons_cl : 5 or more
Tenure : Renter

			Estimate	Standard error
Sample N (wtd) for block		1	2.00	1.26
Number of rooms	:	TOTAL	11.00	7.00
rooms_adj	:	TOTAL	22.00	14.00
rooms_mod	:	TOTAL	11.00	7.00

persons_cl : TOTAL
Tenure : Renter

			Estimate	Standard error
Sample N (wtd) for block		1	3.00	1.34
Number of rooms	:	TOTAL	15.00	6.88
rooms_adj	:	TOTAL	30.00	13.77
rooms_mod	:	TOTAL	15.00	6.88

persons_cl : 1-2
Tenure : Owner/other

			Estimate	Standard error
Sample N (wtd) for block		1	2.00	1.26
Number of rooms	:	TOTAL	13.00	8.54
rooms_adj	:	TOTAL	6.50	4.27
rooms_mod	:	TOTAL	13.00	8.54

persons_cl : 3-4
Tenure : Owner/other

			Estimate	Standard error
Sample N (wtd) for block		1	1.00	1.00
Number of rooms	:	TOTAL	8.00	8.00
rooms_adj	:	TOTAL	4.00	4.00
rooms_mod	:	TOTAL	18.00	8.00

persons_cl : 5 or more
Tenure : Owner/other

		Estimate	Standard error
Sample N (wtd) for block	1	.00	.00
Number of rooms :	TOTAL	.00	.00
rooms_adj :	TOTAL	.00	.00
rooms_mod :	TOTAL	.00	.00

persons_cl : TOTAL
Tenure : Owner/other

		Estimate	Standard error
Sample N (wtd) for block	1	3.00	1.34
Number of rooms :	TOTAL	21.00	9.77
rooms_adj :	TOTAL	10.50	4.88
rooms_mod :	TOTAL	31.00	9.77

persons_cl : 1-2 #4

		Estimate	Standard error
Number of rooms :	TOTAL	17.00	8.26
rooms_adj :	TOTAL	14.50	7.84
rooms_mod :	TOTAL	17.00	8.26
mean_diff_r :	VALUE	-2.50	1.94*
ratio_df_r :	VALUE	.62	.20*

persons_cl : 3-4

		Estimate	Standard error	
Number of rooms :	TOTAL	8.00	8.00	
rooms_adj :	TOTAL	4.00	4.00	
rooms_mod :	TOTAL	18.00	8.00	
mean_diff_r :	VALUE	(M)	-	#5
ratio_df_r :	VALUE	(M)	-	

persons_cl : 5 or more

			Estimate	Standard error
Number of rooms	:	TOTAL	11.00	7.00
rooms_adj	:	TOTAL	22.00	14.00
rooms_mod	:	TOTAL	11.00	7.00
mean_diff_r	:	VALUE	(M)	-
ratio_df_r	:	VALUE	(M)	-

persons_cl : TOTAL

			Estimate	Standard error
Number of rooms	:	TOTAL	36.00	4.10
rooms_adj	:	TOTAL	40.50	9.35
rooms_mod	:	TOTAL	46.00	4.10
mean_diff_r	:	VALUE	-2.00	1.29
ratio_df_r	:	VALUE	.71	.14

Exhibit 8.3 Illustration of Arithmetic Subroutines in the TRANSFORM Step

8.6.1 RECIPROCAL

Works either with:

OLD and new variables, in which case the results are stored into the new variables.

OLD and MODIFY variables, in which case the calculations are done on the OLD variables and overwrite the MODIFY variables.

MODIFY variables only, in which case the calculations use the contents of the MODIFY variables.

Without constants, the calculation produces element by element reciprocals. With CONSTANTS, the constants are used in the numerator of the calculation and the variable in the denominator, consequently opposite of DIVIDE.

8.6 Additional Arithmetic Subroutines

paired add, paired subtract, paired multiply, paired divide

Does not use constants; instead of using the last OLD variable as the second operator, it changes both operators, in pairs

power to raise the first operator to the power given by the second, may use constants

rmultiply highly specialized, differs from multiply by multiplying both the total and count of cases of real with missing and crossed real variables by the second operator (MULTIPLY only multiplies the total)

log reciprocal modifyrepf

8.7 Other Frequently Used Subroutines

collapse

copy reformat

glue

saveful

rprint, repprint

repread
repwrite
binaryread
binarywrite

NOTES