Vilno Table Programming Language

White Paper

Robert James Wilkins

"Produce Complex Statistical Tables"
"Five Times Faster than SAS or Excel"

Vilno Table Programming Language White Paper

December 22, 2011

The Vilno Table software product is currently in beta status, and the test cases in the appendix of this paper include statistical tables produced by the software product.

Each case-by-case example in the appendix is two pages: the statistical table being asked for, and the short program that produces it.

The Vilno Data Transformation software product is a separate older product, with documentation, available at :
http://code.google.com/p/vilno

The Data Preparation White Paper, which will present research and development suggestions for data preparation tools to benefit researchers, is in the pipeline.

The Vilno Table programming language is designed to produce complex statistical tables very quickly and upon request.
A worker using Vilno Table can get his work done more than five times faster than a worker using the SAS programming language or the Excel spreadsheet.
For many examples, a table that requires more than 500 lines of SAS code, needs less than 25 lines of Vilno Table code.
The impact on worker productivity is huge.

****

This statistical programming language is well suited to statisticians and researchers who face this dilemma :
1: You often have a statistical table request
2: You want a rapid response, like, today
3: But resources available to you are more constrained than 20 years ago. Essentially, there are far fewer SAS programmers in your building than in the 1990s. When possible, the SAS programming work has been outsourced. But just-in-time requests don't work well with this arrangement, the necessary paperwork and budget request wouldn't be processed this week, for sure.

In addition, other statisticians and researchers in your building are bombarding the in-house SAS programmer with their own requests: and some of them have enough clout to tell the programmer that they are more important than you.
And because a typical statistical table request can often require several hundred lines of code, there's a good chance the programmer will tell you “I can't do it today”.

Point is – you've got a serious problem.
The Vilno Table programming language is the solution to your problem.
With this new tool, a statistical table request that used to require 5 hours of work will often now require less than twenty minutes.

****

A skeptic might ask “I don't believe it. How is this possible?”
Let's put this as two questions:
How is this possible?   , and
Why is this possible?

If you want to tackle the “how” question first (and that's the more practical approach), I'll give you a few pointers here, and you can go through the examples in the appendix at the back of this white paper.
That's actually the pragmatic way to learn a new programming language: case by case , example by example.
If you're curious about the “why” question, I'll get to that in a second ; first, here are a few helpful tips to assist in reading the actual examples.

There are 16 examples, provided in the last section of this paper, that are actually produced by the current beta-status software product. Each example is two pages: the program that produces the statistical table (usually under 25 lines of code), and the statistical table itself. (If you prefer, you can also look at the “miniature examples” (like on training wheels), on pages 10-13, they are for illustration (and not produced by the beta software product)).

****

A good way to grasp how this language does what it does, is to look at each example (one at a time, of course) , and with that example in front of you, ask yourself four questions:

A: What is the table going to look like going across?
Look at the column statement (note: the keyword col is short for column)

B: What is the table going to look like going down?
Look at the row statement

C: What advanced statistical procedures do you want for this table?
Look at the model statement (lm is linear model, chisq is chi-square).
Also, the names of the desired statistics are typed in the col and row statements (that includes both summary and advanced statistics : n mean % p-value F-statistic etc.).

D: What do you want to count?
Here I'm going to cheat just a little. Just assume each example is one of two situations:
D1: There's only one dataset. You are counting rows in that dataset.
D2: There's a dataset called “PATINFO”. If that's the case, and the table includes N or %, then there is either one dataset (PATINFO) or there are two datasets (PATINFO and another dataset (perhaps called ADVERSE_EVENTS or LABS)). Just assume PATINFO has one row per person, and assume you are counting people. (I'll cover counting techniques for complex data sources later.)

The above four guidelines are good enough to start with.
YOU CAN GO STRAIGHT TO THE EXAMPLES IN THE APPENDIX WITH THAT.

****

Bye.

….

Welcome back.

Why is this possible?

There are several interlocking reasons, but let's explain four reasons:

1: The gap between the level of abstraction implicit in the conversation between the statistician and the programmer, and the level of abstraction at which the chosen programming language operates.

2: The relationship between programming languages and tree data structures

3: Counting

4: A language compiler is different from a language macro system.

LEVEL OF ABSTRACTION

First of the four, consider the mode of abstraction. Basically, the programming language needs to be as close to the language at work as practically possible. This is a general guideline, and it doesn't tell you the compromises and trade-offs that a programming language designer has to make, the devil is in the details.

There's a gap between the level and type of abstraction used by the syntax of the programming language and the level and type of abstraction used by people who work in a certain subject area every day.
If the programming language designer promises to make the gap all but disappear, he'll have to break that promise. But you can and should make the gap as small as practically possible. (A programming language cannot be at the same level as the “spoken work language” , at least not precisely, because language between humans is nuanced, subtle and sometimes ambiguous ; whereas a programming language is technically precise (and by necessity, has precise technical limits)).

SQL is a good example of this: if the marketing of SQL databases over the years is an indicator, one of their intentions was to make SQL very English-like, enough so that it's usable by non-programmers.
If that was one of the goals of the designers of SQL, they failed. How could they not fail, given the difference between programming languages and human languages? But in striving towards that goal, SQL became a better language, and they did succeed in shrinking that gap considerably (when compared to older database interfaces).
A language design involves trade-offs and compromises, you're never going to satisfy everyone. If the SQL team had not tried to shrink that gap, they would have gotten a programming language with rather poor trade-offs. It was important that they try, even though failure was guaranteed.

TREES
One important reason the Vilno Table programming language boosts productivity is the way in which the programming language manages and expresses tree data structures to properly capture and connect the various concepts contained in a statistician's request.

When a statistician walks into a programmer's cubicle with a statistical table request, the request is typically a mock-up table(sketched quickly in pen) and a paragraph of english. That request needs to become code. If the programming language was not properly designed for this work, it needs many hundreds of lines of code. The Vilno Table programming language was designed for this work, the SAS programming language was not.

You can't solve this problem, if you don't get what the problem is. What is a statistician's statistical table request?
A statistician's request is a tree data structure. Is he conscious of this? Possibly not. When a statistical table is sent to the printer(or the screen), the tree data structure is projected to a two-dimensional data structure, rows and columns. But a statistical table is not a two-dimensional matrix. A statistical table is a tree. (This also has practical implications for Excel users, depending on the problem being solved.)

He could also ask for a graph or a data listing. A graph is also a tree. A data listing, like a table, looks like a two-dimensional data structure, but for a different reason: it actually is a two-dimensional data structure.

Consider spreadsheet software, such as Excel or OpenOffice. Consider the way in which the user communicates with a spreadsheet, and you see a spreadsheet thinks of itself as a two-dimensional matrix (A1,B6,etc.), not as a tree. For some types of customized statistical tables, that's bad news for worker productivity.
And if you are a big fan of OLAP, you might think : your friend is asking for a hyper-dimensional cube. No, not really. It's not a three-dimensional data structure that's being requested, nor a four-dimensional one, but a tree-dimensional data structure.

****

Trees matter, a lot.
When assessing the benefits and drawbacks of programming languages, general-purpose and some specialized, the way in which the chosen programming language connects to tree data structures is of paramount concern. And because trees are generalized generic data structures, two programming languages can be very versatile in managing trees, but in two very different ways, making them suitable for two different targeted subject areas.

Take the Lisp programming language. In Lisp, everything is a tree, which can be good or bad. For subtle high-level problems, Lisp is very powerful. For doing more routine tasks, Lisp crams a forest-ful of trees down the end-user's throat.

Or Python. Python is powerful and flexible (even if not quite as powerful as Lisp), yet is easy to learn and use. In order to strike that tricky balance, the choice the designer made for how to control tree data structures was of pivotal importance.

In a lot of scenarios (but not all), the manner in which a tree data structure is used, is best conceptualized as follows: position the tree upside down (root node points upwards, the leaves point downwards). The tree is hovering over what's called a “DOMAIN”, typically representing a problem to be solved, or a subject area of interest (a subject matter from which problems to be solved emerge).

The domain could be a single dataset or a database. It could be a source code file (a programming language parser produces an abstract syntax tree) or another type of text file (as with HTML or XML).

It depends on the context, but a tree sometimes has the role of an eye, or a view – a way of seeing things. And it's possible to have two different trees hanging over the same domain, representing two different viewpoints.

The choice of domain can be correlated with the choice of mode of abstraction.

Consider two choices for domain: a dataset or a paragraph. Summary statistics can be calculated by scanning the dataset, but the paragraph represents a statistical table request. One scenario is a tree over the dataset. The other scenario is a tree over the paragraph – meaning a choice to approach the problem at a higher mode of abstraction. Vilno Table makes this choice.

++========++

There is a tree-management technique that is different from the way trees are manipulated by Python or Lisp (which I'll call here the *-tree method) in the following ways:
The tree itself is explicitly purposed for visual display.
The tree has visual multiplication characteristics, usually denoted by the “*” operator, so that
(A B)*(C D) has 4 leaves ; and a node that's a category name will be multiplied by the range of categorical levels (sex expands to sex*(female male)) .

In the past, the *-tree method has been used to scan a single (fairly homogeneous) dataset (like a two-dimensional survey dataset). But this chosen mode of abstraction is neither subtle enough nor powerful enough for a wide variety of customized statistical table requests.

It's generally believed the first users of the *-tree notation may have been employees of the U.S Bureau of Labor Statistics in the 1970s , to execute tabulation of survey data on a mainframe.

COUNTING THINGS

Vilno Table is a lot better at counting things than either SAS or SQL SELECT. For N-% tables that are customized and draw upon more complex data sources, the syntax of SAS(or SQL) operates at too low a level of abstraction. The result is that a customized table, even without mathematically advanced statistics(p-value, least square mean, etc.), still requires hundreds of lines of code.

To code an N-% table, you have to control :
A: what are you counting ?
B: how is what you're counting being filtered?
Vilno Table allows programmers to express answers to this with just a few lines of code.

But with older tools, you're forced to spend too much time expressing lower-level logic in a sequence of steps to get to the final destination. And since you're forced into so many detours, the chance of a mistake is higher.

THE DIFFERENCE BETWEEN A MACRO SYSTEM AND A COMPILER

A programming language macro system is different from a programming language compiler, and the two tools are suitable for tackling different sorts of problems. This has practical implications for finding solutions to worker productivity problems in many areas ; for example, this difference between macro systems and compilers has presented a major stumbling block for the pharmaceutical industry for two or three decades.

The Vilno Table programming language is a solution to the worker productivity problem for producing customized statistical tables. The SAS MACRO system is not. You can't just hand a SAS MACRO manual to the end user. What's he going to do with it, write a bunch of macros? In this context, SAS MACRO is more like Perl : a tool for a developer, who's given time to come up with a solution, and a chance to sell it to the end-user. This is key: he has to make the case that this new macro library is worth the hassle. This packaged solution would be a candidate for a SML(standard macro library). The SML must be flexible and comprehensive enough so that the end user is never compelled to alter the internals ; only the external syntax (the calling APIs) is for end-user use. But just like any specialized programming language, this SML has to be evaluated in terms of : usability(readability and learning curve), worker productivity, flexibility and reliability.

Pharmaceutical macro developers have had 25 years to come up with this SML (and the assumption in much of the pharmaceutical industry has been that a solution, if it exists, would be some type of SAS MACRO library ; alternative directions of research and development have not been vigorously explored). Now, in 2011, two questions : Does this SML exist? If so, does it meet the end-users' needs for usability, productivity, flexibility and reliability?

What you cannot do is offer a range of “SML”s to the end-user (to compensate for the realization that no single SML candidate is a clean yet fully comprehensive solution). Why should he learn the external syntax of several, if you haven't even persuaded him to learn one?

Moreover, if each pharmaceutical company has it's own in-house “standard” macro library, this is neither a standard nor a solution to the productivity problem. While products in the open market (both open source and proprietary) are not without flaws, in-house corporate applications have a very long history of poor usability characteristics. While limited research and development resources have been cited in the past**, politics is also a reason : a poor design is more likely if the designer knows the outside world will never see it.

Moreover, if the end-user has no say in the matter (he has to use the in-house product whether he wants to or not) then there is no need to sell it to him (to make a convincing case). The problem with that is it greatly reduces the incentive to get the design right in terms of usability and productivity, (actually, similar political logic applies if the product is a packaged(not in-house) product and the sales process focuses on an executive who will never actually be the end-user, again the incentive to accurately assess usability and productivity aspects is diminished because of political reasons).

** Joel Spolsky, Five Worlds is a good essay on this, although he gives shrinkware too much of a free pass.

As I noted in the introduction, a lot of fairly complex statistical tables can be done in under 25 lines of code.
But, in order to teach you this language, I don't want you to look at 25 lines. I want you to look at 3 lines of code.

Is it possible to look at just THREE lines of code, and look at the resulting statistical table, and from that see clearly how the syntax of this new programming language produces the desired table?

Absolutely. And that fact makes the Vilno Table language much easier to learn.

Those critical 3 lines of code are:
The COLUMN statement
The ROW statement
The MODEL statement

This approach is particularly feasible if the end-user is fairly familiar with the incoming database and the variable names in the incoming database have a self-explanatory spelling (“patient_id” instead of “pid”).

One complicating detail is that for certain more complex tables that involve N and %, an experienced end-user looking at the table might ask “What exactly is being counted here?”. For the introductory examples in the next few pages, just assume we are counting people. Later I'll explain the THING statement , in the “N , %, Counting” section (pg 21). (It's part of the section of code that describes the input datasets. Essentially, it's a line of code that says “this is what is being counted”.)

Let's get started: for each of the following miniature examples , you'll see how 2 or 3 lines of code produce the desired statistical table. (For a table with only summary statistics, and no advanced statistics, no model statement is required).

Let's take a look, turn the page …

Miniature Examples :

```
col trt*(n %) ;
row gender ;
```

will become the table :

```
                 Trt
            _______________
            Placebo  Roopla
            _______  ______
            N   %    N   %
Gender
  Female
  Male
```

```
col trt*(n %) ;
row all bodysys*(all nothave prefterm) ;
```

for the following Adverse Event table:

```
                         Trt
                    _______________
                    Placebo  Roopla
                    _______  ______
                    N   %    N   %
All
Body System
  Nervous System
    Have AE
    Not have AE
    Preferred Term
      Headache
      Jitters
```

Well, OK, but what if we want to add a categorical p-value in a column on the right (such as chi-square or Fisher's)?
Easy:

```
model chisq(trt*thisrow?*n) ;
col trt*(n %) pvalue ;
row all bodysys*(all nothave prefterm) ;
```

How about a couple of linear model examples:

```
model lm(response = city) ;
col city*(t_pw pval_pw) all*(fvalue pvalue) ;
row city ;
```

```
          City  , Pairwise Comparisons
         _____________________________
         Bonn      Berlin      Bern           Overall
         _______   _______     _______    ___________________
         t  Pval   t  Pval     t  Pval    F-statistic P-value

City
  Bonn
  Berlin
  Bern
```

Well, what if I want only one pairwise comparison, Bonn vs Berlin?
(note that there will only be one row of statistics in this table.)

```
model lm(response = city) ;
col all*(t_pw("Bonn","Berlin")  pval_pw("Bonn","Berlin") )
    all*(fvalue pvalue) ;
row all ;
```

```
           Bonn vs Berlin                     Overall
         _____________________________    ___________________
         t-statistic   P-value            F-statistic P-value
```

How about summary statistics such as average?

```
col n mean(testscore) std(testscore) median(testscore) ;
row region ;
```

```
        N     Mean   StdDev   Median
        ____  _____  ______   ______

Region
  East
  North
  South
  West
```

Note that for summary statistics(N % mean sum), the model statement is not needed.

The next table is tracking statistics for patients and events:

```
col n n(evt) ;
row eventtype ;
```

```
                 # Patients    # Events
                 __________    ________
Specific Events
  Hot Flush
  MI
  Stroke
  Blood Clot
```

This is an unusual table, because the request is for counting two different types of things: people and adverse event records.

DIFFERENT TYPES OF TABLE FACTORS

Column and row statements consist of table factors, connected by white space (T next to T) and * (T under T) (along with parentheses for grouping table factors together). In a nutshell, the different types of table factors are : the CATs , the STATs, the RESTRICTs, and everything else.

There are categorical table factors, statistical argument table factors (both summary and advanced statistics), boolean expression table factors, and special-purpose table factors.

The categorical table factors play a special role: they influence the visual arrangement of the table in a manner other table factors do not. There is space in the table to print the category name(“Severity”), but underneath the category name, there is space to print every categorical value (“Mild” “Moderate” “Severe”). The category table factors affects how big the statistical table needs to become; the other table factors do not have this effect.

The statistic table factors (N % mean pvalue etc.) are also central: what's the point of a table that doesn't show any statistics?

Restricts are boolean expressions, such as [age<65] or [pat has pterm==”Headache”], that evaluate to true or false.

Some other special-purpose table factors include ALL, HAVE, NOTHAVE. HAVE and NOTHAVE are advanced features explained later. ALL is like a “void” table factor, essentially like a placeholder in the table. ALL does no data filtering (in contrast to categorical and restrict factors). A typical use of ALL is , for example:

```
row   ...*( all  [age<50] )*(N %)
```

There are some special-purpose table factors that are expressions that can be either a table factor or a top-line statement(or footnote statement, i.e. typed outside of the column and row statements). Examples of that:

```
col ...*model.3*...  ;
```

which refers to the third model footnote statement

or

```
col  ...*chisq(A*B*N)*...
```

which is a model statement positioned as a table factor (instead of as a footnote statement)

An expression in the position of a table factor might not apply to the entire table, because some parts of the table are unrelated to that table factor. The same expression as a footnote statement(either typed just above or just below the column and row statements) is , in a sense, more global : the whole table has access to it.

SUMMARY STATISTICS

A table factor can be the name of the desired statistic, either a summary statistic (N % mean std min max sum median) or an advanced statistic (such as pvalue). If it's an advanced statistic, a model expression must also be provided (such as lm(y~a+b+a*b) or chisq(trt*race*N)), but a summary statistic does not need a model statement (although the % statistic may need a denom statement).

N and %, normally thought of as very simple statistics, can be surprisingly complex, because of the way the data source could be structured. The “N and %” section in this paper explains this.

If a summary statistic is specified, then it's calculated by sub-groups, as indicated by the categorical factors for this part of the table. If there are restrict factors for this part of the table, only observations for which the boolean expressions are true are included to calculate the summary statistics. Here's an example: [age<50]*Gender*(mean(weight) std(weight)).

```
col T1  T2*T3  T4*(T5 T6) ;
row S1  S2*S3 ;
```

will become the column-tree (small-version and big-version):

```
            TOP-OF-COL-TREE
____________________________________________
  T1-NODE    T2-NODE         T4-NODE
__________  _________  _____________________
             T3-NODE    T5-NODE    T6-NODE
            _________  __________ __________
```

```
            TOP-OF-COL-TREE
____________________________________________
  T1-NODE    T2-NODE     T4(i.e. Gender)
__________  _________  _____________________
             T3-NODE    Female      Male
            _________  __________ __________
                        T5   T6    T5   T6
                        __   __    __   __
```

Before I explain how this language produces complex statistical tables, I'll need to illustrate how the column and row statements utilize tree data structures, or just enough for now. Later in this paper, I'll finish this discussion on how tree data structures are managed. We'll use the above simple example.

The essential gist : the syntax in the column (or row) statement specifies a tree data structure (and not quite in the way that Lisp does(or Python lists do),because of the * operator). Moreover, when the related categorical values are accessed from the incoming data, the tree is forced to become bigger -because a tree node for each categorical value is inserted under the tree node for the categorical name. The expanded column tree is placed into the top of the page ; the row tree at the left side : the matrix of printed statistics is essentially a cross-product of these two trees.

Because the categorical values cause the tree to grow, there are two trees, the small-column-tree(without categorical values) and the big-column-tree(with categorical values).

When constructing the small-column-tree from the syntax of the column statement, white space (i.e. "T1 T2") means "next to"or "side by side"; and T1*T2 means T2 goes UNDER T1 .

With the above example: under the root "TOP-COL-TREE"node are 3 nodes: a T1 node, a T2 node, and a T4 node. Under the T1 node is nothing. Under the T2 node is just a node labeled T3. Under the T4 node are 2 nodes, a T5 node and a T6 node. This data structure is called the small-column-tree. The row statement renders the small-row -tree similarly.

Next, from the small-column-tree, we construct the big-column-tree; the categorical levels force the tree to become bigger:

Suppose T4 is a categorical table factor : for each categorical level, put a node (labeled with that categorical level) under the T4 node. Whatever sub-trees were under the T4 node in the small tree, place a distinct copy of them [these copies may also need to be made bigger if the sub-tree itself has a categorical table factor. When you have several categorical factors, this becomes a recursive tree-construction process!] under each new categorical level node. (Suppose T4 is Gender : Place a "Female"node and a "Male"node under the T4 node. Under the "Female"node, place 2 nodes, labeled T5 and T6 respectively. The "Male"node also gets 2 nodes underneath, labeled T5 and T6).

This data structure is the big-column-tree, it has been made bigger because of the categorical values. Also, the big-row-tree is done similarly.

Now I need to explain sections, column-sections, and row-sections ; let's go back to the small-column-tree. You can divide the statistical table into vertical portions called "column sections": A column section is equivalent to a path(from the top node to the leaf node) in the small column tree. Therefore, a column section also corresponds to a group of such paths in the big column tree, and therefore, a column-section also corresponds to a group of columns of statistics in the statistical table.
A row section is defined similarly, using the small row tree.

A section of the table is defined to be the intersection of a column section and a row section. If a table has N column sections and M row sections, then the table has N*M table sections.

Pick any two printed statistics in the statistical table: if their position in the table differs only by choice of categorical value, and they do not differ by any table factor in the column or row statement, then they belong to the same table section.

So, for this simple example, let's enumerate the column sections.
They are:
[T1]   [T2,T3] [T4,T5] [T4,T6]
The row sections are: [S1] [S2,S3]

This table has 8 (=4*2) table sections, which are:
[T1; S1]   [T1; S2,S3]   [T2,T3; S1]   [T2,T3; S2,S3]
[T4,T5; S1]   [T4,T5; S2,S3]   [T4,T6; S1]   [T4,T6; S2,S3]

So to summarize: a statistical table is a visual arrangement of three parts: the column tree at the top of the page, the row tree at the left margin, and a matrix of printed statistics. The matrix of statistics can be divided into table sections, because each section is the intersection of a column section and a row section. Also, each table section is naturally identified with a vector of table factors (such as [T4,T6,S2,S3]). Each table section can have, optionally, it's own computational process and data source, including data preparation sequence and choice of statistical analysis.

PARAGRAPHS OF CODE PRIOR TO THE TABLE PARAGRAPH

A program using the Vilno Table programming language can be divided into three main sections.

1: A paragraph of code that specifies the statistical table request. This is the TABLE paragraph, and for obvious reasons, is the main emphasis of this white paper.

2: If needed, data preparation paragraphs of code, that process the original datasets, to provide derived (ready-to-use) datasets. If the incoming datasets are ready for analysis, then data preparation code is often unnecessary, and the entire program will be quite short.

Data preparation has been something of a dilemma for decades. There are important problems that remain unsolved, even today, in part because the subject is dry and tedious : it's obvious there's a problem here, what's not obvious is who's going to fix it. Much research and development still needs to be done here. Yet the quality of statistical analysis can depend, in part, on the rigor of data cleaning.

The “Data Preparation White Paper” is currently in the pipeline, and will address the problems and possible angles of attack. The emphasis of the paper you reading is statistical table production.

The Vilno Table product's options for data transformation are covered on page 41 , and include:
A: direct access to the Vilno Data Transformation Engine
B: the PARITY paragraph

3: Code that describes the incoming datasets, which is explained in the next section.

INPUT DATA DESCRIPTION CODE

This section describes code that comes before (and is a prelude to) the table paragraph (not including data transformation paragraphs, dealt with separately). Most such code is input data description code.

To specify directories where datasets reside:

```
directoryref   a=”/home/tom/area1”  b=”/home/tom/area51”  ;
```

(datasets are then referred to as : a/demog_data , b/labs_data , etc)

If formats are used, first specify where the format file is :

```
formatfile   “/home/sue/formats.txt”  ;
```

For each input dataset , there's a line of code describing it :

```
inputdset  asc  b/lab_data  site patid visit testcode value units  ;
```

(First specify the storage format and name of the dataset. For the current beta software configuration, the format is either “asc” (ascii data file, with meta-data at the top of the file) or “vdt” (binary format for vilno data engine). Other data storage format options will be developed later.)

After the dataset name , specify the list of variable names. A variable name can come with additional information.
A variable name can be typed with an associated format:

inputdset asc b/lab_data   patid   value units~(format=units_format) ;
A variable name can be typed with declared categorical levels:
inputdset asc b/lab_data   patid   trtgroup~[0,20,40]   gender~[“Female”,”Male”] value ;
(There can sometimes be scenarios where such extra meta-data can be informative for the compiler.)

You can also use the inputdset statement to specify what the “parity” (or “row-spec”) of the dataset is:
inputdset asc a/patinfo site patid gender trtgroup 1*(site patid) ;
This states that site and patid are the index columns for this dataset, which is to say that for each value of (site,patid) there is at most one row in the dataset. [Here, site is Clinical-Site-Number and patid is Patient-Number, so that site and patid identifies a person.]

THING STATEMENT

The thing statement is a way of making the programmer's life a lot easier when drafting N-and-% tables. The thing statement describes a type of thing, within the input data, that can be counted. Multiple thing statements are allowed (although not common), because there can be different types of things. [In the current software version, a thing dataset must have one row per thing.] Here's the thing statement:

thing pat uniqval(site patid) a/patinfo ;

Here, PATINFO is the designated thing dataset. Now “PAT” is a type of thing, which can be counted, either in the context of the thing dataset itself (here, PATINFO), or in the context of a (probably larger) dataset into which rows from the thing dataset can be matched (ie: LAB_DATA or VITAL_SIGNS).

If you want this thing statement to be the default choice for N and %, do:
n ~ n(pat) ;

Otherwise, to use the PAT thing definition in the table, you must type N(PAT) or %(PAT) (just typing N or % will do a simple row count since you didn't reset the default).

Sometimes (as when using lm()), you must explicitly inform which variables are categorical, and which are continuous, which is done as follows:
categorical trtgroup gender ;
continuous   age weight ;

A few pointers: like the inputdset statement, the thing statement is considered as a description of the input data. Specifying the datatypes of variables in the inputdset statement is not required. Whether ascii file or vilno-engine-format, the front part of the datafile contains meta-data (variable names and datatypes). If two columns from two datasets share the same variable name spelling, they really should have the same characteristics: datatype(int, float, string) , categorical/continuous status , etc. There will be more data storage formats, besides “asc”(ascii data file) or “vdt”(vilno engine binary format).
A few other top-line statements, typically typed just before the table statements that actually have any real logic, are
title “This is a vital signs statistical table” ;   (title at top of the table)
printto “/home/tom/vitalstable” ;   (where the table is printed to)

N and % and COUNTING

What is being counted? In simpler cases, the input data source is a single dataset, and the rows are being counted.
Remember: each section of the table is associated with a vector of table factors, some of which are categorical factors, and some of which are restrict factors (true/false expressions).
What the statistic N means, depends – in a fairly obvious way – on the categorical and restrict factors for this section of the table: N is a subtotal count for each categorical level(or to be more precise, each combination of categorical levels, if there are 2 or more categorical factors), counting only observations for which the boolean expressions are all true.

For the % statistic , the numerator is the same as N. The denominator is similar, but the set of categorical and restrict factors that determine the calculation of the denominator are different: these are the categorical and restrict factors that are both present in the vector of table factors (for this section) and present in the DENOM statement.

Now, suppose a THING definition is active for this section of the table. Then it gets more complicated. The input data source may be a single dataset, or it might be a pair of datasets.
Why would this be?
You are counting things, where a THING is defined to be equivalent to a row in the designated thing dataset (or equivalently, equivalent to a distinct value of the thing key-column(s) (eg. “site” “patient_id”)).
But here's the catch : you are counting THINGS that have certain CHARACTERISTICS, and the information about those characteristics might or might not be available in the thing dataset (the dataset specified in the thing statement). So the counting process may need access to two datasets.

So if a THING definition is in use for this table section, the counting process must have access to the dataset referred to in the thing statement. This dataset we shall denote the P-dataset (or the thing dataset).
But, the counting process must also have access to a dataset that has all the variable names mentioned in the list of table factors for this table section (in particular, the categorical variables, and variables in the restricts). This dataset we shall denote the M-dataset (or the MAIN dataset). The main dataset is either identical to the thing dataset, or is a dataset into which the rows of the thing dataset can be merged in (or “pulled in”).

++================================++

A couple of additional points when calculating N and % and the M-dataset is different from the P-dataset:

The P-dataset is the THING dataset ; the thing definition essentially states what a “thing” (or “individual” or “experimental unit”) is considered to be. For calculating N and %, if it's not a “thing”, it's not counted. Because a “thing” is considered to be equivalent to a row in the P-dataset, rows in the M-dataset that do not match to a row in the P-dataset are ignored. (For the same reason, 5 rows in the M-dataset that match to the same row in the P-dataset are counted as 1 “thing” (not 5) .)

When the data source is two distinct datasets, the P-dataset and the M-dataset, then it is (we'll see later) sometimes helpful to classify categorical and restrict table factors as either M-level table factors or P-level table factors. A categorical or restrict table factor is an M-level table factor if it contains a variable name that is available only in the M-dataset (and not available in the P-dataset). A table factor is a P-level table factor if all it's variable names are available in the P-dataset.

STATISTICS THAT REQUIRE A MODEL STATEMENT

For summary statistics (such as N % median mean), no model statement is needed. For advanced statistics, such as p-values, the end-user must specify the desired model, using an M-expression (M~model), and the desired output argument (or statistic), using an O-expression (O~out-arg). The M-expression can be either a table factor or a model footnote. The O-expression is a table factor. As of this writing, the current version of the beta software product supports two models: lm (linear model) and chisq (chi-square).

For the linear model, the first (and often only) argument to the M-expression is the model equation :
lm(Y =A + B -1 + A*B)
The independent variables are a mixture of categorical and continuous(although use of the categorical and continuous statements described earlier is often optional, for the lm model it is required). The available statistics include: fvalue and pvalue (F-statistic and it's p-value) ; est, t_c, pval_c (estimate for effect for a continuous variable or a categorical level of a categorical variable (or interaction of said variables), with t-statistic and p-value) ; est_pw, t_pw, pval_pw (estimate for pairwise difference of effects , with t-statistic and p-value). The effect parameter can be a single variable name, or can be a product of variable names(and possibly a mixture of categorical and continuous), like A*B ; the effect parameter is needed(or inferred) for est, t_c, est_pw, t_pw but not fvalue and pvalue. Here are a few examples that should be clear to a reader familiar with linear models.
Assume the input dataset description code includes :
categorical A B C ; continuous V X Y Z ;

1st example:
model lm(Y=X+A) ;
col fvalue pvalue pval_c(X) A*pval_c(A) A*A*pval_pw(A) ;

++==++

The F-statistic could not be simpler(there's only one of them), so let's focus attention on the effect estimates(est t_c pval_c) and the pairwise estimates(est_pw t_pw pval_pw). Let's call the F-statistic group the zero-wise group of statistics, and call the effect estimates group the single-wise group of statistics. Let H be the number of (complex) categorical levels possible via the categorical variables with the effect parameter used for est or est_pw . (For example, the effect parameter is A*B*V : since V is continuous, ignore it. If A has 3 categorical levels and B has 4 categorical levels, then H=3*4=12 possible complex categorical levels). The number of zero-wise statistics is 1 . The number of single-wise statistics is (roughly) H. The number of pairwise statistics is (roughly) H*H . ("roughly" because there's a bit of hand-waving here, I'm ignoring details like the possibility of model overparameterization)

So each single-wise statistic needs a complex categorical level. Each pairwise statistic needs a pair of complex categorical levels. There are two choices: the spectrum of categorical levels is provided by the surrounding categorical table factors OR the end-user chooses just one categorical level , by typing it inside the O-expression :
A*t_c(A) or t_c(A,60)
A*A*t_pw(A) or t_pw(A,60-20)
A*B*t_c(A*B*V) or t_c(A*B*V, 60 "medium")

```
A*B*A*B*t_pw(A*B*V)  or  t_pw(A*B*V,  (60,”high”) - (60,”low”) )
```

++==++

For example, you may prefer to print the single-wise(coefficient estimate) statistics as columns, and the pairwise statistics as a square matrix:

```
model lm(Y=X+A) ;
col all*( est  t_c  pval_c )   A*( est_pw  t_pw  pval_pw ) fvalue pvalue ;
row A ;
```

(Of course fvalue and pvalue are not produced with A-type categorical levels, that works out fine: the F-statistic, with p-value, are printed towards the top right corner of the page, above the row categorical levels).

Or, if you want to see a really complicated example :

```
model lm(Y=A+B+V+A*B+A*B*V) ;
col all*( est  t_c  pval_c )   A*B*( est_pw  t_pw  pval_pw ) ;
row A*B ;
```

(Note that, if the effect parameter is not typed in the O-expression, as in est_pw(A*B), it can sometimes be inferred by the surrounding categorical table factors – but you must specify in the input dataset description code which variables are continuous and categorical.)

If the volume of statistics is too large, and you know beforehand which contrasts you need to see – type the desired categorical levels in the O-expressions:

```
col all*( est(A*B*V,60,”medium”)  pval_c(A*B*V,60,”medium”)  )
     all*( est_pw(A*B*V,(60,”high”)-(60,”low”))  pval_pw(A*B*V,(60,”high”)-(60,”low”))  ) ;
```

++=======================================++

Here's the chi-square test :

```
model chisq(A*B) ;
col A*N chisqvalue pvalue ;
row B ;
```

(Again, the chi-square statistics are not re-printed for each row categorical level, but appear towards the right top corner.)

Here's 3 chi-square tests on the same page (with N % statistics):

```
denom gender ;
model chisq(gender*thisrowcat*N) ;
col gender*(N %) chisqvalue pvalue ;
row A B C ;
```

(the top of the page is for GENDER vs A, the middle for GENDER vs B, and the bottom of the page is for GENDER vs C.)

Note the variator keyword : thisrowcat . For each table section, the variator is re-interpreted (to be A or B or C). If there's more than one row categorical factor in a table section, the innermost row categorical factor is used by default.

THE ONLYIF, SUBPOP, DENOM, AND MODEL STATEMENTS

Here we discuss some expressions that are usually in the table paragraph but outside of the column and row statements. Some of these expressions, however, can be placed inside the column(or row) statement, either as a table factor, or as a parameter inside a table factor.

When the expression appears as a top-line statement(above or below the column and row statements, sometimes referred to as a footnote statement), the expression is potentially applicable to the entire table(globally). When the expression is in a table factor, the effect is local.

The denominator statement impacts the % statistic:

```
denom  trtgrp  gender  [age<50]  ;
```

The denominator statement is a list of categorical and restrict factors, as the above example shows.

++====================================++

Consider a table section where the % statistic is requested. Consider the vector of table factors for this table section. The denominator factors are those categorical and restrict factors present in both this section's vector and the denominator statement. The denominator is calculated according to these denominator factors : within each categorical level (example, if there are 2 denominator categorical factors: within the group of people who are female and in the placebo group , etc.) and for observations for which the boolean expressions evaluate to true (these from the denominator restrict factors, example: [age<50]).

In the “N %” section the M-level dataset and the P-level dataset are explained. If the P-level dataset is different from the M-level dataset, a denominator factor has to be a P-level factor (it cannot use a variable name that is in the M-dataset yet not available in the P-dataset).

Note: the difference between the denominator and the numerator (or the N statistic) is simple: for the % numerator(which is the same as the N statistic) use all the categorical and restrict factors for this table section. The denominator uses only the categorical and restrict factors for this table section that also are mentioned in the denominator statement.

++=================================++

The model expression (such as lm(y=x) or chisq(A*B)) can be accessed in three ways:

1. As a table factor :
col ...*chisq()*... ;
2. As a footnote statement, when the table only needs one model expression :
model chisq(a*b*n) ;
col … chisqvalue pvalue ;
3. The table needs two or more model expressions : here, you can use option 1 (as table factors), but breaking them out as model footnote statements, underneath the col and row statements, may be easier

to read:

```
col ….. ;
row .. ;
model.1  chisq(...) ;
model.2  lm(....) ;
model.3  lm(...) ;
```

If you use multiple model footnotes, they will be indexed 1, 2, 3 and that indexing is used to match the output-arguments(statistics) in the column(or row) statement with the correct model footnote. Suppose we wish to refer to the second model footnote. There are two ways to do that:

```
col  … model.2 *( fvalue pvalue    … )   ;
```

or

```
col …  *( fvalue.2  pvalue.2    …. )   ;
```

++====================================++

The SUBPOP statement is a list of categorical and restrict factors that influences how the input data is delivered into a statistical model (such as lm() or chisq()). It is usually a footnote statement (i.e., typed outside the col/row statements), but it can also be a parameter to a model-expression.

The way in which the SUBPOP statement is interpreted is similar to the DENOM statement: consider a table section where an advanced statistic is requested (one that requires a model expression, i.e. lm,chisq); consider the vector of table factors for this table section. The subpop factors are those categorical and restrict factors present in both this section's vector and the subpop statement. Suppose for example,   the subpop factors are two categorical factors, called P and Q, and one restrict factor, [age<50]. Observations for which any subpop restrict factor are False are omitted. If P has 2 categorical levels, and Q has 3 categorical levels, then P*Q has up to 6 categorical levels : hence the model, such as lm() or chisq(), is executed separately for 6 subgroups.

Note: within each subgroup, P and Q are essentially fixed constant, so it does not make sense for P and Q to show up in the model expression's regression equation.

The subpop expression can be a stand-alone footnote statement, or inside a model expression :

```
subpop P Q [age<50] ;
```

or

```
model lm(Y=A+B+A*B  ,  subpop( P  Q  [age<50] ) ) ;
```

++=======================================++

(of course, a subpop statement can have multiple restrict factors, this example uses just one.)

One point should be made clear : the categorical and restrict factors for the table section of interest DO NOT have an automatic effect on the model (unless, of course, they are also mentioned in the subpop statement). Note the difference : categorical and restrict table factors DO have an automatic effect on summary statistics such as N and MEAN.

++===++

ONLYIF is a statement with a single restrict factor. A table paragraph can have more than one ONLYIF footnote statement. An ONLYIF expression can be as a stand-alone footnote statement, or as a parameter inside a model expression :

```
onlyif[ age<50 ] ;     (after col/row statements)
```

or

```
model lm(Y=A+B , onlyif[ age<50 ] ) ;
```

The onlyif statement plays a different role than a normal restrict table factor, and is less often used . It is not used as a table factor.

First, note what makes the onlyif statement different from DENOM or SUBPOP :
onlyif[age<50];
can have an active effect in table sections where “[age<50]” is not present as a restrict table factor. (By contrast, “denom [age<50] ;” (or “subpop [age<50] ;”) has no effect in table sections where “[age<50]” is not present as a table factor.)

However, there are sections of the table for which the onlyif statement has no effect, for the following reason : different table sections are allowed to have different data sources. The onlyif statement does not apply if it's restrict factor does not make any sense for the table section's data source.

NORMAL RESTRICTS , AND PH RESTRICTS

A restrict is a boolean expression, using variable names from the data source, used to filter data prior to statistical calculations. A restrict is usually in the role of a table factor, but can also appear as an onlyif expression. Example : [trtgroup==”Placebo” and weight<200]

Vilno Table allows for a special type of boolean expression, not typically seen in other programming languages, called here a Ph boolean expression. Here's an example:
[PAT has (visit>3 and weight>200)]
A Ph boolean expression is of the form (PAT has normal-boolean-expression) or
(PAT nothas normal-boolean-expression) where:
PAT is a type of thing established by a thing definition statement, and the normal boolean expression has variable names from a data source such that it's possible to pull rows from the thing dataset into that data source. Because it's possible to “pull-in” (or merge in) rows from the thing dataset into the data source, a row in the data source can “belong to” a unique row in the thing dataset. Hence the expression : “PAT has (some quality)”.

For example, suppose the thing dataset has one row per value of (site,patid) (and site and patid are of course variable names in the thing dataset), then the data source for the inner boolean expression could be a dataset with columns for site and patid (and usually allowed to have multiple rows per value of (site,patid)).
The data source can have additional columns through a “PULL-IN adjustment” (see page 35) (which includes columns from the thing dataset itself, hence the inner boolean expression can have variable names from the thing dataset).

A Ph expression can be the entire restrict factor, but can also be combined with other boolean expressions :
[PAT has (visit>3 and weight>200) or PAT has (prefterm==”hypertension”) or age>70]

++===================++

Since (PAT has (normal-boolean-expression)) has a distinct True/False value for each thing (ie for each row in the thing dataset), you can think of it as a pre-calculated boolean variable(true/false) that is a column appended to the thing dataset just before the table calculations are done. That has the following subtle implications:
Now, if you refer to the text that explains the difference between M-level table factors and P-level table factors (in the “N %” section , page 21) : since the inner boolean expression contains variable names from the M-dataset, you might conclude that [PAT has boolean-expression] is an M-level factor. But it's not, because (PAT has boolean-expression) is treated as a single synthetic variable name in the P-dataset (the P-dataset is the thing dataset, recall).
The upshot, (in a table section where the PAT definition is active) :
[prefterm==”Headache”] is an M-level table factor, BUT
[PAT has (prefterm==”Headache”)] is a P-level table factor

VARIATORS AND OTHER SPECIAL-PURPOSE STUFF

Here we discuss variator keywords, and a few other special-purpose keywords or table factors, that provide more flexibility. Their meaning is re-evaluated for each table section. (This chapter is advanced).

Variator keywords in the DENOM or SUBPOP statements: these include keywords such as THISROWCAT, TOPROWLEVEL, FACTORUNK.

THISROWCAT is the row categorical table factor. By default, if there is more than one such table factor, it's the innermost row categorical table factor. So, THISROWCAT could mean GENDER in one part of the table, and AGEGROUP elsewhere.

TOPROWLEVEL is the outermost row table factor, if it is a categorical or restrict table factor. If the outermost row table factor is not a categorical or restrict table factor, it is ignored (no effect on the denom or subpop filtering process). It is suitable when all the outermost row table factors are categorical or restrict table factors, or the ALL keyword:

```
row  ( all  [age<50]  [age>=50]  gender ) * ( …..... )   ;
```

FACTORUNK is translated to a synthetic restrict factor :

```
[ v1!=Null  and  v2!=Null  and  …..  ]
```

The restrict factor will include variable names in categorical and restrict table factors. For example, if the table section is :   gender*[age<50] , then the synthetic restrict factor is to include observations where gender and age have known(non-missing) values.

You can also include some variator keywords (N , src , THISROWCAT , THISROW?) inside a model expression : these translate to variable names, usually variable names in the incoming datasets, or synthetic or created variable names.

[src=bloodpres] is a special-purpose table factor that designates a chosen continuous variable name (not categorical) for a given table section.

```
row   ...*[src=bloodpres]*(mean min)  ;    is equivalent to :
row   ...*(mean(bloodpres)  min(bloodpres))  ;
```

HAVE and NOTHAVE

HAVE and NOTHAVE are table factors sometimes used when there is an active thing statement for the table section of interest, and the data source is a pair of distinct datasets (possible with the N and % statistics, see pg 21) . For illustration, let's use the following example: for this table section, the vector of table factors includes : TRTGROUP , BODYSYS , PREFTERM , [SEVERITY>1] . The statistic factor is either N or %. The data source is a pair of datasets , AE(adverse events) and PATINFO. PATINFO has just one row per person, AE can have more than one row per person ; therefore PATINFO is the P-dataset(the THING dataset) and AE is the M-dataset. TRTGROUP is in PATINFO; BODYSYS,PREFTERM,SEVERITY are in AE ; so the P-level tables factors are: just TRTGROUP ; and the M-level table factors are : BODYSYS, PREFTERM, [SEVERITY>1].

(Reminder, it is in the input data description code, at the top of the file, where the PATINFO dataset is declared to be the thing dataset :
thing pat uniqval(site patid) a/patinfo ;
n ~ n(pat) ;
This PAT definition states that a thing is equivalent to a row in the PATINFO dataset.)

Let's explain the HAVE table factor :
HAVE(AE) is essentially similar to a restrict factor : include only observations for which there is at least one matching row in the AE dataset. In fact, if AE had a column called INFLAG_AE(whose value was fixed to 1 for each row), then ...*HAVE(AE)*N has the same effect as ..*[INFLAG_AE==1]*N.

In addition, HAVE(AE) can help to identify the data source : if it were the case that there are no categorical or restrict table factors that use variable names from the AE dataset.

If the data source is already identified for this table section (because the thing definition active for this section identifies PATINFO as the P-dataset and variable names in the categorical and restrict table factors identify AE as the M-dataset) , then these three choices of table factors, HAVE or HAVE(AE) or ALL, essentially mean the same thing.

++==============================++

NOTHAVE

The NOTHAVE table factor does the logical reverse of the HAVE table factor. Let's start with the simplest example : no categorical or restrict factors, just NOTHAVE(a/AE)*(N %). This counts people (in PATINFO) who do not have any matching rows in the AE dataset. A slightly more complex example: [severity>1]*prefterm*nothave*(N %) . This means, for each disease category (say prefterm=”headache”), count the people who do not have moderate to severe headaches.

Now, going back to the more complex example at top of this page, let's show how NOTHAVE works in more detail:

Determine which categorical and restrict table factors are M-level factors and which are P-level table factors. (Remember, if the table factor has a variable name that's unavailable in the P-level dataset, it's an M-level table factor).

Next, for purpose of illustration, focus on a specific table cell within the table section, say, for trtgroup=”Placebo” and bodysys=”CNS” and prefterm=”Headache”.
Next, calculate N1 according to all the categorical and restrict table factors, as is normal with the N statistic, ie : the number of people in the Placebo group who have records in the AE dataset such that (bodysys=”CNS” and prefterm=”Headache” and severity>1).
Next, calculate N2 according to only the P-level categorical and restrict table factors (ignoring the M-level factors), ie : the number of people in the Placebo group.

Now, when the NOTHAVE feature is in use, the N statistic (and also the numerator in the % statistic) is calculated as N2-N1, ie: the number of people in the Placebo group who do not have any records in the AE dataset such that (bodysys=”CNS” and prefterm=”Headache” and severity>1).

The denominator of the % statistic , as usual , is calculated according to the denominator statement and the categorical and restrict table-factors, regardless of the presence of the NOTHAVE factor.

THISROW?

The THISROW? feature is a way of doing a categorical statistical test for each row in the table, usually directly related to the N and % statistics on that row.
This is possible by (re-)defining a boolean (true/false) variable for each row (however, this boolean variable must be reconstructed for each row, making THISROW? an advanced computational feature).

Consider this example : the thing dataset for the entire table is PATINFO, which has one row per person (so a thing is a person) ; and the model expression chisq(thisrow?*trtgroup*N) is used.

For each row, determined via the categorical and restrict row factors, you have people who "belong" to this row, and people who do not. This sets up a boolean (true/false) variable named "THISROW?" ; since "THISROW?" has one true/false value for each person (each thing), it's like a new synthetic column that can be appended to the thing dataset.
This synthetic boolean variable is one of the input variables provided to the categorical statistical model (here, the chi-square test). Hence the categorical test is re-executed for each row.

Don't confuse this with THISROWCAT. Normally, within each table section, a statistical model (such as lm() or chisq()) is only executed once(aside from subpop statement categories). The THISROW? feature is very different. Within the table section, it's a different statistical model execution for each row categorical level – hence for each row. (For this purpose, column categorical factors are ignored.)

The THISROW? feature is not available for models that use a continuous input variable – since all the input variables are categorical, a smaller dataset of aggregate sub-counts can be put into the chi-square procedure, instead of providing a modified copy of the THING dataset for every single row (which would be CPU-inefficient).

When using THISROW?, it's a good idea to use a thing statement. However, the P-level and M-level datasets do not have to be two distinct datasets, they can be the same dataset (in contrast to the NOTHAVE feature).

TABLES' CATEGORICAL SPECTRUM

How big is the statistical table? What determines which categorical levels are included, and which are not? (I'll refer to this issue as the table's CATEGORICAL RANGE). For example, there could be a row for (gender=”female”,bodysys=”CNS”,prefterm=”Headache”) and there might not be a row for (gender=”male”,bodysys=”CNS”,prefterm=”Headache”). Similar considerations, of course, apply to columns. (Also note, because of the crossing of rows and columns, some table cells will have no actual data, due to sparse data.)

Usually, the following default method works fine: the range of categorical levels (such as (trtgroup=”Placebo” & bodysys=”CNS” & prefterm=”Headache”)) is big enough to accommodate the incoming data and the resulting statistics.
While that's usually enough, the advanced features NOTHAVE and THISROW? are exceptions. When you run across such exceptions, there are two options:

First option: choose an alternative, somewhat more complicated, default method. For the NOTHAVE feature, the alternative setting is the cartesian product of the M-level categorical levels and the P-level categorical levels.

The second option is to allow for a more explicit syntax to specify which categorical level combinations are included. This syntax could be typed as a one-line footnote statement, or, to target a subset of the table, could be typed as a table factor.
An example : CAT-RANGE(bodysys prefterm ; ae_dset) ** CAT-RANGE(gender trtgroup ; patinfo)
(Derive a categorical range from one dataset, then a range from another dataset, then “**” denotes the cartesian product of the two).
This would require more work from the end user, but would provide more flexibility.

Also note, when using a categorical table factor with a known(and small) range of levels in the data, if you want all the levels to print out (even if the data is sparse), type in the whole range, like this:
col ...*trtgroup(= “Placebo” “Roo 20mg” “Roo 40mg”)*..   ;

DETERMINATION OF THE DATA SOURCE: IMPLICIT OR EXPLICIT

For each section of the table, there is a corresponding list of of table factors, [T1,T5,T6] , which is interpreted to figure out the needed computational process. Part of this is the determination of the data source – ie where is the data for the computational process coming from. Most of the time, the identifiers in the table factors – variable names, mostly, and also dataset names – are interpreted to render a non-ambiguous determination of the data source.

For smaller programs, with few(or none) data transformation paragraphs preceding the table paragraph, it's unusual that more explicit sourcing syntax would be required (although explicit syntax would still be an option). The same is the case if there are several data transformation paragraphs preceding the table paragraph and a dataset-usage statement (“using-datasets patinfo advevents9 ; “) is added to the table paragraph (so the datasets needed only by the data preparation paragraphs are effectively ignored by the table paragraph). So, while more explicit sourcing syntax is seldom needed, there is a choice : implicit data source, via the list of table factors for this section of the table, or explicit data source syntax (as a footnote, or a table factor).

The data source might not be an exact copy of one of the input datasets; it could be a limited transformation of the available datasets : first, the data source could be a pair of datasets (possible with N or %, and thisrow?, see the “N %” section for more); second, the dataset is allowed to be augmented by a “PULL-IN adjustment” : if DATASET3 has variable v8, and DATASET4 has one row for each value of v8, then variable names in DATASET4(but not DATASET3) can become available to an augmented version of DATASET3 (note the number of rows in DATASET3 cannot increase in the adjustment; ie – rows in DATASET4 that fail to match to an existing row in DATASET3 would not be retained).

TREES (PART II)

Here is the second part of a discussion on how tree data structures are integral to this programming language. (The first part was to explain just enough to move on to showing how to use this software, here I provide a few, more subtle, clarifications). Recall, in the earlier section, I showed how the column tree and row tree relate to partitions of the statistical table I called sections, and each section is identified with a list of table factors.

Each section can be expected to include how many printed statistics? Well, suppose this section has two categorical table factors, one categorical factor has 3 categorical levels, the other categorical factor has 4 categorical levels. Then this section has up to 3*4=12 printed statistics (depending on the source data, less than 12 is possible).

In the following few pages, you'll see another visual example of column trees, row trees, and table sections. Included is an illustration of how the large-row-tree is reformatted to more easily fit in the left side of the page. A side effect of this is that some rows on the page will not correspond to a leaf node of the row tree, but will correspond to intermediate row tree nodes. Usually, these rows will not have printed statistics.

As I said earlier, each table section may have it's own computational process and data source, but two table sections are not required to have separate computational processes (an obvious example, F-statistic and it's p-value (fvalue and pvalue) cannot be in the same table section: the two sections will differ by one table factor (fvalue vs pvalue), but the conceptual compiler will recognize it as the same computational process (and not waste CPU cycles)).

```
col trt*gender*(n % sum)
row all rating*( [age<50] all )

will become the column-tree (small-version and big-version):

   Trt
_________
  Gender
_________
N  %  sum


                    Trt
___________________________________________
        Placebo               Roopla
_____________________  ____________________
        Gender                Gender
_____________________  ____________________
  Female       Male      Female      Male
_________   _________  _________  _________
N  %  sum   N  %  sum  N  %  sum  N  %  sum


and the row-tree (small-version and big-version):


 all          rating
_____   __________________
         [age<50]    all
        ___________ ______



 all                            rating
_____   ______________________________________________________
              Low               Medium          High
        _________________ _________________ __________________
         [age<50]    all   [age<50]    all   [age<50]    all
        ___________ ______ ___________ ______ ___________ ______
```

But if the big-row-tree is reformatted to fit better onto the page on which the statistical table is printed:

```
All
rating
  Low
    [age<50]
    all
  Medium
    [age<50]
    all
  High
    [age<50]
    all
```

Now, when the row tree is reformatted in such a manner, note that not all of the printed rows correspond to a leaf node of the row tree.

Looking at the column tree, one sees that the table has 3 column sections:
A: trt*gender*N
B: trt*gender*%
C: trt*gender*sum

Looking at the row tree, the table divides into 3 row sections:
R: all
S: rating*[age<50]
T: rating*all

I give these column and row sections labels (A,B,C & R,S,T) for illustrative purposes(next page).

	Trt											
	Placebo						Roopla					
	Gender						Gender					
	Female			Male			Female			Male		
	N	%	sum	N	%	sum	N	%	sum	N	%	sum
All	AR	BR	CR	AR	BR	CR	AR	BR	CR	AR	BR	CR
rating												
Low												
[age<50]	AS	BS	CS	AS	BS	CS	AS	BS	CS	AS	BS	CS
all	AT	BT	CT	AT	BT	CT	AT	BT	CT	AT	BT	CT
Medium												
[age<50]	AS	BS	CS	AS	BS	CS	AS	BS	CS	AS	BS	CS
all	AT	BT	CT	AT	BT	CT	AT	BT	CT	AT	BT	CT
High												
[age<50]	AS	BS	CS	AS	BS	CS	AS	BS	CS	AS	BS	CS
all	AT	BT	CT	AT	BT	CT	AT	BT	CT	AT	BT	CT

So, if the table has 3 column-sections and 3 row-sections then the table has 3*3=9 sections, which I've labeled as :
sections AR, BR, CR, AS, BS, CS, AT, BT, and CT in the area where the statistics are printed, so that you can see where the 9 sections are located.

CONFIGURATION OF THE CURRENT VERSION OF THE PRODUCT

This section explains the current configuration options for the Vilno Table software product. It is beta status (not yet version 1.0). This version is developed and tested on Linux, and will later be available on Linux, Windows, and Apple.

Vilno Table does not do (directly) row-by-row data processing, it delegates this to other software components. Hence, there are two configuration options: choice of data transformation engine and choice of statistical analysis software. The current version gives data processing work to the Vilno Data Transformation Engine product (a data transformation product, older than Vilno Table, that has functionality similar to SQL SELECT and the SAS datastep) and gives statistical analysis work to R (an open source implementation of the S programming language).

For future versions of Vilno Table, possible choices of data transformation back engine include
1: Vilno Data Transformation Engine
2: SAS/BASE (SAS datastep, proc transpose, proc means, proc sql)
3: others?
Possible choices of statistical analysis back engine may include :
1: S (either R or S-Plus)
2: SAS/STAT
3: SPSS (if there is a demand for it)
4: others?
(It may be possible , with a configuration with access to more than one statistical software, to produce tables that have p-values from two different statistical products on the same page.)

Why not use SQL (alone) as a data transformation back engine? SQL has some data transformation functions (such as many-to-many joins) but lacks certain procedural features that are sometimes used in complex data transformations.

Vilno Data Transformation (VDT) is developed in C++.
Vilno Table is developed in Python and C++.

When Vilno Table gives work to do to Vilno Data Transformation Engine or R, it hands over generated code (here, respectively, either Vilno Data Transformation syntax or the syntax of the S programming language) for the back engine components to execute. Obviously, these generated code files are not presented to the end-user (having to review them would usually be a waste of time, since the purpose of Vilno Table is to boost worker productivity).
In the current configuration, Vilno Data Transformation has another role, if the end-user wishes to do data preparation prior to statistical table work, one option is to write Vilno Data Transformation syntax paragraphs to do this data preparation work.

The PARITY paragraph is a type of syntax for data transformation that works at a higher level of abstraction than more old-fashioned syntaxes (VDT, SAS, SQL).
The Parity paragraph is designed as part of the Vilno Table programming language, but the feature is not yet up and running in the current version of the software product, which is why none of the examples in the appendix uses the parity paragraph.

DATA TRANSFORMATION: VDT AND THE PARITY PARAGRAPH

This chapter is about data transformation features provided with Vilno Table. The primary function of Vilno Table is to produce complex statistical tables, hence Vilno Table is not a data transformation product. However, Vilno Table depends upon data transformation components that play a key supporting role. Any statistical software product that is not provided with a supporting data preparation facility will be inflexible and difficult to work with.

Some readers may ask : by data transformation you mean what exactly? Data transformation is an activity that is extremely diverse, multi-stage, and difficult to categorize, no one can agree on what to call it : data preparation, data cleaning, data munging, data integration, getting the data ready for analysis, data scrubbing, ETL, and so on. By data transformation, I generally mean an activity after data collection and before reporting and analysis. Data correction, which is data cleaning DURING the data collection period (and corrections are made to the original data repository, not just a secondary set of derived datasets) is very different.

While this chapter will focus on what's available, data transformation is a subject that requires more research and development, the Data Preparation White Paper, in the pipeline, will address these issues.

The Vilno Table programming language product depends on two data transformation features: Vilno Data Transformation (VDT) and the PARITY paragraph.
VDT is a separate product, whose syntax is at a lower level of abstraction (similar to the SAS datastep and SQL SELECT). Vilno Table depends upon VDT in the same way that a Fortran compiler depends upon an assembler language compiler (many compilers for Fortran, C, etc will read the end-users' file (in Fortran), produce an assembler language file, and have the assembler language compiler convert the assembler language file to machine code).
Moreover, if the end-user wishes to manually do some data preparation, he can type paragraphs of code using the syntax of the VDT programming language, in the same file where he puts Vilno Table code. Vilno Table will takes these VDT-syntax paragraphs and hand them over to VDT to execute. (Hence VDT is playing a dual role here.)

The PARITY paragraph, like the table paragraph, is part of the Vilno Table programming language (however, in the current beta version of the product, the parity paragraph is not yet operational, that is why none of the test cases in the appendix use it). Anything you can do with a PARITY paragraph, you can also do with VDT code, but there are some types of data transformation that the PARITY syntax can do with many fewer lines of code than VDT syntax, because the PARITY paragraph uses a higher level of abstraction.

++=======================++

So how about a compare and contrast of VDT and PARITY? (Note that later in this chapter we'll use a data preparation example to show a side-by-side comparison of SQL, SAS, VDT, and Parity).
Comparing Parity to VDT is a bit like comparing Python to C : The Parity and Table paragraphs use a higher level of abstraction than VDT.
VDT syntax is more old-fashioned – someone used to SQL and SAS would pick it up right off the bat.

The Parity paragraph is especially useful for some types of data transformation (of a quasi-geometrical reshaping nature) that are often useful just prior to sketching a statistical table. For this reason, the Parity paragraph is a good complement to the Table paragraph .

There are other types of data transformation, such as early-stage-nitty-gritty data cleaning, where the Parity paragraph might not offer a huge advantage over a VDT paragraph. (and the more old-fashioned coding techniques in the VDT programming language will always have a role in day-to-day data cleaning).

Let's put it this way, sometimes when preparing the datasets for analysis, one can divide the data transformation work into two stages:
Early-stage data transformation is nitty-gritty data cleaning, and deciding what adjustments to make regarding things that went wrong with the data.
Late-stage data transformation is a generalized transposing and reshaping (here the Parity paragraph shines).
(However, it is not always clear how to separate these two stages: in some work situations the two stages are easy to divide, when that's not the case the data cleaning is more challenging).

I also want to make the distinction between THIS-GENERATION DATA TRANSFORMATION and NEXT-GENERATION DATA TRANSFORMATION. This-generation data transformation uses coding methods that programmers have been using for decades : techniques used in SQL SELECT, SPSS, SAS, VDT. It often involves old-fashioned logical constructs that have been in use since at least the 1970s : assignment (x=y), if-else syntax, for and while loops ; and also uses the context of input and output tables organized as rows and columns.

Although VDT is at a slightly higher abstraction level than the SAS datastep, it's still considered this generation data transformation.

Next generation data transformation is data transformation using new programming paradigms with the purpose of improving usability(learning curve and readability) or worker productivity, or both. Much next generation data transformation relies upon research and development that has not been done yet, and the Data Preparation White Paper will say more about that. The Parity paragraph is a next-generation data transformation tool that is designed as part of the Vilno Table programming language and is explained in more detail in this chapter.

TREATMENT-EMERGENT : AN ILLUSTRATIVE DATA TRANSFORMATION EXAMPLE

As I've noted, any statistical software needs access to a data preparation component, otherwise the end-user gets stuck. To illustrate the different methods available for data transformation, especially the methods provided with Vilno Table (Vilno Data Transformation and the PARITY paragraph), we will focus here on a specific data transformation example: the derivation of the treatment emergent adverse event dataset from the original adverse event dataset. We will focus on how this work task is completed with the following options: SQL, SAS, Vilno Data Transformation, and the Parity paragraph. That way, you can compare these options side by side.

Note: you will soon see that the solution to this work task using the Parity syntax requires only four lines of code, far more concise and succinct than the other options. That just shows, once again, that deep and fundamental research can pay off.

++====================++

To illustrate the different options for data transformation, let's use the following example: deriving the treatment-emergent AE dataset from the original AE dataset. The incoming AE dataset has variables: PATID , VISIT, BODYSYS, PREFTERM, SEVERITY.
PATID is patient number. VISIT is visit number, in the range: 1,2,3,4,5 ; where 1,2 are baseline visits and 3,4,5 are post-baseline visits.
BODYSYS and PREFTERM are categorical variables that represent disease categories (such as (bodysys,prefterm) = (“cardiovascular”,”stroke”) or (bodysys,prefterm)=(“gastrointestinal”,”ulcer”)).
(BODYSYS and PREFTERM are either string or integer, if integer then with an associated format, however the format is for printing purposes, and is not used in the data transformation calculation.)

SEVERITY is an integer variable, equal to 1, 2, 3 , which respectively mean “Mild”, “Moderate”, “Severe” (degree of severity) ; so it's treated as an ordered categorical variable.

From AE, we will derive AE2, the treatment-emergent adverse event dataset. For each level of bodysys and prefterm, there is a row in AE2 for each person such that either :
1: That adverse event occurred in the post-baseline period but not the baseline period
or
2: Even though the event did occur during the baseline period, it occurred post-baseline at a higher severity than observed during baseline.

This implies that for each level of (patid,bodysys,prefterm), there is at most one row in AE2. And adverse event records occurring at baseline and not getting worse after that are omitted altogether.

AE2 will have variables PATID, BODYSYS, PREFTERM, SEVERE_B, SEVERE_P , where SEVERE_B and SEVERE_P are the maximum severity codes baseline and post-baseline respectively.

To keep it simple, assume severity is never missing in AE.

Of course, this is just one small example, and data preparation is an extremely diverse topic. Nonetheless, this example is useful for showing how different data transformation programming tools use different techniques.

Since SQL is, by far, the most well known syntax, let's start with that. Just to make comparison easier, I avoid sub-queries ; instead the result of each SELECT statement is placed into a named dataset : the CREATE VIEW feature will suffice.

```
create view  data1 (patid,bodysys,prefterm,severe_b)  as
 select patid, bodysys, prefterm, max(severity)
  from ae
  group by patid, bodysys, prefterm
  where visit<=2 ;

create view  data2 (patid,bodysys,prefterm,severe_p)  as
 select patid, bodysys, prefterm, max(severity)
  from ae
  group by patid, bodysys, prefterm
  where visit>2 ;

create view ae2 (patid,bodysys,prefterm,severe_b,severe_p) as
  select d1.patid, d1.bodysys, d1.prefterm, d1.severe_b, d2.severe_p
  from data2 d2   left outer join   data1 d1
    on d1.patid=d2.patid  and d1.bodysys=d2.bodysys  and d1.prefterm=d2.prefterm
  where d2.severe_p>d1.severe_b  or  d1.severe_b is null  ;
```

++================================++

We'll see how to do this with the SAS programming language, since some readers may be familiar with that:

```
data base1 ;
  set ae ;
  if visit<=2 ;

proc sort data=base1 ;
  by patid bodysys prefterm severity ;

data base2 ;
  set base1 ;
  by patid bodysys prefterm severity ;
  if last.prefterm ;
  rename severity=severe_b ;

data post1 ;
  set ae ;
  if visit>2 ;
```

```
proc sort data=post1 ;
  by patid bodysys prefterm severity ;

data post2 ;
  set post1 ;
  by patid bodysys prefterm severity ;
  if last.prefterm ;
  rename severity=severe_p ;

data ae2 ;
  merge base2 post2 ;
  by patid bodysys prefterm ;
  if severe_p>severe_b  or severe_b=.  ;
```

++==============================++

Next, we use VDT (Vilno Data Transformation). I'm actually showing you two different ways to do this work task with VDT, which I'll refer to as the VDT-A approach and the VDT-B approach. Well, why?
Well, with SAS (and to some extent, SQL), when you need an additional data transformation, you probably have to write an additional paragraph of code. A "proc means"(one type of paragraph in SAS) is one data transformation. A SAS datastep is a single data transformation (plus the merge process). That's not always a bad thing. But if you have a sequence of related data transformations and each data transformation is quite brief (reducible to one line of code), this requirement forces you to write a fair amount of boilerplate.

The VDT-A approach uses the syntax of VDT, but also uses only one data transformation per VDT code paragraph (aside from the merge of input data). The reason: the VDT-A code example is quite easily compared side-by-side with SAS or SQL : one paragraph of VDT can be contrasted with a PROC MEANS (SAS) paragraph, the next paragraph can be analogous to a SAS datastep, and so on.

Hence, users of SQL or SAS may find the VDT-A code example a good way to put a toe into the water, prior to reading the VDT-B code example.

The VDT-B approach uses the syntax of VDT, without this self-imposed restriction. A VDT paragraph can contain multiple data transformations, if the programmer wishes. The VDT-B code example is shorter than previous examples (although not as short as the PARITY paragraph).

Here is the VDT-A code example :

```
inlist a/ae ;
if (not (visit<=2))  deleterow ;
sendoff(w/data2a) patid bodysys prefterm severity ;

inlist w/data2a ;
select severe_b = max(severity) by patid bodysys prefterm ;
sendoff(w/data2) patid bodysys prefterm severe_b ;
```

```
inlist a/ae ;
if (not (visit>2))  deleterow ;
sendoff(w/data3a) patid bodysys prefterm severity ;

inlist w/data3a ;
select severe_p = max(severity) by patid bodysys prefterm ;
sendoff(w/data3) patid bodysys prefterm severe_p ;

inlist w/data2 w/data3 ;
mergeby patid bodysys prefterm ;
if (not (severe_p>severe_b or severe_b is null))  deleterow ;
sendoff(w/ae2) patid bodysys prefterm severe_b severe_p ;
```

Here is the VDT-B code example, just 12 lines of code :

```
inlist a/ae ;
if (not (visit<=2))  deleterow ;
select severe_b = max(severity) by patid bodysys prefterm ;
sendoff(w/data2) patid bodysys prefterm severe_b ;

inlist a/ae ;
if (not (visit>2))  deleterow ;
select severe_p = max(severity) by patid bodysys prefterm ;
sendoff(w/data3) patid bodysys prefterm severe_p ;

inlist w/data2 w/data3 ;
mergeby patid bodysys prefterm ;
if (not (severe_p>severe_b or severe_b is null))  deleterow ;
sendoff(w/ae2) patid bodysys prefterm severe_b severe_p ;
```

++==========================++

And here's how to do it using a PARITY paragraph:

```
each patid bodysys prefterm :
  severe_b  =  [visit<=2][max(severity)]  ;
  severe_p  =  [visit > 2][max(severity)] ;
  if severe_p>severe_b  or severe_b=.  ;
```

You are probably thinking, you forgot to type in a few lines there. Nope, it just requires FOUR lines of code. Saving the statistical programmer mountains of work-hours and pain, that's what the Vilno initiative is all about.

THE PARITY PARAGRAPH

The PARITY paragraph involves a different way of looking at the data. Every dataset has a parity , (and we'll see later every variable, as a consequence has a parity). The parity of the PATINFO dataset is (pat_id) – meaning pat_id is a column in PATINFO, and there's only one row per value of pat_id. The parity of VITAL_SIGNS may well be (pat_id,visit_num), implying just one row per value of (pat_id,visit_num). AE(short for ADVERSE_EVENTS) is a dataset with columns pat_id and visit_num. But it won't have parity (pat_id,visit_num) , because each patient-visit could have many rows in the dataset.

So, if the other datasets have a clearly defined parity, then the AE dataset has a parity that is simply VOID (or “undefined”). As it turns out, there's more to it than that: yes, AE has a void parity, but, if we are in a situation where pat_id and visit_num are being used as parity columns, and AE does have these columns, then AE has parity (pat_id,visit_num)++ , which means AE has those columns, and moreover, has (possibly) multiple records per value of (pat_id,visit_num).

But, then, in another context, AE could have parity (pat_id,bodysys,prefterm)++ . It may seem fuzzy, because the parity of AE changes depending on the computation you're in the middle of, but it's still very useful.

A parity paragraph is a piece of code where you choose a parity (say (patid) , or say (patid,visitnum), or say (patid,bodysystem,prefferedterm)), and you write a few lines of code that execute in the context of that parity.
Here's a very simple example :

```
each pat_id bodysys prefterm  :
  baseline_sev  =  [visit<=2][max(severity)]  ;
  postbase_sev  =  [visit>2][max(severity)]  ;
```

First note : each computation is done within each level of the chosen PARITY. (Hence information about patient #8 is not mixed up with information about patient #9, and if patient #8 has “headache” records and “ulcer” records, a computation on “headache” records is not affected by the presence of “ulcer” records).

Also, see that the calculation of the following form:

```
   Y = [bla-bla][bla-bla][blabla] ;
```

I call a chain calculation, or a “zoom-in” calculation : you start with a (fairly large) amount of data (subdivided by the parity of the current paragraph , ie an amount of data for (pat_id=8, bodysys=”CNS”, prefterm=”Headache”)). Then, through a sequence of steps, transform that data into a smaller amount, then a smaller amount, then a single value. (This sequence of data transforms is done within each parity level, data from one parity level is not mixed with another parity level).

So, (for a given choice of (pat_id,bodysys,prefterm)), first subset to the rows such that visit<=2 (the baseline rows) , then take the maximum of the severity code variable.
So BASELINE_SEV is a new variable that receives a computed statistic, and there is one such computed statistic for each level of (pat_id,bodysys,prefterm).
Note that BASELINE_SEV, and the newly created dataset where it's located, has parity (pat_id,

bodysys, prefterm) .

A parity paragraph begins with the top-level statement, which simply states what parity the paragraph is operating at (and, optionally might name input datasets to pool in, and give the output dataset a name). Underneath this first line, the body consists of multiple statements, where each statement can take the following forms:

1: A typical chain calculation :
Y = [bla-bla][bla-bla][bla-bla]
where you take a mass of data, and shrink it and aggregate it, and assign the resulting value to Y.

2: Procedural statement, using syntax familiar to all: assignment statements, if-else syntax, for/while syntax

3: Calculate a new boolean variable, using a boolean expression that might include a few very short boolean chain calculations, here are two examples :

enough_data = any[visit<2 and score!=.] and any[visit>=2 and score!=.]

postbaseline_only = notany[visit<2 and score!=.] and any[visit>=2 and score!=.]

4: More complex examples are possible, chain expressions (both string/numeric and boolean) can be inside arithmetic(or boolean) expressions, which can be used in procedural statements (assignment, if/else, etc.). (But in many examples, such complexity is not needed : a few statements, consisting of short chain calculations, suffice.)

If the parity paragraph is at (patient,visit) parity, then new variables you write computational results to will be at (patient,visit) parity as well. Variables that are read from do not have to be at (patient,visit) parity.

Each step in the chain of a chain calculation transforms the data, subsetting or shrinking, or aggregating the data (MAX, AVG, etc.). The steps in the chain are executed left to right (and ANY and NOTANY executed last). The variable names in the chain are used to identify the data source (like the table paragraph), unless explicit syntax is used to specify the source.
Here are the sorts of links that can be used in the chain of calculations:
[dataset_name] explicitly specify the data source
[x<y] boolean expression, to take a subset of observations
[v] choose this variable (or column)
[MAX(v)] choose this summary statistic, where v is a variable name(or column)
[unique_check v] choose this variable , you are expecting a single value, at this stage within the chain, for each parity level, but if that's not the case, raise a warning message.

[v!=.] pretty common example, include only observations where v has a non-missing value
[high visit] determine the maximum value of the variable “visit”, and include observations for which visit is this maximum (do not confuse with [max(visit)]).

[v!=.][high visit]   (and not [high visit][v!=.])
is a good example of where the order of steps in the chain matters : you want to use the most recent visit that has at least some non-missing data, if you do [high visit] first, you could get a visit with no data at all (and the second-most recent visit may have data that you want access to).

```
[visit = 3 4 5]   just means  [visit=3  or  visit=4  or  visit=5]
Y1 , Y2  =  [bla-bla][bla-bla][ min(v) , max(v) ]
```

++===============++

% CHANGE BMD example

Let's do another example with PARITY syntax : bmd is a continuous variable (associated with skeletal health) that's measured each visit, and also each region (bone region = “Hip”, “Spine”, etc.). Visit numbers have values 1, 2, 3, 4, 5. The baseline period visits are 1 and 2.
For each bone region, there is a single bmd measurement for each visit (unless it's a missing value).

Let's calculate summary statistics for % change from baseline BMD, calculated two ways : visit 5 or LOCF (last observation carried forward).
We can't calculate % change BMD within a TABLE paragraph, so we'll use data transformation code to do this. We could use VDT code to do this, but PARITY syntax will result in much less code.

```
each site patid :
  at_risk  =  age>55  or  weight>170  ;


each site patid region :
  bmd_base  =  [visit = 1 2][bmd!=.][high visit][bmd]  ;
  bmd_v5     =  [visit = 5][bmd]  ;
  bmd_locf   =  [visit = 3 4 5][bmd!=.][high visit][bmd]  ;
  [post_way postbmd]  =  [bmd_v5 bmd_locf]  ;


each site patid region post_way :
  pctchg  =  ((postbmd – bmd_base)/bmd_base)*100  ;
```

You've already seen chain statements. But this is a transposing data description line :

```
  [post_way postbmd]  =  [bmd_v5 bmd_locf]  ;
```

It is a statement to transpose downwards the continuous data bmd_v5 and bmd_locf : the continuous data from bmd_v5 and bmd_locf is poured into the column POSTBMD, which is paired with the categorical column POST_WAY (which has two categorical levels: “bmd_v5” and “bmd_locf”)

With the data preparation done, summary statistics for % change BMD can be done with a table paragraph :

```
onlyif[ at_risk=true ] ;
col  n  mean(pctchg)  std(pctchg) ;
row  post_way*region  ;
```

Note that POST_WAY is a synthetic categorical variable you've just set up, with two categorical levels (for visit=5 and for LOCF).

Instead of using parity coding, you could do this in SAS – but it would require a substantial amount of boilerplate code. You could also do this in VDT – again it would require a fair amount of boilerplate (although less than SAS).

PARITY : ABSTRACT IMPLICATIONS

What are the theoretical implications and questions arising from the parity paragraph?

The incoming data is organized as a set of tables (each table having rows and columns), however the parity paragraph does involve a different way of conceptualizing the data, almost as if it superimposes one data model on top of the older standard data model. Unfortunately, this superimposition is not theoretically complete.

This situation points to a direction for research and development in data models.

Here, a paradox: these theoretical questions can be difficult and time consuming, however on a case-by-case basis, with the examples that occur in practice, the PARITY paragraph works quite well and without ambiguity.

That said, these clues hinting at the need for further theoretical research should not be surprising: the parity paragraph is not for table production, but for data transformation, and data transformation is a subject that still requires a large amount of applied and theoretical research and development.

One question: when and how can you superimpose one data model upon another data model in a clean and complete manner?

The parity paragraph utilizes multi-dimensional structures superimposed on the data source, however these multi-dimensional structures can shift and replace themselves like a kaleidoscope (as a result they don't always represent permanent storage choices).

The data source is typically a set of datasets, each dataset is a two-dimensional structure (rows and columns). However, the levels in a categorical column can represent an additional dimension. When you are using categorical columns in a certain fashion, you are effectively exploiting “hidden” dimensions.

That's where the concept of the PARITY paragraph comes from (it also influences the TABLE paragraph).

Hence, the Vilno Table programming language is all about exploiting multiple dimensions in just the right way to give a turbo boost to worker productivity (and by extension, usability).

VILNO DATA TRANSFORMATION (VDT)
VDT is a data-processing workhorse that other software applications can depend on. It is a separate product, with documentation available separately. The VDT programming language is in the same category as SQL, SPSS, and SAS.

The central feature of VDT is a paragraph of code that reads in one or more input datasets, reads, modifies, processes data, writes resulting derived data to one or more output datasets. (And each dataset is a two-dimensional structure: a table with rows and columns).
Which means, it's a paragraph of code whose functionality and computing methods are very similar to a paragraph of SQL SELECT or a SAS datastep.

++===========++

If there is more than one input dataset, there is some sort of merge process just prior to the principal data transformation. Again, this merge process is quite familiar to someone experienced in SQL or SAS , you can merge in an “interleaving” fashion or a “join-product” fashion ; with the “interleaving” method , rows from one input dataset go UNDERNEATH rows from another input dataset ; with the “join-product” method rows from one input dataset are MATCHED TO rows from another input dataset.

The “interleaving” mode is the same as the SQL UNION method (and the SET option in the SAS datastep). The “join-product” mode is essentially a CARTESIAN PRODUCT , same as the very well known SQL JOIN feature. (When doing the “join-product” technique, you sometimes distinguish between a many-to-many merge and a many-to-one merge).

Both these modes provide the option of being executed within each strata-level, where the choice of strata is specified by a short list of group-by variables (eg, merge in by “country” “state”). (Again, just like in SQL or SAS).

After the input data is read in (or merged in), the main data transformation is done, and the results are written to an output dataset. (In more complicated situations, after the data is merged, the programmer can choose to do a sequence of data transformations, and after any step in the sequence can choose to write the intermediate dataset to a different output dataset).

++============++

The most common type of data transformation is a unit of procedural code that is executed for each row in the incoming data table. “Procedural” here means old-fashioned computing techniques you learned in CompSci 101 : assignment statements, if-else statements, loops(“for” and “while”), (and also the option to delete a row or make a duplicate copy of a row).
Other data transformations include aggregate statistics (N(count), sum, average, etc.), and transposing (rows become columns and columns become rows).

For more detail, see the documentation of the syntax of the Vilno Data Transformation programming language that is provided with the VDT product, available to the public since 2007.

Vilno Table Programming Language White Paper
Appendix A
Case-By-Case Test Examples Produced Using The Vilno Table Product

Table A1: Example of 4 categorical variables: summary stats and chi-square

TOPTOP

	trt				chisqvalue	pvalue
	A		B			
	n	%	n	%		
TOPTOP						
zcode					8.333	0.003892
7	6	100				
8			6	100		
color					3	0.08326
blue	1	16.67	5	83.33		
red	5	83.33	1	16.67		
gender					0.3333	0.5637
female	2	33.33	4	66.67		
male	4	66.67	2	33.33		
agegrp					0.3333	0.5637
0	3	50	3	50		
1	3	50	3	50		

```
title "Table A1: Example of 4 categorical variables: summary stats and chi-square" ;

directoryref a="/home/robert/test" ;

inputdset asc a/patinfo1 patid trt zcode color gender agegrp 1*(patid) ;
inputdset asc a/ae1 patid bodysys prefterm ;
# leave out pat defn for this test program

printto "/home/robert/test/output3" ;
denom trt ;
model chisq(thisrowcat*trt*n)
col trt*(n %) chisqvalue pvalue ;
row zcode color gender agegrp ;
```

Table A2: Example of linear model stats in a table

TOPTOP

	Region												Model Coefficient			F-statistic	
	East			North			South			West			Estimate	T	P-value	F	P-value
	Pairwise			Pairwise			Pairwise			Pairwise							
	Estimate	T	P-value	Estimate	T	P-value	Estimate	T	P-value	Estimate	T	P-value					
TOPTOP																	
Region																15921	0
East				-4.432	-4.515	0.0002	0.506	0.318	0.7536	-1.000	-0.986	0.3346	108.0	54.76	0		
North	4.432	4.515	0.0002				4.938	3.177	0.0044	3.432	3.496	0.0020	112.4	56.84	0		
South	-0.506	-0.318	0.7536	-4.938	-3.177	0.0044				-1.506	-0.946	0.3544	107.5	34.72	0		
West	1.000	0.986	0.3346	-3.432	-3.496	0.0020	1.506	0.946	0.3544				109.0	55.27	0		

```
title "Table A2: Example of linear model stats in a table" ;

directoryref a="/home/robert/test" ;

inputdset asc a/data_lm2 region temp cscore ;

categorical region ;
continuous temp cscore ;

printto "/home/robert/test/out_lin02" ;


model lm(cscore = region-1+temp) ;

col region*all*( est_pw t_pw pval_pw ) all*(est t_c pval_c) all*(fvalue pvalue) ;
row region ;


label all "Pairwise" "Model Coefficient" "F-statistic"    fvalue "F" pvalue "P-value"   region "Region"
      est_pw "Estimate"  t_pw "T"  pval_pw "P-value"      est "Estimate" t_c "T" pval_c "P-value"  ;
```

Table A3: AE table, with chi-square (75-patient dbase)

TOPTOP

	trt						all		pvalue
	0		20		40		n	%	
	n	%	n	%	n	%			
TOPTOP									
all	25	100	25	100	25	100	75	100	NaN
have	14	56	19	76	6	24	39	52	0.001017
bodysys									
Cancer									
all	1	4	1	4			2	2.67	0.5983
nothave	24	96	24	96	25	100	73	97.33	0.5983
prefterm									
Lung Cancer			1	4			1	1.33	0.3629
Melanoma	1	4					1	1.33	0.3629
Cardiovascular									
all	1	4	1	4	5	20	7	9.33	0.0804
nothave	24	96	24	96	20	80	68	90.67	0.0804
prefterm									
Blood Clot					1	4	1	1.33	0.3629
MI	1	4			4	16	5	6.67	0.0617
Stroke	1	4	1	4	1	4	3	4	1
Gastrointestinal									
all	11	44	6	24			17	22.67	0.0010
nothave	14	56	19	76	25	100	58	77.33	0.0010
prefterm									
Bellyache	2	8					2	2.67	0.1281
Cramps	3	12	3	12			6	8	0.1958
Flatulence	10	40	5	20			15	20	0.0019
Ulcer	2	8					2	2.67	0.1281
Neurological									
all	3	12	15	60	3	12	21	28	0.0001
nothave	22	88	10	40	22	88	54	72	0.0001
prefterm									
Agitation	2	8	2	8	2	8	6	8	1
Headache	3	12	14	56	2	8	19	25.33	0.0001
Skeletal									
all	7	28	4	16	2	8	13	17.33	0.1707
nothave	18	72	21	84	23	92	62	82.67	0.1707
prefterm									
Broken Leg			1	4	1	4	2	2.67	0.5983
Skull Fracture	1	4					1	1.33	0.3629
Vertebral Fracture	7	28	3	12	1	4	11	14.67	0.0506

```
title "Table A3: AE table, with chi-square (75-patient dbase)" ;

directoryref a="/home/robert/test" ;


inputdset asc a/patinfo3 patid trt 1*(patid) ;
inputdset asc a/advevt3a patid bodysys prefterm ;
thing pat uniqval(patid) a/patinfo3 ;
n~n(pat) ;

printto "/home/robert/test/outp01" ;
denom trt ;

model chisq(thisrow?*trt*n)

col (trt all)*(n %) pvalue ;
row all have(a/advevt3a) bodysys*(all nothave prefterm) ;
```

Table E1: N and % simple example

	TOPTOP			
	trt			
	A		B	
	n	%	n	%
TOPTOP				
bodysys				
CNS				
prefterm				
Headache	3	50		
Jitters	1	16.67		
Cardio				
prefterm				
MI	1	16.67	1	16.67
Stroke			1	16.67
Gastro				
prefterm				
Bellyache			1	16.67
Skeletal				
prefterm				
Broken_Foot			2	33.33
Fracture	1	16.67	1	16.67

```
title "Table E1:  N and % simple example" ;

directoryref a="/home/robert/test" ;

inputdset asc a/patinfo1 patid trt 1*(patid) ;
inputdset asc a/ae1 patid bodysys prefterm ;
thing pat uniqval(patid) a/patinfo1 ;
n~n(pat) ;

printto "/home/robert/test/output1" ;
denom trt ;
col trt*(n %) ;
row bodysys*prefterm ;
```

```
Table E2: AE table, summary stats
                               TOPTOP

                     ___________________________
                            trt          all
                     _________________ _________
                        A        B     n    %
                     ________ ________ __ ______
                     n   %    n   %
                     _ ______ _ ______
TOPTOP
  all                6 100    6 100    12 100
  have               4  66.67 4  66.67  8  66.67
  bodysys
    CNS
      all            4  66.67           4  33.33
      nothave        2  33.3  6 100     8  66.67
      prefterm
        Headache     3  50              3  25
        Jitters      1  16.67           1   8.33
    Cardio
      all            1  16.67 1  16.67  2  16.67
      nothave        5  83.3  5  83.3  10  83.33
      prefterm
        MI           1  16.67 1  16.67  2  16.67
        Stroke                1  16.67  1   8.33
    Gastro
      all                     1  16.67  1   8.33
      nothave        6 100    5  83.3  11  91.67
      prefterm
        Bellyache             1  16.67  1   8.33
    Skeletal
      all            1  16.67 2  33.33  3  25
      nothave        5  83.3  4  66.7   9  75
      prefterm
        Broken_Foot           2  33.33  2  16.67
        Fracture     1  16.67 1  16.67  2  16.67
```

```
title "Table E2: AE table, summary stats" ;

directoryref a="/home/robert/test" ;

inputdset asc a/patinfo1 patid trt 1*(patid) ;
inputdset asc a/ae1 patid bodysys prefterm ;
thing pat uniqval(patid) a/patinfo1 ;
n~n(pat) ;

printto "/home/robert/test/output2" ;
denom trt ;
col (trt all)*(n %) ;
row all have(a/ae1) bodysys*(all nothave prefterm) ;
```

Table E3: AE table, summary stats, TRT*GENDER, denom=trt

TOPTOP

	trt								all	
	A				B				n	%
	gender				gender					
	female		male		female		male			
	n	%	n	%	n	%	n	%		
TOPTOP										
all	2	33.33	4	66.67	4	66.67	2	33.33	12	100
have			4	66.67	3	50	1	16.67	8	66.67
bodysys										
CNS										
all			4	66.67					4	33.33
nothave	2	33.33	0	0	4	66.67	2	33.33	8	66.67
prefterm										
Headache			3	50					3	25
Jitters			1	16.67					1	8.33
Cardio										
all			1	16.67	1	16.67			2	16.67
nothave	2	33.33	3	50	3	50	2	33.33	10	83.33
prefterm										
MI			1	16.67	1	16.67			2	16.67
Stroke					1	16.67			1	8.33
Gastro										
all					1	16.67			1	8.33
nothave	2	33.33	4	66.67	3	50	2	33.33	11	91.67
prefterm										
Bellyache					1	16.67			1	8.33
Skeletal										
all			1	16.67	1	16.67	1	16.67	3	25
nothave	2	33.33	3	50	3	50	1	16.67	9	75
prefterm										
Broken_Foot					1	16.67	1	16.67	2	16.67
Fracture			1	16.67			1	16.67	2	16.67

```
title "Table E3: AE table, summary stats, TRT*GENDER, denom=trt" ;

directoryref a="/home/robert/test" ;

inputdset asc a/patinfo1 patid trt gender  1*(patid) ;
inputdset asc a/ae1 patid bodysys prefterm ;
thing pat uniqval(patid) a/patinfo1 ;
n~n(pat) ;

printto "/home/robert/test/output2b" ;
denom trt ;
col (trt*gender all)*(n %) ;
row all have(a/ae1) bodysys*(all nothave prefterm) ;
```

Table E4: AE table, summary stats, TRT*GENDER, denom=trt and gender
TOPTOP

	trt								all	
	A				B				n	%
	gender				gender					
	female		male		female		male			
	n	%	n	%	n	%	n	%		
TOPTOP										
all	2	100	4	100	4	100	2	100	12	100
have			4	100	3	75	1	50	8	66.67
bodysys										
CNS										
all			4	100					4	33.33
nothave	2	100	0	0	4	100	2	100	8	66.67
prefterm										
Headache			3	75					3	25
Jitters			1	25					1	8.33
Cardio										
all			1	25	1	25			2	16.67
nothave	2	100	3	75	3	75	2	100	10	83.33
prefterm										
MI			1	25	1	25			2	16.67
Stroke					1	25			1	8.33
Gastro										
all					1	25			1	8.33
nothave	2	100	4	100	3	75	2	100	11	91.67
prefterm										
Bellyache					1	25			1	8.33
Skeletal										
all			1	25	1	25	1	50	3	25
nothave	2	100	3	75	3	75	1	50	9	75
prefterm										
Broken_Foot					1	25	1	50	2	16.67
Fracture			1	25			1	50	2	16.67

```
title "Table E4: AE table, summary stats, TRT*GENDER, denom=trt and gender" ;

directoryref a="/home/robert/test" ;

inputdset asc a/patinfo1 patid trt gender  1*(patid) ;
inputdset asc a/ae1 patid bodysys prefterm ;
thing pat uniqval(patid) a/patinfo1 ;
n~n(pat) ;

printto "/home/robert/test/output2c" ;
denom trt gender ;
col (trt*gender all)*(n %) ;
row all have(a/ae1) bodysys*(all nothave prefterm) ;
```

Table E5: Mean, Median, Std examples, A/B/Y dataset

	TOPTOP															
	a															
	F				M				X				Y			
	n	mean	std	median	n	mean	std	median	n	mean	std	median	n	mean	std	median
TOPTOP																
b																
G	3	19	2.65	18	3	13	2.65	14					2	8.5	0.71	8.5
H	4	14.25	1.89	13.5	2	14.5	3.54	14.5	1	8		8				
I	12	19.83	11.13	17	3	16	3.61	15	1	8		8				

```
title "Table E5: Mean, Median, Std examples, A/B/Y dataset" ;

directoryref a="/home/robert/test/r_qa" ;


inputdset asc a/data_medi1 a b y   ;


printto "/home/robert/test/t1/outp21" ;

onlyif[y!=.] ;

col a*(n mean(y) std(y) median(y)) ;
row b ;

# onlyif statement to omit observations for which Y is missing
#  so that N is in sync with mean, std
#  ( similar effect can be done with [y!=.]*n )
```

Table G1: Summary stats, and boolean expressions

	TOPTOP							
	all		gender=='male'		gender=='female' or patid<10		pat has bodysys=='CNS'	
	n	%	n	%	n	%	n	%
TOPTOP								
all								
all	12	100	6	50	11	91.67	4	33.33
trt=='A' or bodysys=='Skeletal'	6	50	5	41.67	5	41.67	4	33.33
trt								
A								
all	6	100	4	66.67	6	100	4	66.67
trt=='A' or bodysys=='Skeletal'	4	66.67	4	66.67	4	66.67	4	66.67
B								
all	6	100	2	33.33	5	83.3		
trt=='A' or bodysys=='Skeletal'	2	33.33	1	16.67	1	16.67		

```
title "Table G1: Summary stats, and boolean expressions" ;

directoryref a="/home/robert/test" ;

inputdset asc a/patinfo1 patid trt gender 1*(patid) ;
inputdset asc a/ae1 patid bodysys prefterm ;
thing pat uniqval(patid) a/patinfo1 ;
n~n(pat) ;

printto "/home/robert/test/output6" ;
denom trt ;


col (all [gender=="male"] [gender=="female" or patid<10] [pat has bodysys=="CNS"] )*(n %) ;
row (all trt)*(all [trt=="A" or bodysys=="Skeletal"]) ;
```

Table G2: Summary stats, and boolean expressions

	TOPTOP					
	trt				all	
	A		B		n	%
	n	%	n	%		
TOPTOP						
all	6	100	6	100	12	100
gender						
female	2	33.33	4	66.67	6	50
male	4	66.67	2	33.33	6	50
gender=='female'	2	33.33	4	66.67	6	50
bodysys=='Skeletal'	1	16.67	2	33.33	3	25
pat has bodysys=='Skeletal'	1	16.67	2	33.33	3	25
pat nothas bodysys=='Skeletal'	5	83.33	4	66.67	9	75
gender=='female' or bodysys=='Skeletal'	1	16.67	4	66.67	5	41.67
gender=='female' or pat has bodysys=='Skeletal'	3	50	5	83.33	8	66.67
gender=='female' or pat nothas bodysys=='Skeletal'	5	83.33	5	83.33	10	83.33

```
title "Table G2: Summary stats, and boolean expressions" ;

directoryref a="/home/robert/test" ;

inputdset asc a/patinfo1 patid trt gender 1*(patid) ;
inputdset asc a/ae1 patid bodysys prefterm ;
thing pat uniqval(patid) a/patinfo1 ;
n~n(pat) ;

printto "/home/robert/test/output7" ;
denom trt ;


col (trt all)*(n %) ;
row  all gender  [gender=="female"]  [bodysys=="Skeletal"]
[pat has bodysys=="Skeletal"]
[pat nothas bodysys=="Skeletal"]
[gender=="female" or bodysys=="Skeletal"]
[gender=="female" or pat has bodysys=="Skeletal"]
[gender=="female" or pat nothas bodysys=="Skeletal"]
;
```

Table G3: AE, chi-square(not use format)

TOPTOP

	trt				all		pvalue
	A		B		n	%	
	n	%	n	%			
TOPTOP							
have	4	66.67	4	66.67	8	66.67	0.5403
bodysys							
CNS							
all	4	66.67			4	33.33	0.0662
nothave	2	33.3	6	100	8	66.67	0.0662
prefterm							
Headache	3	50			3	25	0.1824
Jitters	1	16.67			1	8.33	1
Cardio							
all	1	16.67	1	16.67	2	16.67	0.4386
nothave	5	83.3	5	83.3	10	83.33	0.4386
prefterm							
MI	1	16.67	1	16.67	2	16.67	0.4386
Stroke			1	16.67	1	8.33	1
Gastro							
all			1	16.67	1	8.33	1
nothave	6	100	5	83.3	11	91.67	1
prefterm							
Bellyache			1	16.67	1	8.33	1
Skeletal							
all	1	16.67	2	33.33	3	25	1
nothave	5	83.3	4	66.7	9	75	1
prefterm							
Broken_Foot			2	33.33	2	16.67	0.4386
Fracture	1	16.67	1	16.67	2	16.67	0.4386

```
title "Table G3: AE, chi-square(not use format)" ;

directoryref a="/home/robert/test" ;

inputdset asc a/patinfo1 patid trt 1*(patid) ;
inputdset asc a/ae1 patid bodysys prefterm ;
thing pat uniqval(patid) a/patinfo1 ;
n~n(pat) ;

printto "/home/robert/test/output4" ;
denom trt ;

model chisq(thisrow?*trt*n)

col (trt all)*(n %) pvalue ;
row have(a/ae1) bodysys*(all nothave prefterm) ;
```

Table G4: AE table, with chi-square (use a format)

TOPTOP

	trt				all		pvalue
	A		B		n	%	
	n	%	n	%			
TOPTOP							
have	4	66.67	4	66.67	8	66.67	0.5403
bodysys							
Skeletal							
all	1	16.67	2	33.33	3	25	1
nothave	5	83.3	4	66.7	9	75	1
prefterm							
Fracture	1	16.67	1	16.67	2	16.67	0.4386
Broken_Foot			2	33.33	2	16.67	0.4386
Gastro							
all			1	16.67	1	8.33	1
nothave	6	100	5	83.3	11	91.67	1
prefterm							
Bellyache			1	16.67	1	8.33	1
Cardio							
all	1	16.67	1	16.67	2	16.67	0.4386
nothave	5	83.3	5	83.3	10	83.33	0.4386
prefterm							
MI	1	16.67	1	16.67	2	16.67	0.4386
Stroke			1	16.67	1	8.33	1
CNS							
all	4	66.67			4	33.33	0.0662
nothave	2	33.3	6	100	8	66.67	0.0662
prefterm							
Headache	3	50			3	25	0.1824
Jitters	1	16.67			1	8.33	1

```
title "Table G4: AE table, with chi-square (use a format)" ;

directoryref a="/home/robert/test" ;
formatfile "/home/robert/test/formats.txt" ;

inputdset asc a/patinfo1 patid trt 1*(patid) ;
inputdset asc a/ae2 patid bodysys~(format=bodysysf) prefterm~(format=ptermf) ;
thing pat uniqval(patid) a/patinfo1 ;
n~n(pat) ;

printto "/home/robert/test/output5" ;
denom trt ;

model chisq(thisrow?*trt*n)

col (trt all)*(n %) pvalue ;
row have(a/ae2) bodysys*(all nothave prefterm) ;
```

Table G5: Subject Disposition

	TOPTOP								
	trt						all		pvalue
	0		20		40		n	%	
	n	%	n	%	n	%			
TOPTOP									
all	25	100	25	100	25	100	75	100	NaN
status									
Completed Study	17	68	6	24	25	100	48	64	0.00000
Discontinued	7	28	15	60			22	29.33	0.00002
Enrolled Phase IV	1	4	4	16			5	6.67	0.06169
reasdisc!=''									
reasdisc									NaN
Death	5	20	5	20			10	13.33	0.0559
Lost to Follow-up			7	28			7	9.33	0.0004
Patient Choice	2	8	2	8			4	5.33	0.3477
Physician Advice			1	4			1	1.33	0.3629

```
title "Table G5: Subject Disposition " ;

directoryref a="/home/robert/test" ;


inputdset asc a/patinfo3 patid trt status reasdisc   1*(patid) ;
inputdset asc a/advevt3a patid bodysys prefterm ;
thing pat uniqval(patid) a/patinfo3 ;
n~n(pat) ;

printto "/home/robert/test/t1/outp03b" ;
denom trt ;

model chisq(thisrow?*trt*n)

col (trt all)*(n %) pvalue ;
# row all status*(all reasdisc) ;
row all status [reasdisc!=""]*reasdisc ;
```

Table G6: Linear model, quine dataset (also printstat statement)

TOPTOP

	all			all	
	est	t_c	pval_c	fvalue	pvalue
TOPTOP					
eth					
N	-8.688	-3.412	0.0008432		
sex					0.0006790
M	1.900	0.7229	0.4710		
age				4.574	
F1	-2.676	-0.701	0.4844		
F2	6.236	1.628	0.1057		
F3	5.115	1.274	0.2049		

```
title "Table G6: Linear model, quine dataset (also printstat statement)" ;

directoryref a="/home/robert/test/ms" ;

inputdset asc a/quine eth sex age lrn days ;

categorical eth sex age lrn ;
continuous days ;

printto "/home/robert/test/ms/out_quin1" ;


model lm(days = eth+sex+age) ;

col all*(est t_c pval_c) all*(fvalue pvalue) ;
row eth sex age ;

# let us show the printstat statement, to avoid duplicate prints
printstat fvalue.c1 @ age ;
printstat pvalue @ sex.r1 ;

# .c1 ~ first occurence of "fvalue" in col statement
# but since "fvalue" and "pvalue" only typed in once in col/row code,
#  don't need to use .c1 or .r1
```

Table G7: Pairwise, selected comparisons only (not all of them)

TOPTOP

	t_pw	pval_pw	t_pw	pval_pw
TOPTOP				
all	0.3197	0.7506	5.435	0.000001806

```
# first test pgm for lm() with square dataset ( 4 x 4 anova )
#  (variable grp has 4 categorical levels, and variable trt has 4 levels)

title "Table G7: Pairwise, selected comparisons only (not all of them)" ;

directoryref a="/home/robert/test/r_qa" ;
inputdset asc a/square grp trt response v2 ;
categorical grp trt ;
continuous response v2 ;
printto "/home/robert/test/sq/out_sqpgm01" ;

model lm( response = grp + trt + grp*trt ) ;

col  t_pw(grp*trt,(6,"B"),(6,"D"))  pval_pw(grp*trt,(6,"B"),(6,"D"))
     t_pw(grp*trt,(8,"C"),(8,"D"))  pval_pw(grp*trt,(8,"C"),(8,"D"))   ;
row all ;



# type in desired contrasts, in statistic argument expression
#  otherwise, all pairwise contrasts = really big table

# Compare the (6,"B") group to the (6,"D") group, for example.
# If you wanted all pairwise comparisons, then do not type in desired contrasts,
# instead do :
# col grp*trt*grp*trt*pval_pw ;
```

Table G8: AE table, with chi-square (75-patient), & trtemgt derive

TOPTOP

	trt						all		pvalue
	0		20		40		n	%	
	n	%	n	%	n	%			
TOPTOP									
all	25	100	25	100	25	100	75	100	NaN
have	14	56	19	76	6	24	39	52	0.001017
bodysys									
Cancer									
all	1	4	1	4			2	2.67	0.5983
nothave	24	96	24	96	25	100	73	97.33	0.5983
prefterm									
Lung Cancer			1	4			1	1.33	0.3629
Melanoma	1	4					1	1.33	0.3629
Cardiovascular									
all	1	4	1	4	5	20	7	9.33	0.0804
nothave	24	96	24	96	20	80	68	90.67	0.0804
prefterm									
Blood Clot					1	4	1	1.33	0.3629
MI	1	4			4	16	5	6.67	0.0617
Stroke	1	4	1	4	1	4	3	4	1
Gastrointestinal									
all	11	44	6	24			17	22.67	0.0010
nothave	14	56	19	76	25	100	58	77.33	0.0010
prefterm									
Bellyache	2	8					2	2.67	0.1281
Cramps	3	12	3	12			6	8	0.1958
Flatulence	10	40	5	20			15	20	0.0019
Ulcer	2	8					2	2.67	0.1281
Neurological									
all	3	12	15	60	3	12	21	28	0.0001
nothave	22	88	10	40	22	88	54	72	0.0001
prefterm									
Agitation	2	8	2	8	2	8	6	8	1
Headache	3	12	14	56	2	8	19	25.33	0.0001
Skeletal									
all	7	28	4	16	2	8	13	17.33	0.1707
nothave	18	72	21	84	23	92	62	82.67	0.1707
prefterm									
Broken Leg			1	4	1	4	2	2.67	0.5983
Skull Fracture	1	4					1	1.33	0.3629
Vertebral Fracture	7	28	3	12	1	4	11	14.67	0.0506

```
title "Table G8: AE table, with chi-square (75-patient), & trtemgt derive" ;

directoryref a="/home/robert/test" ;
directoryref w="/home/robert/tmp/workdsets" ;

inputdset asc a/patinfo3 patid trt 1*(patid) ;
inputdset asc a/advevt3b patid visit bodysys prefterm severity ;
thing pat uniqval(patid) a/patinfo3 ;

*********************************************
vdt

/// convertfileformat asciitobinary(a/advevt3b->a/advevt3b) colspecsinfirstrow ;

inlist a/advevt3b ;
if (not (visit==1))  deleterow ;
select severity = max(severity) by patid bodysys prefterm ;
sendoff(w/data2) patid bodysys prefterm severity ;


inlist a/advevt3b ;
if (not (visit>1))  deleterow ;
select severity_p = max(severity) by patid bodysys prefterm ;
sendoff(w/data3) patid bodysys prefterm severity_p ;

inlist w/data2 w/data3 ;
mergeby patid bodysys prefterm ;
if (not (severity_p>severity or severity is null))  deleterow ;
sendoff(w/ae_data) patid bodysys prefterm ;

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

n~n(pat) ;

printto "/home/robert/test/outp02" ;
denom trt ;

model chisq(thisrow?*trt*n)

col (trt all)*(n %) pvalue ;
row all have(w/ae_data) bodysys*(all nothave prefterm) ;
```