2. Recoding Variables in CREATE

2.1 Overview

This chapter covers the most common statements appearing in the recode section of the CREATE step. Additional statements and features are described in Volume III.

Section 2.2 describes arithmetic assignment statements and elementary arithmetic operations in VPLX. In fact, they barely need introduction, since they are essentially identical to conventions of SAS and Fortran 90. Almost equally familiar are the functions in section 2.3, which may be used in arithmetic assignment statements.

Section 2.4 introduces set assignment statements, which do not have a ready analogue in either SAS or Fortran (*1*). These statements are generally not required for simple applications, but they facilitate operations on a series of variables. Many aspects of the VPLX syntax employ the notion of sets, denoted by standard set notation, "{ }".

Section 2.5 describes conditional statements in VPLX. These statements follow almost identically the syntax of Fortran 90, including the enhancements to FORTRAN 77 (*2*) to allow use of symbols (">", "<", *etc.*) expressing arithmetic relations.

Section 2.6 describes the PRINT statement, which already appeared in examples in Chapter 1.

For completeness, section 2.7 describes specialized functions, including MISSING. These functions are more complex than those in section 2.3 and are required less frequently. The reader may thus regard this section as optional.

2.2 Familiar Arithmetic Assignment Statements

SAS, FORTRAN 77 (except for semicolons), and Fortran 90 (with optional semicolons) follow the same basic syntax for arithmetic assignment statements. Although there are some important differences between statements of a computer language and equations in conventional mathematical notation (*3*), generally the languages follow mathematics closely. For completeness, this section reviews the rules of the syntax, even though most readers will be able to skim this material rapidly and safely.

Example i1-1 in Exhibit 1.2 includes two simple arithmetic assignment statements:

I.2.2

```
room_ratio = rooms/persons;

    overcrowded = 1;
```

A number of similar examples appear in example i1-3. Except for the conventions for variable names, each of these statements could easily fit into a SAS or Fortran program. Some observations about arithmetic statements:

C The left-hand side names a variable that is created or revised as a result of the calculation.

C The right-hand side is an arithmetic expression. The expression may mix constants and variables.

  In VPLX, variables on the right-hand side must be defined by previous statements. Neither SAS nor Fortran place this condition (*4*). A variable may be defined by inclusion on an input list or appearance on the left-hand side of a previous arithmetic statement.

C If the variable on the left-hand side of the equation has been previously defined, it may be used on the right-hand side as well. For example,

```
count = count + 1;
```

  Regardless of the complexity of the expression on the right-hand side, the value of the left-hand side variable is not changed until the entire expression has been evaluated.

The basic arithmetic operators are:

+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Exponentiation. The two asterisks must be contiguous.

Additionally, “-” and “+” may also be used as unary operators, as in the following examples:

```
r10 = -a ;

r11 = +a;
```

This usage is restricted to the beginning of an expression or expression enclosed by parentheses.

An arithmetic expression may contain several arithmetic operators. In SAS, Fortran, and VPLX, parentheses "( )" may be used to establish the order of operations. For additional clarity, VPLX allows brackets "[ ]" to be used interchangeably with parentheses for this purpose.

When the order of operations is not established by parentheses or brackets, then the arithmetic operators are interpreted according to a precedence order. Multiplication and division are given a higher precedence than addition and subtraction. In turn, exponentiation takes precedence over the other four. Addition and subtraction are treated as equivalent in this hierarchy, so that they are simply evaluated from left to right. Similarly, multiplication and division are also evaluated from left to right. Unlike Fortran, VPLX does not accept compound exponentiation, a**b**c, without an explicit indication of order with parentheses, (a**b)**c or a**(b**c).

Exhibit 2.1 illustrates some features of arithmetic expressions.

```
a=1; b=2; c=3; d=4;

r1 = a + b + 1.5;    ! r1=4.5

r2 = 1+.5*d+1.5;     ! r2=4.5

r3 = a - b + 1.5;    ! r3=.5

r4 = b**2;           ! r4=4

r5 = b**d ;          ! r5=16

r6 = a*b + c*d;      ! r6=14

r7 = a*(b + c)*d;    ! r7=20

r8 = a*b/c*d;        ! r8=2.666..

r9 = (a*b)/(c*d) ;   ! r9=.16666..

r10 = -a;            ! r10 = -1

r11 = +a;            ! r11 = 1  but r11=a; is both equivalent
                     !          and simpler.

print r1-r11 / options nprint = 1;
```

Exhibit 2.1 An extract from i2-1.lis. The full listing includes the output of the print statement to confirm the results shown as comments.

The example illustrates a number of points:

- r1 and r3 show how addition and subtraction are interpreted from left to right. A similar point about multiplication and division is made by r8.

- C r2 and r6 show multiplication taking precedence over addition.

- C Parentheses change the interpretation of r7 compared to r6 and r9 compared to r8.

Missing values. The basic arithmetic operators generate missing values in two situations:

- C Division by zero.

- C Exponentiation, a**b, if:
  - b=0 and a=0
  - b is a negative integer, and a=0
  - b <0, is not an integer, and a#0
  - b >0, is not an integer, and a<0

When part of an arithmetic expression becomes missing, then subsequent arithmetic operations using the missing value also produce missing values. Generally, any missing component of an arithmetic expression on the right-hand side will cause the entire expression to become missing. (The only exception to this rule is that the MISSING function, to be described in section 2.7, returns 1 rather than missing if it evaluates a missing expression.) If the final result of the arithmetic expression on the right-hand side of the statement is missing, then 0 will be stored into the real variable on the left-hand side.

2.3 Familiar Functions in VPLX

Although spellings differ slightly in some instances from SAS, Table 2.1 exhibits familiar functions available in VPLX.

Table 2.1 Functions in VPLX with familiar Fortran analogues.

Function	Syntax	Conditions Producing Missing Values
Square root	sqrt(x)	x < 0 or x missing
Absolute value	abs(x)	x missing
Natural logarithm (base *e*)	log(x)	x #0 or x missing
Exponentiation of *e*	exp(x)	x missing
Integer	int(x)	x missing

Exhibit 2.2 illustrates each of these.

```
a=1; b=2; c=3; d=4;

r1 = sqrt(d/2);      ! r1=1.414...

r2 = abs(b-c);       ! r2=1

r3 = log(4);         ! r3=1.386...

r4 = exp(r3);        ! r4=4

r5 = int((d+a)/b) ; ! r5=2

r6 = int (2.5) ;     ! r6=2

r7 = int (-2.5) ;    ! r7=-2

print r1-r7 / options nprint = 1;
```

Exhibit 2.2 An extract from i2-2.lis. The full listing includes the output of the print statement to confirm the results shown as comments.

2.4 Set Assignment Statements

Several VPLX statements use "{ }" to denote an ordered set. Set assignment statements allow a set of variables to be assigned values simultaneously. As a simple example,

```
{x1 x2 x3 x4} = 0;
```

stores 0 into all four variables on the left-hand side. More generally, a set of variables may be equated to an arithmetic expression on the right-hand side,

```
{x1 x2 x3 x4} = [a/b + c/d]/[1/b+1/d];
```

stores the same weighted average of a and c into all four variables on the left-hand side.

Sets may be used on the right-hand side as well. On the right-hand side, the sets may be composed of a mixture of variables and constants,

```
{x1 x2 x3 x4} = {a b c d};
  {y1 y2} = {x1 0} ;
```

Example i1-3 in the previous chapter included several examples.

```
{c_pp_mis1 - c_pp_mis12} = {pp_mis4 - pp_mis15};
{c_rrp1 - c_rrp12} = {rrp4 - rrp15};
{c_age1 - c_age12} = {age4 - age15};
```

Each set is composed of 12 variables. Within a set, ranges of variables may be denoted in the same manner as SAS. For example, the first set assignment in the previous example is equivalent to

```
{c_pp_mis1 c_pp_mis2 c_pp_mis3 c_pp_mis4 c_pp_mis5 c_pp_mis6
 c_pp_mis7 c_pp_mis8 c_pp_mis9 c_pp_mis10 c_pp_mis11 c_pp_mis12} =
  {pp_mis4  pp_mis5  pp_mis6  pp_mis7  pp_mis8  pp_mis9
   pp_mis10 pp_mis11 pp_mis12 pp_mis13 pp_mis14 pp_mis15};
```

The right-hand side of a set assignment may consist of an arithmetic expression that combines sets, constants, and single variables,

```
{x1 x2 x3 x4} = 100* {a b c d};
  {y1 y2} = (x1+x2)*{x1 1}/(x1+x2+x3) ;
```

Although a set assignment may often be decomposed into a series of arithmetic assignment statements as in section 2.2, there is an important difference. All right-hand side computations are completed before any variable on the left-hand side is changed. Thus, the set assignment expresses a single operation on a vector rather than a series of individual equations. For example, if a is defined but r1-r4 have not yet been defined, then

```
{r1 - r4} = {a r1 r2 r3} + 1 ;
```

is not a legitimate set assignment and will produce an error condition, even though

```
r1 = a+1; r2 = r1+1; r3=r2+1; r4=r3+1;
```

is acceptable as four separate assignment statements. Under these circumstances, however,

```
{r1 - r4} = a + {1 2 3 4} ;
```

is acceptable and accomplishes the same purpose.

As a notational shortcut, if a set on the right-hand side is identical to the set on the left-hand side, including with respect to order, then the abbreviation {*} may be used on the right-hand side,

```
{x1 x2 x3 x4} = 100* {*};
```

Following the outline from section 2.2, the basic rules for set assignments are:

- The left-hand side names a set of variables that are created or revised as a result of the calculation.

- The right-hand side is an arithmetic expression. The expression may mix constants, variables, and sets composed of constants and variables with the same number of elements as the left-hand side.

  Variables on the right-hand side, whether by themselves or included as elements in a set, must be defined by previous statements. A variable may be defined by inclusion on an input list or appearance on the left-hand side of a previous arithmetic statement.

- If any variable in the set on the left-hand side of the equation has been previously defined, it may be used on the right-hand side as well. Regardless of the complexity of the expression on the right-hand side, none of the values of the variables on the left-hand side is changed until the entire expression on the right-hand side has been evaluated.

Example i2-3 illustrates features of set assignments.

```
a=1; b=2; c=3; d=4;

{r1 r2} = sqrt(d/2) ;          ! r1=r2=1.414...

{r3 - r6} =  100 * {a b c d}
              / (a+b+c+d) ;    ! r1=10 r2=20 r3=30 r4=40

{r7 - r8} = log ({1, b}) ;     ! r7 = 0 r8=.693...

print r1-r8 /
   options nprint = 1;

{r3 - r6} = {*}/(r3+r4+r5+r6); ! r1=.1 r2=.2 r3=.3 r4=.4

print r3-r6 /
   options nprint = 1;

{r11-r14} = a +{1 2 3 4};      ! r11=2 r12=3 r13=4 r14=5

print r11-r14 /
   options nprint =1;
```

Exhibit 2.3 An extract from i2-3.lis. The full listing includes the output of the print statement to confirm the results shown as comments.

2.5 Conditional Statements

2.5.1 An Example Example i1-1 in Exhibit 1.3 included the following example of a conditional,

```
if persons > rooms then;                 ! Note: overcrowded will be
   overcrowded = 1;                      ! 0 or 1.
end if;
```

If there are more persons than rooms, the variable overcrowded will be set to 1, otherwise it will remain unchanged (at the default value of 0 in this instance). This section is organized into subsections to cover each aspect of the syntax governing IF and related statements.

2.5.2 Arithmetic Relationships VPLX provides six arithmetic relationships, the first five of which are syntactically identical to Fortran 90. The sixth is similar to an operator in SAS.

Table 2.2 Logical operators in VPLX. In all three cases in which two characters are used to express the relationship, they must be consecutive, that is, without intervening blanks. No symbol is available for .in..

Relationship	Standard Representation	Symbolic Representation
Equals	.eq.	= =
Greater than	.gt.	>
Greater than or equal	.ge.	>=
Less than	.lt.	<
Less than or equal	.le.	<=
Is a member of	.in.	

The VPLX implementation differs from Fortran in an important respect, however. Similar to SAS, VPLX treats any real variable within 10^{-12} of an integer as that integer. The quantities before and after the relation are each examined for such closeness to an integer. When they are equated to the integer, the result of the comparison may deviate from the result given by Fortran, which does not have such a convention concerning closeness of a real number to an integer.

If either arithmetic expression to the left or right of the first five relationships, or to the left of the sixth is missing, then the relationship will be missing, that is, neither true nor false.

The last relationship, .in., differs from the others. A variable or arithmetic expression is expected to the left of .in., but a set of constants must follow on the right. For example,

```
if rooms .in. {3,4,5} then;
   rooms3_5 = 1;
end if;
```

The condition is satisfied if rooms takes the values 3, 4 or 5. Ranges may be expressed within the set, so that the following is effectively equivalent to the preceding example as long as rooms is an integer (*5*),

```
if rooms .in. {3-5} then;
   rooms3_5 = 1;
end if;
```

2.5.3 Range Specifications As the previous example illustrates, range specifications may use "-" within sets. Within a set, the "-" does not correspond to the arithmetic operator. More generally, arithmetic operations are not allowed within sets. The conventions used to interpret ranges in the set following .in. are used in other parts of the language, such as with CLASS and CATEGORICAL statements described in section 3.

Sets may mix single values, such as "1" and bounded ranges, such as "1-3". Additionally, two forms of open ended ranges, such as low - 1 and 1 - high, are also accepted, where the 1 could be replaced by any other constant. A set may include more than one such element. Commas may be used to prevent ambiguities:

```
if   x1 .in. {1-3} then;

if   x1 .in. {1,-3} then;
```

the first of the two checks the single range from 1 to 3 while the second checks for the numbers 1 or -3 (*6*). When negative values are involved, single values and ranges must be set off by commas.

The following may be used in range specifications within sets:

- **low** - to indicate the lowest possible number, when used as a lower bound in a range. For example, `low - -50` indicates any number less than or equal to -50.
- **high**- to indicate the highest possible number, when used as an upper bound in a range. For example, `100000 - high` indicates 100,000 or higher.
- **res** - (for "residual") to represent any value not previously classified. It excludes, however, instances in which the expression is missing. This range is not generally used with IF statements, but instead with CATEGORICAL, and CLASS statements in Chapter 3.

In interpreting ranges, VPLX implements the SAS convention that any number within 10^{-12} of an integer is that integer.

If the upper end of a range includes a decimal fraction, it is implicitly continued with 9's instead of 0's, e.g. the range (1-1.9) will include any number from $1\text{-}10^{-12}$ to $2\text{-}10^{-12}$ in the interval. Similarly, anticipating the syntax used for CAT and CLASS statements in chapter 3, (1.0-1.09/1.1-1.19/1.2-1.29) will assign values from $1\text{-}10^{-12}$ to 1.0999999... to the first category, 1.1 to 1.1999999... to the second, and 1.2 to 1.29...... to the third. Special care must be taken with this convention, however, for negative values (*7*).

Lower ends of ranges are not altered except by the imposition of the rule that any number within 10^{-12} of an integer is that integer. Thus, an integer at the lower end is extended downward by 10^{-12}, as illustrated in the previous example. Non-integer lower ends are not altered.

2.5.4 Logical Operators VPLX includes three logical operators, .and., .or., and .not.. The first of these, .and., is applied to two logical expressions and is true only if both expressions are true. The second, .or., provides an inclusive or of two logical expressions. It will yield true even if only one condition is true and the other false or missing. The third, .not., changes a single expression from false to true and true to false, leaving missing unchanged.

Parentheses ‘( )’ or brackets ‘[ ]’ may be used to specify the order of logical operations. In their absence, a hierarchy of operations exists. .And. takes precedence over .or.. For example,

```
if a == 1 .and. b == 2 .or. c == 4 .and. d == 3 then;
   r1 =1;
end if;
```

is the same as,

```
if (a == 1 .and. b == 2) .or. (c == 4 .and. d == 3) then;
   r1 =1;
end if;
```

but different from,

```
if a == 1 .and. (b == 2 .or. c == 4) .and. d == 3 then;
   r1 =1;
end if;
```

.Not. takes precedence over both .and. and .or..

2.5.5 IF-THEN-ELSE Blocks VPLX allows blocked IF-THEN-ELSE blocks similar to Fortran or SAS. Statements following ELSE are executed only if previous conditions have not been satisfied. For example,

```
if persons > rooms then;
   overcrowded = 1;
else if persons == rooms then;
  borderline = 1;
else;
  less_dense = 1;
end if;
```

Only one of the three variables, overcrowded, borderline, or less_dense, will be set to 1 by these statements. If a variable, such as overcrowded, did not exist prior to the block, then it is given the initial value 0 if the corresponding conditions are not met, for example, if persons is not greater than rooms.

In general, the construction begins with an IF statement of the form:

```
IF  [(] logical expression [)] THEN ;
```

One or more VPLX commands may then follow, which will be executed for the observation only if the logical expression is true. Optionally, one or more ELSE IF statements may then follow:

```
ELSE IF  [(] logical expression [)] THEN ;
```

The statements that follow are executed only if the condition is true and if the conditions for all previous IF or ELSE IF statements are false. Whether or not any ELSE IF statements are used, a single

```
ELSE;
```

may follow. The subsequent VPLX statements will be executed if the previous conditions have not been met. In all cases, however, the sequence must be ended by

```
END IF ;
```

One END IF (or ENDIF) must appear for each initial IF. Note that this syntax is closer to Fortran than to SAS, particularly since a single END IF is used here instead of the multiple END statements appearing in the SAS syntax.

As in Fortran or SAS, IF blocks may be nested within other IF blocks. Exhibit 2.4 provides an example, where the block of statements following the first IF includes a complete IF-THEN-ELSE construction nested inside, including its concluding END IF statement. Indentation of the code corresponding to the nesting of conditions improves the readability of the code. Like SAS and Fortran, VPLX does not assign meaning to the degree of indentation.

If an error occurs in forming the nested blocks, such as i) a missing IF statement, ii) an ELSE IF statement following an ELSE statement as part of the same IF block, or iii) a missing END IF, VPLX will list the input command file with interspersed indications of the depth of nesting of the IF blocks, referred to as the “IF level” in the diagnostic listing. The “Total nesting” in the listing is identical to the “IF level” in this case. When WHILE and DO are added to the VPLX language, then the “Total nesting” will account for all nesting of IF, WHILE, and DO (*8*). These annotations may be used to help find the programming problem. Exhibit 2.5 shows an example.

```
a=1; b=2; c=3; d=4;

if a/b < c/d then;              !  This is true
   test = sqrt (c/d - a/b) ;
   if test < 1 then;
      r1 = 1;                   !  r1 = 1;
   else;
      r2 = 1;
   end if;
else if a == 1 then;            !  Never evaluated
   r3 = 1;                      !       "
end if;

print r1 - r3 /                 ! r1 =1; {r2 r3} = 0;
      options nprint = 1;
```

Exhibit 2.4 An extract from i2-4.lis. The full listing includes the output of the print statement to confirm the results shown as comments. Since logical expression following the first IF is true, the condition a == 1 following ELSE IF is never checked, even though it is also true.

```
  a=1; b=2; c=3; d=4;

  if a/b < c/d then;              !  This is true
  <IF level:  1  Total nesting:  1>

     test = sqrt (c/d - a/b) ;

     if test < 1 then;
     <IF level:  2  Total nesting:  2>

        r1 = 1;                   !  r1 = 1;

     else;
     <IF level:  2  Total nesting:  2>

        r2 = 1;

  else if a == 1 then;            !  Never evaluated
  <IF level:  2  Total nesting:  2>

VPLX ERROR CODE VP1025

VP1025 -  ELSE IF following ELSE is not allowed.

          (Technical: called from CMPAR1 (through CMPRNT))

OUT= FILE DELETED
```

Exhibit 2.5 An extract from i2-5.lis. The result of omitting the END IF statement for the nested IF block following the first condition is to leave an ELSE IF following an ELSE, as noted by the error message. The “<IF level:...” annotations indicate the depth of nesting, and show in this case that the offending ELSE IF statement is at the same depth, 2, as the preceding ELSE statement. This helps to identify that the imbedded IF block was not closed by an END IF.

Exhibit 2.5 illustrates a VPLX error message. VPLX has identified a fatal error and reported a specific error code, VP1025. VPLX will then read the detailed text of the error message from an ASCII file, VPLERROR.TXT, and include it in the print file, if VPLERROR.TXT has been properly installed (*9*). If VPLX does not print the detailed message, it is possible to find it by searching the ASCII file VPLERROR.TXT for the specific code, VP1025 in this case, with any editor.

2.6 PRINT

The PRINT statement has appeared extensively in previous examples. There are two basic forms:

```
PRINT varlist ;
```

or,

```
PRINT varlist / option nprint = n  ;
```

where *varlist* is a list of variables and *n* is an integer constant. The second form prints only the specified number of cases. Previous examples in this section employed nprint = 1 when a calculation began with “a=1; b=2;”, *etc.*, to avoid printing the same outcomes multiple times. Both are executable statements, that is, they are executed at the point in the calculation indicated by the program.

Either form may be used with IF. For example,

```
if (weight - baseweight) / weight > .25 .or.
   (weight - baseweight) / weight < -.10 then ;
      print surveyid weight baseweight / option nprint = 100;
end if;
```

will print only cases with large relative differences between weight and baseweight. Note that when weight = 0, the logical expressions will be missing and no printing will occur. Alternatively,

```
if (weight - baseweight) / weight > .25 .or.
   (weight - baseweight) / weight < -.10 .or.
   (baseweight > 0 .and. weight == 0) then ;
      print surveyid weight baseweight / option nprint = 100;
end if;
```

could include cases under the third condition. Note that precedence order for .and. over .or. means that the use of parentheses is not required in the expression, but they serve to clarify the meaning of the code.

2.7 Miscellaneous Functions

2.7.1 MISSING The missing function returns 1 if an arithmetic or logical expression is missing and 0 if it is not. Consequently, the evaluation of the MISSING function never itself produces a missing value. Example i2-6 illustrates these points.

```
a=1; b=2; c=3; d=4;

if 1/(b-2*a) > 0 then ;          ! The calculation returns missing
   r1 = 1; end if;               ! so r1 = 0

if .not.(1/(b-2*a) >0) then ;    ! Because the logical expression
   r2 = 1; end if;               ! within the parentheses is missing,
                                 ! not does not change this, so r2 = 0

if missing (1/(b-2*a)) then ;    ! missing returns 1 (= true) so r3 = 1
   r3 = 1; end if;

if missing (1/[b-2*a]>0) then ; ! Because the logical expression
   r4 = 1; end if;               ! is missing, r4 = 1

if missing(.not.(1/[b-2*a] >0)) ! Similarly, r5 = 1
     then ;
   r5 = 1; end if;

if missing ( missing (1/[b-2*a]) )
     then ;
   r6 = 1; end if;               ! Trick example, r6 = 0

print r1 - r6 / options nprint=1;
```

Exhibit 2.6 An extract from i2-6.lis, illustrating application of MISSING to both arithmetic and logical expressions. The full listing includes the output of the print statement to confirm the results shown as comments.

The MISSING function thus provides a means to handle special conditions under which the results of a calculation might be missing. There is also a *real with missing* type, described in Volume III, chapter xx, that may be useful for extensive problems potentially involving missing values, but the placement of this topic in the third volume suggests that this approach is seldom required.

2.7.2 IDCHANGE This function of one or more variables returns 1 the first time it is called and on each occasion in which one or more of the variables have changed value since its previous call. It considers a variable unchanged if its value is within 10^{-3} of the previous value. It does not assume that the variables are ordered.

Each separate instance of IDCHANGE in a program keeps track of its own list of previous values.

As a programming hint, it is possible to consider survey IDs of up to 12 digits or so without causing problems of precision, but longer IDs should be expressed as two or more variables to limit the length of each. IDCHANGE will accept an arbitrarily long list of variables.

```
create  in = i1-1.dat  out = vplx1.vpl ;  ! Beginning of CREATE step


  input  rooms persons  weight cluster    ! input statement to read
          stratum /format (5F2.0) ;       ! data  ("section #1")

  if idchange (stratum) then; print stratum ; end if;

  if idchange (stratum cluster) then;
     print stratum cluster ; end if;

    Stratified jackknife replication assumed

    Size of block   1  =             3

    Total size of tally matrix =     3

**** End of CREATE specification/beginning of execution
PRINT request    1
     stratum            1.0000
PRINT request    2
     stratum            1.0000          cluster             1.0000
PRINT request    2
     stratum            1.0000          cluster             2.0000
PRINT request    1
     stratum            2.0000
PRINT request    2
     stratum            2.0000          cluster             3.0000
PRINT request    2
     stratum            2.0000          cluster             4.0000
PRINT request    1
     stratum            3.0000
PRINT request    2
     stratum            3.0000          cluster             5.0000
PRINT request    2
     stratum            3.0000          cluster             6.0000
```

Exhibit 2.7 An extract from i2-7.lis, illustrating application of IDCHANGE.

NOTES

1. There are, of course, ways to accomplish the same tasks as the set assignment statement through a number of statements in both languages.

2. The official spellings are FORTRAN 77 and Fortran 90. VPLX continues to be written in FORTRAN 77. This documentation uses “Fortran” generically.

3. Two examples of this difference are the requirement that multiplication be explicitly denoted by '*' in the computer languages and the acceptance of statements such as “x=x+2;” as valid executable statements in the computer languages. As a mathematical equation, x=x+2 has no real solution.

4. VPLX does not have a GO TO feature, which both SAS and Fortran do. With GO TO, a valid program may be written where left-hand side variables are defined by subsequent program statements. Both SAS and Fortran consequently allow variables on the right-hand side to be undefined or possibly defined by subsequent statements.

   SAS will execute the program in this circumstance but generally make the left-hand side variable missing if any of the variables on the right-hand side are missing at the time of execution.

   Most Fortran compilers report as a diagnostic the use of variables on the left-hand side that are not defined by any statement in the program unit. If a left-hand side variable is defined by a statement in the program, but during execution has not been defined by the time of its use in the statement, the result is unpredictable in the Fortran standard, although specific Fortran implementations may default to 0.

5. VPLX intervals such as 3-5 expand the upper limit to just below the next integer, so the second example actually is true for all values of rooms between 2.999999999999 and 5.999999999999. The form of the first statement would look for values of rooms within tolerance of the integers 3, 4, and 5 only. For integer rooms, the results are identical, but for noninteger rooms the second condition could be true more often than the first form of the example.

6. This statement is generally correct, but a more precise statement requires the rules on decimal extensions in the previous note. The range 1-3 actually becomes the range $1\text{-}10^{-12}$ to $4\text{-}10^{-12}$. Although the range comes close to 4, it does not include it, and any number read as 4 from a file will be recognized as not falling within 1-3. Similarly, (1,-3) checks for $1\text{-}10^{-12}$ to $1+10^{-12}$ and $-3\text{-}10^{-12}$ to $-3+10^{-12}$.

7. In the current implementation, the same rules are applied regardless of whether the bounds are positive or negative. This creates an asymmetry, however. For example, {-10 - -1} denotes any value from $-10\text{-}10^{-12}$ to -10^{-12}, whereas {1 - 10} denotes any value from $1\text{-}10^{-12}$ to $11\text{-}10^{-12.}$

8. IF, WHILE, and DO must be jointly considered to determine whether they are properly nested. There is not a fixed date for the addition of WHILE and DO.

9. Installation instructions are in Appendix A2. In Windows 95/NT, for example, the vplerror.txt file has to be present as c:\vplx\vplerror.txt (the use of upper or lower case in the spelling of the file name has no effect in this environment). Under circumstances in which the error file cannot be installed to this folder/directory, an alternative is to begin each VPLX command file with a statement “errorfile *filename*;” where *filename* gives the full path to the error file. The use of this statement is shown in i2-5.crd and i2-5.lis, even though the given filename is the default in this case.