Manipulating Data:
Elements of the DATA Step Language

Paul M. Dorfman
CitiCorp AT&T Universal Card
Jacksonville, FL

ABSTRACT

In the SAS® DATA step, you can manipulate data by instructing SAS what to do. However, SAS understands only instructions given in its own tongue. The DATA step language (SAS DSL) is simple and powerful, its syntax is crisp and highly readable, and it is easy to learn. In this presentation, we will try to introduce the basics of speaking SAS. Just like in any language, a valid SAS phrase contains meaningful expressions that follow an intelligible sequence. These two principal parts will be considered one at a time:

1. Expressions - basic blocks from which statements are built: Constants - Variables- Arrays- Assignment and Sum- Operators- Functions.
2. Control Flow - the order of execution of instructions: Conditional execution-Branching-Repetition. The presentation should provide an idea about the overall structure and main building blocks of the language used in the SAS DATA step. Some aspects (functions, specific statements) will be touched just briefly as part of the big picture and covered in detail in other Intro to SAS presentations.

I. INTRODUCTION

In a software package as diverse as the SAS System, it would seem to be difficult to pinpoint the most important part. Yet in reality, it is surprisingly easy: The single most powerful element of the entire system is the SAS DATA step. The reason is simple: While SAS procedures and other pre-programmed SAS modules can be applied to solve a variety of particular problems incredibly well, none of them can do it all. Because the language of the SAS DATA step (from now on referred in this paper as SAS DSL: SAS DATA step Language) is an entire language in its own right, you could use it, in principle, to implement just about any programming algorithm of your own (whether it actually makes sense in a particular situation is a different story!). Thus in the Base SAS, the DATA step is the place where programming in its strict sense mainly occurs.

Becoming a professional SAS DSL speaker (or writer if you will) comprises two complementary components that should be embraced simultaneously: Applying it as quickly and as extensively as possible to solve a variety of practical tasks and learning SAS DSL grammar and usage. Restricting the learning process to the former will most probably result in a large amount of distasteful and inefficient code, while the reverse will make one a chef who knows all the recipes but has never actually cooked. Luckily, which part to start with is not a chicken-or-egg question, for it is impossible to speak a foreign language without learning a word. Likewise, you cannot dive head first into the SAS DSL programming without having digested at least a marginal amount of its vocabulary and grammar beforehand.

This is what we are going to concentrate on in this tutorial: Learning the basics of SAS DSL grammar and usage.

II. FACTS

1. WHAT KIND OF LANGUAGE IS SAS DSL?

Historically, SAS DSL has its roots buried in PL/I and resembles its syntax very closely. It seems all the more logical that initially, the SAS System was mainly written in PL/I as its underlying software. Whether by chance or on purpose, this choice has proven to be extremely successful. In many (not so humble) opinions, PL/I is the best and most powerful third generation language (3GL) ever created, and of all languages, its syntax, and therefore that of SAS DSL, is most wonderfully balanced. It is crisp and clear. It is English-like without the viscous wordiness of COBOL and technically concise without being cryptic like C.

From the standpoint of the language generation, it is hard to classify SAS DSL in any clear-cut way. It definitely possesses practically all the features of a 3GL, its level being sufficiently low for implementing just about any common algorithm - although not as low as that of PL/I or C. On the other hand, it is laden with 4GL features such as automatic control flow, list processing, and Output Delivery System.

Besides, SAS DSL comes equipped with an incredible number of ready-to-go programs on call, such as functions, call routines, formats, and informats. They make it possible to achieve, in a few keystrokes and with the utmost confidence in the final correct result, many programming goals that in other 3GLs have to be attained through tedious coding, whose actual quality, being dependent on the local programming expertise, very often goes very wrong. Thus, the variety of powerful and versatile programs on call is, in a sense, a 4GL feature as well.

2. WHAT IS THE SAS DATA STEP?

From the standpoint of a typical multi-step SAS program, the DATA step, along with SAS procedures, is one of separately compiled, top-down executed programs. From the standpoint of general programming, the DATA step is a complete, stand-alone computer program. As such, it has its memory managed separately from any other step, and, all by itself, the DATA step constitutes a complete programming environment equivalent to that of a whole language. For example, any COBOL program can be replicated in a single DATA step (only it is likely to have 1/5 the lines of code, take 1/5 the time to write, and work right the first time).

3. COMPILATION AND EXECUTION

Just as any other general computer program, the DATA step has two distinctive phases:

1. Compilation.
2. Run.

The compilation phase is necessary to do the following:

1. Check the program syntax and translate SAS DSL instructions into machine language (the one the computer understands).
2. Acquire enough computer memory necessary to run the step and organize it appropriately.

3. Perform actions and execute the instructions that should be carried out before the step has begun to run, for example, initialize selected variables and/or arrays.

The only difference between this process and a similar process used in other languages is that the object of compilation (executable file or load module) cannot be saved and subsequently run without the source code and SAS being active on the machine. This means (apart from the special way the SAS software is made available for use) that each time a DATA step is submitted, it has to be recompiled, and that once the source code is lost, the program cannot be run. However, neither could be considered a disadvantage. SAS DATA step compiler is extremely fast, and the compilation time is usually infinitesimally short, even for large DSL programs. As to the ability to run a program without the source code, it is considered insane even in the shops using languages that do provide for it, because it makes it impossible to change, update, correct – in other words, maintain the program.

4. THE SHELL

Any DATA step begins with the keyword DATA and ends when it has encountered the keyword RUN. It will also stop if it sees another keyword DATA or keyword PROC, or if it is the end of the entire batch SAS program, because in this case the SAS Supervisor (the internal program SAS uses to coordinate all steps) understands that either the next step is about to begin, or the current step is final.

There are plenty of good reasons to always finish a step up with the RUN keyword. For instance, if you are running SAS interactively, and the last step in the program is not closed with RUN, you will be waiting forever before seeing a result, even though there is a message displayed telling that the step is running. There also exist situations when establishing an explicit step boundary with RUN is required for other things to happen after the step has executed. Here it is assumed that it is always the case, and each DATA step has RUN as its last statement. In other words, we will view the space between the DATA and RUN keywords as a *shell*, within which a DSL program is written.

Hence, in principle, any conceivable DATA step program looks like this:

```
Data < output SAS data set list > ;
  …
  < SAS DSL program instructions >
  …
Run ;
```

In the simplest case, when no output SAS data sets are to be created, it is indicated by the keyword _NULL_. The simplest DATA step program containing no instructions at all is thus:

```
Data _Null_ ;
Run ;
```

III. THE COMPONENTS

1. AB OVO

The usefulness of the program above is somewhat limited by the fact that at run time, it does nothing, for there are no instructions to execute. However, for our current purposes, it is not completely useless, because there are a number of things one can ascertain from it.

First of all, there are two distinct *language statements* starting with DATA and RUN.

Secondly, each statement is separated from the subsequent statements with a semicolon. And, in fact, one of the fundamental SAS syntax rules mandates, that, yes,

- Each DSL statement must end with a semicolon.

Thirdly, there are certain words in DSL that mean something special to the DATA step compiler; that is, unlike other words, they tell the compiler to do concrete things. Such words are called keywords. SAS relies on their presence, sequence, and place in the program to figure out what it is that you are trying to tell the software to do.

Finally, if we change the entire program to the upper case, or reverse the case of each letter, it will still run the same way. To SAS, the words case, Case, and CASE all mean the same. In other words:

- You can write a DATA step program in any case you want - SAS could not care less.

Now let us write a more involved (and useful) program that will let us notice, examine, and learn more interesting things. Suppose you want to submit a SAS program at any time during the working hours, and you want it to tell you, depending on the current time, what action to take and how much money you have earned so far, given an hourly rate. Here it is:

```
Data _Null_ ;
  Retain Hour_Rate 100.00
     Start_Time '09:00't ; Length Message $ 15 ;

  Hours_Worked = ( Time() - Start_Time ) / 3600 ;
  Money_Earned = Round (Hours_Worked*Hour_Rate, 0.01) ;

  If      Hours_Worked => 8 Then Message = 'Go Home!'  ;
   Else If Hours_Worked => 4 Then Message = 'Lunch!'    ;
      Else                        Message = 'Keep Coding!' ;

       Put Message Money_Earned = Dollar7.2 ;
Run ;
```

This, still a very basic program, already incorporates almost all components shared with its more complex siblings.

2. FREE-FORM

First, let us notice two quite convenient features of SAS DSL:

- There is no particular column in the code where a statement must begin.
- A statement can be split between two (or more) lines of code.
- More than one statement can be located on a single line, as a semicolon delimits the statements from each other.

So, even though it is a *good programming practice to have one statement (or several short statements with identical meaning) per line*, this discretion belongs to the programmer, not the compiler. To the latter, a statement is a group of symbols between two semicolons; the rest is up to the programmer. All this can be summed up as follows:

- SAS DSL is a free-form language.

Being free form is an extremely important and convenient feature of the language. It lets *you* decide what style of coding – spacing, indentation, blocking – you prefer, and it does not force you to start certain statements in certain columns. Just ask any SAS or PL/I programmer who has also programmed in COBOL.

3. TOKENS

Looking at the above program, a naked human eye will immediately distinguish a variety of different things seeming to possess an intrinsic meaning. For example, in the group

Hours_Worked*Hour_Rate,

Hours_Worked, the asterisk, and Hour_Rate all appear to mean something specific, although there are no blanks between them to tell them apart. In part, the SAS compiler thinks the same way. When it looks at the program, it separates the asterisk, based on its special meaning, from the rest of the words, and interprets them based on the syntax rules and context.

Such smallest program unit, which SAS compiler perceives as having an intrinsic meaning, is called a *token*. There are only three types of tokens in SAS DSL:

- Words
- Symbols
- Constants

Words differ in nature, depending on whether they are *keywords* or *names*. Names and symbols combine in *expressions*. Keywords and expressions make up *statements*. Let us consider the SAS DSL elements one at a time.

4. WORDS

As noted above, all words in a DATA step program fall into two distinct categories:

- Keywords
- Names

5. KEYWORDS

Keywords are words that have a special predefined meaning to the SAS compiler – and hence in the SAS syntax. Most often a keyword is used to begin a SAS statement, but they also occur at particular locations inside statements. In the sample program above, the keywords are DATA, RETAIN, LENGTH, ELSE, PUT, RUN (at the beginning of statements), IF (at the beginning and inside statements), and _NULL_ (inside the DATA statement). There are dozens of other keywords in SAS DSL – listing all of them here would mean retyping a good chunk of the SAS Language reference, and it is not the goal here. However, we will meet more keywords while discussing other language elements and sequence control.

6. NAMES

A name in a SAS DATA step program is a word the programmer comes up with to identify an object intended to be utilized or manipulated by the program. Looking a bit forward, such objects, for instance, may include variables, SAS data sets, statement labels, and arrays.

This definition does not imply, however, that any collection of symbols of any length can be used for a name: The name must *at least* be a *valid SAS name*, plus other limitations may apply, depending on the object type.

What is a valid SAS name? It is an identifier that:

1. Is no longer than 32 characters.
2. Begins with a letter or underscore.
3. For other characters, has letters (A-Z, any case), digits (0-9), or underscores.

For example, the names used in the sample program and shown below on the left are valid; a slight alteration in the direction of not complying with the rules above (the column on the right below) makes them invalid:

Valid Names	Invalid Names
Hour_Rate	Hour Rate
Start_Time	Start-Time
Message	1Message
Hours_Worked	#HoursWorked
Money_Earned	Money$Earned

Note that the case of the identifiers is irrelevant: They can be all in upper case, lower case, title case, sentence case, or mixed.

As mentioned, there are objects whose naming conventions are even more restrictive than a valid SAS name. Such is the case with formats and informats allowing only 6 to 7 characters in their names. File references (filerefs), i.e. names used to point to external files, mainly have to conform to the naming conventions of the operating system used. Thus, under OS/390 (or z/900, for that matter), they are limited to 8 characters, allowing, on the other hand, the dollar and pound signs in the names of filerefs.

At the first glance, the rules above seem to impose a lot of restrictions on SAS names. However, it is a wrong impression. They are no more restrictive than names in other languages, yet enjoy a tremendous privilege: SAS DSL has virtually no *reserved words*. That is, the SAS compiler is smart enough to understand the phrase like

IF THEN = 10 THEN IF = THEN

unambiguously from the context, by identifying the first IF and second THEN as a keyword, and giving the first and third THEN the meaning of a variable. Note that even though *it is not recommended* to code this way, the feature completely rids the SAS programmer of the pesky burden imposed by hundreds of reserved words in some other languages (notably COBOL), where using a reserved word for a user-defined name means a failure to compile. Once again, SAS shifts the responsibility of maintaining the correct syntax from the programmer to the compiler. Let us summarize it as another SAS DSL advantage:

- SAS DSL has virtually NO reserved words to remember and/or to avoid.

6.1. SPECIAL NAMES

The above notion is all the more amazing in the light of the fact that there are also dozens of *special names* in DSL that SAS uses for its internal purposes. Most of them begin and end with an underscore, but only few of them are genuinely reserved, that is, a programmer cannot use them arbitrarily as user-defined names. However, even in these cases, there is usually a highly logical reason for having it this way. For example, such reserved words as _ALL_ (points to all variables currently in memory), _ NUMERIC _ (the same for all numeric variables) cannot be used as a variable name - but it is illogical to use a name for a single variable and a collection of variables at the same time, hence such usage is banned. On the other hand, such reserved names as _N_, _IORC_, _I_, and numerous others can be actually used by a programmer for uses other than intended (*sometimes*, it makes sense) with no harmful consequences whatsoever.

7. EXPRESSIONS

An alert reader has no doubt noticed that symbols and constants have been left out of discussion so far, even though the logic would certainly dictate it. This is because it appears even more logical to discuss them while considering SAS DSL expressions – the largest meaningful blocks of which SAS statements are composed.

A natural language sentence consists of one or more expressions. A natural expression is, loosely speaking, a group of words and punctuation symbols, collectively bearing a concrete meaning. The same is basically true about a SAS DSL statement. Defined in a more formal manner, an expression is a group of words and symbols resolving to a concrete value.

Let us identify a couple of expressions in the sample code above:

```
( Time() - Start_Time ) / 3600
Round (Hours_Worked*Hour_Rate, 0.01)
```

Actually, each of these expressions are *composite expressions*, because they are in turn composed of expressions conjugated by SAS operators - in the case above, arithmetic operators (-, /, and *). So, the question is, what kind of tokens can a *simple expression* contain? A simple expression can only incorporate:

- Constants
- Variables
- Array references
- Function calls

Any expression, by its definition, resolves to a value. But in SAS, all values are of two data types only: Numeric and character. Accordingly, there are only two types of SAS expressions:

- Numeric, i.e. resolving to a number.
- Character, i.e. resolving to a character string.

For instance, both expressions given as an example above are numeric. However, in the expression

'Character ' || 'expression' ,

the concatenation operator || combines two character constants producing a (longer) character value; hence, this is a character expression. Now let us look at the components making up expressions separately.

7.1. CONSTANTS

A constant is a value in a DATA step program that the program cannot change. Once again, as it is the case with all SAS values, there are two major types of constants: Numeric and character. In the sample program above, all of the

100.00, '09:00'T, 15, 3600, 0.01

are numeric constants, while all of the

'Go Home!'
'Lunch!'
'Keep Coding!'

represent character constants. In turn, each constant type has several subcategories. First, let us consider numeric constants.

7.1.1 Numeric Constants

When you code something like 0.035, or '05AUG2001'D, or even 1.34E-15, its meaning is obvious to you, but the compiler first must convert each of these things into a form suitable for SAS internal consumption. SAS interprets them using special subroutines called informats. At compilation time, the informats take numeric constants coded in the program and convert them into 8-byte floating point numbers. Numeric constants can be one of the following:

1. Decimal/scientific notation. This is a conventional way you would write a number using a decimal point and a leading sign if necessary, with no commas separating orders of magnitude. Or, if a number is quite large, you may choose to write 1230000000, say, as 1.23E+10 (plus sign is not mandatory). E.g., 1.234, 2.34, -3.1416, 0.001956, 1.6E-19, 4.8E+13 are all valid decimal/scientific notation constants. SAS interprets their meaning using a standard numeric informat W.D.
2. SAS date. Date constants are written as '05aug2001'D or '17jan97'D. SAS uses DATE9. and DATE7. informats to convert date constants into the number of days since the beginning of 1960, and then stores the number internally, just as it would store any numeric value.
3. SAS time. When you write '23:10:44.123'T, you are telling SAS that it is 23 hours, 10 minutes, 44 seconds, and 123 milliseconds since the beginning of the day. However, SAS must understand it only as the number of seconds since the beginning of the day – and hence performs the necessary transformation using its TIME. informat before the constant can be used in the running program. The number of seconds thus obtained is stored internally as a numeric variable.
4. SAS datetime. This is written as, for example, '31aug2000 18:27'DT, and its meaning is obvious for a human. SAS makes it obvious to the computer by turning it into the number of seconds from the beginning of 1960 using the informat DATETIME15. Then it stores the number internally.
5. Hexadecimal. If you have a constant number in the hexadecimal notation and need to use it in a program, SAS will use its HEX. informat to interpret it for you (so you do not have to convert it to a decimal or binary base yourself) and store it appropriately. Just append an X to the end of the number; and if it starts with a letter (letters from A through F used as digits from ten to fifteen in hex), precede it with a 0 – otherwise the compiler will be confused thinking that it is a valid SAS name. Thus, 5A7F8x is a valid hex constant, but the hex number DEC45x is not. To make it valid, write it as 0DEC45x.

SAS DSL approaches the issue of interpreting constants quite efficiently, by converting them to their proper internal representations at compile time. It makes it unnecessary to do the transformations at run time – possibly many times if, as it often happens, a statement containing a constant is executed repeatedly.

7.1.2. Character Constants

There are only two kinds of character constants:

1. Character literal. It is the most usual character constant written as a character string enclosed in single or double quotes. 'I am hungry…', '12345', 'SAS Version 9' are examples of character literals, called literal perhaps because they are *literally* the values they represent – not just values whose actual meaning to the computer must be further interpreted.
2. Character hexadecimal. Each character in a character string has a predefined number (rank) from 0 to 255 in a so-called collating sequence. Any number up to 255 can be written as a 2-position hexadecimal number. The lowest character is thus ranked 0, and it is 00 in hex. The highest character is ranked 255, which is FF in hex (16*15+15). So, each character can be written as a 2-position hex number. A character hex constant thus always has an even number of hex digits enclosed in quotes and is marked with the letter X at the end. For example, '000000'x, 'FFFFFFFF'x, '40'x are valid character hex constants. Most often, such constants are used to indicate unprintable characters that, by their nature, cannot be typed in the program explicitly.

7.2. VARIABLES

To manipulate data, the programmer needs a data object whose memory contents can be changed, altered, replaced, copied, or erased by a program. This all-important purpose is served by SAS *variables*.

What is a variable? The loose definition just given is actually not all that bad. But let us take a little bit more mechanical approach and examine the first three statements after the DATA statement in the sample program:

```
Retain Hour_Rate 100.00 Start_Time '09:00't ;
Length Message $ 15 ;
Hours_Worked = ( Time() - Start_Time ) / 3600 ;
```

7.2.1. Variable Mechanics

The SAS compiler parses the program in a strictly top-down manner. From the context of the first statement, it understands that RETAIN is a keyword, and 100.00 and '09:00'T are constants. Hour_Rate is the first valid SAS name in a right place to be interpreted as a variable. It is the first variable in the entire program; it has not been referred before; and it is being asked that it should be assigned a numeric value. To SAS, that means an order to organize an 8-byte cell for a numeric variable in the area of memory where retained numeric variables should dwell. Boxes in memory have no names, but only numbers called addresses. Thus SAS must prepare a table in memory where the number of the cell allocated for Hour_Rate will be associated with this name. Being logical, SAS stores some other useful information in the table together with the name and address – in this case, that the variable is numeric, and hence it is 8 bytes long. The table (called *symbol table*) is organized in such a way that given a variable name, all associated information can be retrieved almost instantly. Having been done with all that important work, SAS can now convert 100.00 to its 8-byte internal number and move it into the cell in memory it has prepared for it.

Essentially the same process will be repeated with Start_Date; only because it is not named in any RETAIN statement, nor is it retained by default, SAS allocates a cell for it at the beginning of a different area in memory, where all non-retained numeric variables belong. Yet another memory area will be used to store the variable Message. Upon seeing that in the LENGTH statement, Message is declared as character (because of the dollar sign preceding it), SAS allocates a 15-byte long cell in the domain where non-retained character variables will dwell. At this time, SAS does not yet know what to move there (the value is *missing*), so it uses blanks – standing as missing values for character variables.

A similar situation occurs with Hours_Worked. SAS knows that it is numeric because there is a numeric expression waiting to be resolved and assigned to it, but it can only happen at the run time – the TIME() function returns the time at the moment when the instruction is being executed. Therefore, so far, at compile time, the value for Hours_Worked is missing. This variable is numeric, so the corresponding missing value should be numeric. For this purpose, SAS uses one of the special NAN (not a number) binary values called *standard numeric missing value* and moves it into the cell (now residing in the non-retained numeric area of memory) prepared for Hours_Worked. Using its standard numeric format, SAS *prints* it as a period (.). This very period should be used as a constant if you want to explicitly initialize a variable to a standard missing value during compile time.

7.2.1. Automatic Variables

You may not have defined any variables in your DATA step program, yet there are always at least three automatic variables in the step DSL allocates no matter what: _N_, _ERROR_, and _IORC_. At compile time, _N_ and _IORC_ are set to 1, and _ERROR_ is set to 0. At run time, _ERROR_ is changed to 1 if either a run-time error condition occurs, or an I/O operation sets _IORC_ to a value other than 0. _N_ is assigned the number of times (accumulated independently in some internal variable) program control is transferred to the instruction immediately following the DATA statement.

Also, the DATA step compiler creates other automatic variables, such as _I_ and _IORC_, and many others when it detects certain statements in the step.

Automatic variables reside in areas of memory from where nothing is written to any output SAS data set - which is another way to say that they are *automatically dropped*.

7.3. ARRAYS

In most cases, variables in a program are called by name when they are incorporated in an instruction prescribing what to do with their values. The same most often happens in life with things less abstract than SAS variables, for example, with months, when we mark dates in a calendar. However, one may find it more convenient to call the months by their ordinal number instead of by name. SAS arrays provide an opportunity to do exactly that with SAS variables in a program.
For example, if we have seven variables SUN, MON, TUE, WED, THU, FRI, SAT, and if we need to do something with them in a program, we could call them only by their names. However, if we add one of the statements
ARRAY Day (*) SUN, MON, TUE, WED, THU, FRI, SAT ;
ARRAY Day (7) SUN, MON, TUE, WED, THU, FRI, SAT ;

before any of the variables above have been referenced, we will be able to refer to them *both by their native name and ordinal number*. To SAS, SUN and Day(1), TUE and Day(3), and so on, will mean all the same. The number inside the parentheses is called *array index or subscript*. It does not have to be a constant; instead, it can be any numeric expression resolving to a number between 1 and 7. Note that the array bounds are set at compile time, and they must be integer constants (or macro variables resolving to such) – the compiler does not tolerate even a decimal point.

Array references can be used in expressions just as well as variables. Their advantage over non-arrayed variables is that references to arrayed variables can be made dynamic by changing the value of the index. For instance, if at run time, you need to move the value of 123 to all seven variables, you can set X to 1, and then instruct SAS to repeat the instruction Day(X)=123 seven times, each time changing the value of X up by a unity.

Arrays are data structures so powerful and with so many more features and uses in SAS DSL that this little array subsection could be easily expanded into a thick tome. Here, we have just enough room to briefly sketch where arrays belong in the DATA step.

7.4. FUNCTIONS

A function is an encapsulated stand-alone program that return a value given (an)other value(s) called argument(s). In real-word SAS programming, it is quite difficult to find an expression without a function call. Even in the program as basic as the one being used as a sample, TIME() and ROUND() are both function calls.

There are no user-defined functions in SAS DSL (which is certainly a disadvantage compared to some other tongues, notably PL/I or C). However, this is to a large degree compensated by the fact that SAS provides hundreds of intrinsic (i.e. coming with the language, pre-programmed, guaranteed-to-work-right) functions, from finding a character in a string, to evaluating the distance between words, to computing almost anything that could be thought of being necessary to calculate in business and statistics.

7.5. OPERATORS

To make a new expression out of several existing expressions in a natural language, people use prepositions. In programming in general and SAS DSL in particular, this role is played by the tokens called *operators*. Expressions they conjugate are called *operands*. In the sample program, for example, the function Time(), variable Start_Time, and constant 3600 are all elementary expressions (consisting of a single element). The subtraction operator creates the expression Time() - Start_Time, and then the division operator uses it and 3600 as operands to create the expression (Time()-Start_Time) / 3600. Finally, the assignment operator (=) tells SAS to take the value, to which the expression on the right resolves, empty the memory cell belonging to the variable Hours_Worked, and fill it with the resolved value instead.

Operators discussed above are represented by symbols, but it does not have to be the case. They can also be keywords. For instance, MAX, NOTIN, AND, OR operators are keywords. However, they, as well as many others, can also be written as symbols. Thanks to the wisdom of DSL developers, the symbols and keywords used for SAS operators make their purpose pretty much self-explanatory. The way arithmetic operators are used in formulae, such as shown above, conforms to the standard algebraic rules, as well as the priority of their evaluation. If there is any question about the default priority of evaluating an expression, it can be always enclosed in parentheses, which will force SAS to evaluate it first.

7.6. ASSIGNMENT

Assignment statement is used to replace the value of a variable in its memory cell. The assignment operator looks like an equal sign, but by no means is it an equivalent of an ordinary algebraic assignment. This is compounded by the fact that in SAS, an equal sign may be also used as a comparison operator, also written as EQ. The general assignment form is as follows:

< Variable > = < Expression > ;

What occurs here is basically very simple:

1. The expression on the right is evaluated.
2. The resulting value is stored in some intermediate memory location.
3. The content of the variable cell is erased.
4. The new value is moved to the variable cell.

For example, in the light of this process, let us dissect an assignment operation

X = X * Y ,

Algebraically, it is meaningless. However, it is perfectly meaningful programmatically: It retrieves the value of X from the X-cell and multiplies it by the value extracted from the Y-cell; then the result is used to replace whatever value originally resided in the X-cell (and was multiplied by Y) before. Any confusion can be avoided if the above 'formula' is thought of as 'Set X to the product of X and Y'.

7.7. SUM STATEMENT

There is another form of assignment statement pertaining to addition only (or subtraction, for that matter):

< Variable > + < Expression > ;

When the compiler sees such a thing, it:

1. Allocates a cell for the variable in the area of memory segregated for retained numeric variables.
2. Moves 0 to the cell.

This is logical because in most cases, the SUM statement is used to accumulate numeric stuff, and such fields are usually zeroed out before the accumulation begins. At run time, the expression on the right of the plus sign is evaluated, the result is added to the value of the variable, and the sum is assigned to the variable.

If instead of adding the value to the variable, the same amount needs to be subtracted, a unary operator can be appended to the left of the operand-expression:

< Variable > + - < Expression > ;

Those who care about the symmetry and would like to make SUM statements stand out in the code more prominently, might want to double the single plus used in the additive statement:

< Variable > ++ < Expression > ;

It has the same effect, as a single plus but is very easy to spot. And, of course, any C/C++ programmer will understand its meaning without saying.

II. CONTROL FLOW

Now that we have seen, albeit briefly, what sort of stuff SAS DSL instructions comprise, it is necessary to get an idea how to control the order in which they are executed at run time. Such order is called *control flow*. The instruction currently being executed is said to have control. After it has been executed, control is transferred in the top-down manner straight to the next instruction, in the absence of special branching instructions altering the top-down sequence control.

1. SELECTION

The sample program we started with contains no such provisions until the IF-THEN-ELSE structure is encountered:

```
If      Hours_Worked => 8 Then Message = 'Go Home!';
Else If Hours_Worked => 4 Then Message = 'Lunch!'  ;
Else                          Message = 'Keep Coding!' ;
```

At this point, depending on the time of the day, one of the three instructions is performed. This is termed *conditional execution*; and such a structure itself is called *selection*. After the selection has been made, the remaining instructions are executed in a straight sequence, control reaches the bottom of the step, and the program stops.

The instruction executed when a selection is made does not have to stand-alone. It can be combined with other instructions executed when the same criterion is met. However, in this case syntax demands that all of them must be enclosed in a so-called DO-END block. For example, if in the case #2, you would like to display the values of all variables at this point in the program, it could be coded as follows:

```
If Hours_Worked => 8 Then Message = 'Go Home!';
Else
If Hours_Worked => 4 Then Do;
       Message = 'Lunch!' ;
       Put _All_ ;
End ;
Else Message = 'Keep Coding!' ;
```

When numerous changes to the program's logic can be anticipated, it makes sense to place even a single conditional instruction within a block, for then it is much easier to insert any number of additional instructions into the same block as needed.

2. PLAIN BRANCHING (GoTo)

The excerpt above could be written in a different style using the GOTO statement. GOTO performs what is known as simple branching:

```
If Hours_Worked => 8 Then GoTo One ;
If Hours_Worked => 4 Then GoTo Two ;
If Hours_Worked => 4 Then GoTo Three ;
One: Message = 'Go Home!'; GoTo Exit ;
Two: Message = 'Lunch'; Put _All_; GoTo Exit ;
Three: Message = 'Keep Coding!' ;
Exit:
```

To mark the point to which a GOTO transfers control, statement labels are used. Above, they are represented by the keywords One, Two, Three followed by the colons. A label must be a valid

SAS name, and of course, labels must be unique. Any statement can be preceded by a label without the harm of affecting control flow. They become operational only if a GOTO transfers control to one of them. Note that the logic of simple branching renders the ELSE keywords unnecessary.

Programming this way instead of using a selection structure is not particularly recommended. Moreover, SAS provides a special SELECT structure specifically designed to program this kind of situation without a single GOTO or IF-THEN-ELSE:

```
Select ;
  When (Hours_Worked => 8) Message = 'Go Home!';
  When (Hours_Worked => 4) Do;
        Message = 'Lunch!' ;
        Put _All_ ;
  End ;
  Otherwise Message = 'Keep Coding!' ;
End;
```

In actuality, GOTOs have advantages, and do no harm when used with discretion and understanding of control flow.

3. REPETITION

In programming, the concept of the repetitive execution is one of the most significant. It is because of the repetitive execution that the computer can be instructed to perform almost the same stuff millions of times, altering its modus operandi with each iteration just as much as prescribed by the programmer – which is exactly the kind of thing computers are needed for in the first place.

The GOTO command is the most elementary instruction that, in a combination with IF, provides for organizing a controlled repetition. In the example of the array Day(*), filling all its elements with 123 could be coded this way:

```
ARRAY Day (7)  SUN, MON, TUE, WED, THU, FRI, SAT ;
X = 1 ;
Assign: Day(X) = 123 ;
       X = X + 1 ;
       If X < 8 then GoTo Assign ;
```

Once control reaches the conditional statement, it evaluates the current value of X. If it is still within the array boundaries, control is transferred back to the ASSIGN label, and the process repeats until X has become equal to 8. After that, control simply goes to the next instruction. Since it will inevitably happen, it prevents the structure from iterating endlessly.

Luckily, SAS DSL provides a separate piece of syntax, the so-called *DO loop structure*, solely devoted to the task of organizing repetitive execution in a flexible and robust manner, thus rendering the IF-GOTO pair almost unnecessary for this purpose (with quite rare exceptions of intricate, high-performance cases). Let us see how it might look like using the array example above:

```
Do X=1 By +1 Until ( X = 7 ) ;
  Day (X) = 123 ;
End;
```

The block of statements between the DO and END statements is called the *body of the loop*. In this case, it consists of a single statement, Day(X)=123. How does SAS know that this syntax instruct it to execute the body repeatedly – as opposed to a mere DO-END block we have seen earlier? The whole trick is in the presence of one of the two keywords – BY and UNTIL – in the head of the loop. Either one can cause the loop to iterate, albeit in different modi operandi.

The first thing that occurs in the loop head is 1 being moved to the variable X. That this *loop counter* is also an array index has no special meaning: To SAS, it is just another numeric variable. The presence of such a *FROM-value* in the head of the DO loop makes it being termed an *iterative DO* (although there is no significance in such terminology: The ability to iterate is the main feature of any loop). It is important that the FROM-value is assigned only once and forever, before the loop begins to iterate. The presence of the UNTIL condition tells SAS to:

1. Execute the body of the loop.
2. *At the bottom of the loop*, increment X by 1.
3. Still at the bottom of the loop, check whether the condition in the parentheses is true.
4. If it is, terminate the loop hand control over to the next instruction.
5. Otherwise, transfer control back to the top of the loop.

Apparently, the logic is precisely the same as that of IF-GOTO, but important advantages lie in not having to organize the loop by hand and eluding the code inquisition looking for GOTOs as if they were witches. Moreover, using a DO loop does not preclude one from exiting it at any point from within the loop by using the same old good GOTO, even though it is almost never warranted.

The UNTIL condition can be any conditional expression, not necessarily an array index comparison. In fact, for the latter, SAS provides a special keyword, TO, where you can specify the upper value of the index to be tested. The loop above could thus be (and usually is) written as

```
Do X=1 By 1 To 7 ;
  Day (X) = 123 ;
End;
```

The TO clause can follow the BY clause and vice versa. Just like with the FROM-value, a *TO-value* is assigned only once, before the loop starts iterating. This means, in part, that one should not be fearful to use expressions as TO and FROM values, for they are evaluated but once.

Obviously, with UNTIL and in the absence of a TO-value, the loop will iterate forever unless the UNTIL condition becomes true at some point in time. Oftentimes, though, it is more convenient to make a loop iterate *while* a condition is still true – which is, quite logically, achieved by coding WHILE instead of UNTIL. In this case, the condition is checked *before the body is executed even once* – so there is a possibility that the loop will never iterate once. Coding our sample loop using DO-WHILE will look like this:

```
Do X=1 By +1 While ( X < 8 ) ;
  Day (X) = 123 ;
End;
```

In the UNTIL loop, when X has become 8 at the bottom, control is immediately moved to the instruction following END. In the WHILE loop, control instead goes to the top of the loop where the condition X < 8 is tested. Since now it is false, the body is skipped, control is transferred all the way past the END, and the loop stops.

3.1. ITERATING AD INFINITUM

So, if a DO-UNTIL loop stops iterating when and if its parenthesized expression evaluates to true (that is, to any number *but* zero or any missing value), a DO-WHILE loop ceases repetition when and if its condition becomes false. Conversely, if the UNTIL condition is always false, or the WHILE condition is always true, the loop will iterate forever. Since 0 is always false, and 1 is always true, an infinite looping may be organized as either of the following:

```
Do Until (0)  ;
  < body of the loop >
End ;

Do While (1) ;
  < body of the loop >
End ;
```

The same effect can be achieved by placing FROM- and BY-values in the loop header all by themselves: The loop will iterate forever, even if the BY-value is 0. This comes in very handy when it is desirous to organize an infinite repetition and initialize a numeric value before the loop starts in the loop head, without a separate statement:

```
Do X=1 By 0 ;
  < body of the loop >
End;
```

However, the presence of a FROM-value without a BY-value will cause the loop to iterate maximum once, no matter what an UNTIL or WHILE clause may contain, if anything (and of course it will not iterate once if the WHILE condition is false a priori).

3.2. GETTING OUT OF A LOOP

What is a practical value of organizing an endless iteration? It adds another layer of flexibility to the repetition structure by making it possible to stop the loop using a condition *in its middle*, rather than on the top or at the bottom already provided by WHILE and UNTIL. SAS provides two principally different ways of branching from inside a loop, regardless of whether it is infinite or not:

1. LEAVE statement.
2. CONTINUE statement.

Suppose that in our sample loop, we want to fill the array with 123 only up to the current weekday. Of course, its number can be put in the UNTIL condition or TO-value. But it can be also be achieved in the following manner:

```
Do X=1 To 7 ;
  Day (X) = 123 ;
  If WeekDay ( date() ) = X then Leave ;
End;
```

The LEAVE statement transfers control immediately past the END statement to the next instruction. It is precisely the same as if we coded:

```
Do X=1 To 7 ;
  Day (X) = 123 ;
  If WeekDay ( Date() ) = X then GoTo Exit ;
End;
Exit :
```

So, in cases when it is easier to organize logic in an infinite loop than in a top- or bottom-terminated loop, or when it is necessary to branch out from the middle, a conditional LEAVE allows doing so without resorting to a GOTO.

The CONTINUE statement also provides for a kind of branching, but instead of transferring control *past the bottom* of the loop (past the END statement), it transfers it *to the bottom* of the loop (just before the END statement) without affecting any things that occur at the bottom of the loop by default (incrementing the loop index, comparing it to the TO-value, or evaluating an UNTIL condition, if any). As an example, suppose that we want to fill out only the array elements corresponding to the even days (in other words, skip the odd ones):

```
Do X=1 To 7 ;
  If Mod (X, 2) > 0 Then Continue ;
  Day (X) = 123 ;
End ;
```

The Mod(X,2) function returns the remainder of X divided by 2, i.e. a number greater than zero if X is odd. If this is the case, the CONTINUE statement causes control to jump right before the END statement, thus skipping the assignment when X is odd. Of course, by logic, it is exactly the same as if we wrote using a label for the END statement:

```
Do X=1 To 7 ;
  If Mod (X, 2) > 0 Then GoTo Bottom ;
  Day (X) = 123 ;
  Bottom :
End ;
```

Of course, above, the goal could be reached without either CONTINUE of GOTO, by imposing an opposite condition on the assignment statement itself, but for illustration purposes, this example is perhaps as good as another.

4. FILE PROCESSING

One of the most compelling reasons why we need iterative structures is the necessity to process files with a great number of records. It can be an external file, a SAS data file, a view to an RDBMS – it does not really matter. What does matter is that in most cases, the file processing means to do the following things:

1. Perform some preparatory work.
2. Read all or many records one by one and apply the same logic to each of them, for instance:
    a. Identify relevant data coming with each record as variables
    b. Manipulate certain quantities according to certain rules
    c. Output information for each or selected records (or none)
3. Maybe output summary information, i.e. pertaining to all records collectively.

4.1 PRACTICAL EXAMPLE OF FILE PROCESSING

As a practical example of applying this principle, let us imagine a SAS data file (set) called STATES having 2 variables: 1) 2-byte character variable STATE containing a state abbreviation code 2) numeric variable WATER representing the annual precipitation in millimeters for a certain year. What we want to do is:

1. Print the time when the step started running in the SAS log.
2. Read the file STATES and calculate the average WATER across all states beginning with the letter A, accounting only for non-missing values of WATER.
3. Write the observations containing the states starting with A and with non-missing WATER values to a SAS data set AWATERNMISS.
4. Print the average amount in the log in whole mm.
5. Print the total processing time in the log in seconds and milliseconds.
6. Stop.

Here is one way to write a SAS DATA step conforming to these specifications:

```
Data AwaterNmiss ( Keep = State Water )  ;
  Time = Time() ;
  Put Time = Time. ;
  Do Until ( End_of_File = 1 ) ;
    Set States End = End_of_File ;
    If State NE: 'A' or Water = . then Continue ;
    NonMiss     ++        1 ;
    TotalWater ++ Water ;
    Output ;
  End ;
  Average = Round ( TotalWater / NonMiss, 1 );
  Put Average = Comma. ;
  Time = Round (Time() – Time, .001 ) ;
  Put Time = ;
  Stop ;
Run ;
```

It deserves some explanation:

1. Since an output SAS data set is needed, it is named on the DATA statement.
2. Function TIME() obtains the current time from the computer clock. It is assigned to the variable TIME.
3. PUT statement prints TIME in the log using TIME. format.
4. DO starts the loop reading the file record after record, but the loop should know where to stop. The last record on the file is marked by the system. To recognize it, SAS provides an END=<variable> option on the SET statement. The variable named in END= (here chosen as End-of-File) has the value 0 for all records but the last. When the last record has been read, the software detects the end-of-file marker and moves 1 to End-of–File. At the bottom of the loop, it makes the UNTIL condition true, and the loop terminates.
5. The SET statement always reads the *next record* from the file (at the beginning of the file, it is the first record), so that in the next iteration of the loop, the next record is read. When the record is read, its values for STATE and WATER are moved into the like named cells in memory the compiler has prepared, thus replacing the previous contents of the cells.
6. NE: operator compares the first byte of STATE with 'A'. If a match is not found, CONTINUE transfers control to the bottom of the loop, and the next iteration begins (unless the end of file has been detected). Otherwise the condition after the OR operator is checked. If true, CONTINUE executes as described. If neither condition is true (STATE does begin with A and WATER is not missing), the next body statement is executed.
7. The Sum statement adds 1 to the variable NONMISS used to count the number of A-states with non-missing WATER, and WATER is added to TOTALWATER used as an accumulator.
8. OUTPUT statement moves the current values of STATE and WATER from their cells in memory to the output record and writes the record (observation) out.
9. After the loop is terminated (all records have been processed), AVERAGE is calculated. The ROUND function rounds the value to the nearest integer, because of 1 as the second argument.
10. AVERAGE is printed in the log using the COMMA. format to show commas separating integer orders of magnitude.
11. The start time determined at the beginning of the step is stored in the memory cell called TIME. The meaning of the statement Time = Round(Time()-Time,.001) is this. Retrieve the value of TIME from its cell; subtract it from the current time; use the result to replace the original value of TIME in its cell. Finally, round the result to the nearest thousandth (because of .001 as the second argument).
12. Print the execution time in the log.
13. STOP. This statement is necessary, otherwise the statements from the top of the step to the SET statement will be re-executed.

4.2. AUTOMATIC CONTROL FLOW

There exists a much trumpeted SAS DSL feature instructors routinely use to underline differences between DSL and other languages: The *automatic control flow*. Contrary to the pretty common faith, it only manifests itself when the compiler detects a file-reading keyword, such as SET, MERGE, UPDATE, or INPUT. In this case, the compiler constructs the loop exactly similar to the DO-UNTIL loop (together with end of file checking) by default, and all programming statements written in the step end up being encapsulated in this *automatic 'observation loop'*, as it is frequently called. In simple cases, making use of this invisibly present loop allows to avoid writing an explicit loop as shown above. In addition, SAS busies itself with two things:

1. Moves missing values to all variables not residing in the memory areas the compiler has designated as 'retained only'.
2. Counts the number of times control is transferred to the top of the loop by incrementing an internal counter and moving its value quantity to the automatic variable _N_ right before the next iteration begins.

However, SAS leaves no gaps between the DATA statement and the beginning of its automatic loop, or between the bottom of the loop and the RUN statement. That is, no programming statements can be inserted before and after the automatic loop. Emulated explicitly, the automatic loop might look like this:

```
Data ….. ;
  < Move missing to all non-retained variables >
  Do Internal = 1 By +1 Until ( End_of_File ) ;
    _N_ = Internal ;
    < Top of the step as we see it >
     ……
    < Our instructions >
     ……
    <If no explicit output statement elsewhere OUTPUT>
    < Bottom of the step as we see it >
  End ;
  Stop ;
Run ;
```

It is all dandy if the body of the loop is only what matters, because then the automatic loop makes things a little bit more concise, e.g. by saving DO, END, OUTPUT, and maybe STOP statements. However, all the advantage in thus attained parsimony evaporates as soon as there is something to be done before and/or after reading the whole file. Indeed, how do we do it, being limited only to the possibility of writing instruction inside the loop? Here is how the difficulty is usually circumvented:

```
Data AwaterNmiss ( Keep = State Water )  ;
  If _N_ = 1 Then Do ;
    Time = Time() ;
    Put Time = Time. ;
  End ;
  If End_of_File Then Do ;
    Average = Round ( TotalWater / NonMiss, 1 );
    Put Average = Comma. ;
    Time = Round (Time() – Time, .001 ) ;
    Put Time = ;
  End ;
  Set States End = End_of_File ;
  If State NE: 'A' or Water = . Then Delete ;
  NonMiss    ++       1 ;
  TotalWater ++ Water ;
Run ;
```

Note that above, the DELETE statement is an exact counterpart of the CONTINUE statement in that it transfers control directly to the very bottom of the loop (in this case, implicit one), whence it goes back to the top, thus bypassing the (implicit) OUTPUT and having the rather oblique meaning of 'deleting the observation'.

If this, 'conventional', style and this kind of programming logic (or lack of it thereof) suits one better, it can be of course perceived as a matter of personal preference. Keep in mind, though, that it makes the computer ask the question about _N_=1 and End_of_File=1 just as many times as there are records to read. So, if you have, as it often happens nowadays, 100,000,000 or so observations to process, you might want to rethink the 'standard' approach. Moreover, it is not accidental that when using the automatic loop, the testing for the end of file must be placed at the top of the step, contrary to the stream of consciousness. If it were located after the file processing statements, where it logically belongs, you might not see AVERAGE computed and AVERAGE and TIME printed at all! The reason is simple. Suppose the values in the last record satisfy the condition

If State NE: 'A' or Water = . Then Delete ;

If that is the case, control goes to the top of the implicit loop; from there, all instructions are executed serially until control bumps into the SET (of other I/O) statement having nothing to read, and thus the step is terminated. No instruction placed after the conditional sentence are executed if it evaluates true, which would be the fate of AVERAGE and TIME, were not they placed *before* SET.

4.3. LINK SUBROUTINES

Those familiar with GOSUB subroutine in BASIC or COBOL paragraphs will be pleased to know that this kind of functionality is fully supported in SAS DSL. You can define a block of statements, supply a statement label for its first statement, and close it with a RETURN statement:

```
CALLME: < SAS statements> RETURN ;
```

Then anyplace in the code CALLME can be invoked as follows:

```
LINK CALLME ;
```

In fact, it can be called from CALLME itself, albeit no more than 10 nested levels are allowed. When CALLME is called, control is transferred to the statement following the label and on until the entire block is performed; then control is returned (hence the RETURN verb) to the instruction immediately after LINK.

4.3.1. 'Logical'

If you decide to use the 'logical' SAS DSL programming (in the sense above), the most natural place for a subroutine is after the STOP – for it guarantees that no subroutine can be executed unless called. For example, if you wanted to combine the two SUM statements above in a LINK module ACCUM, and the statements computing and printing average – in a module MEANS, it might look like this:

```
Data AwaterNmiss ( Keep = State Water )  ;
  Time = Time() ;
  Put Time = Time. ;
  Do Until ( End_of_File = 1 ) ;
    Set States End = End_of_File ;
    If State NE: 'A' or Water = . then Continue ;
    Link ACCUM ;
    Output ;
  End ;
  Link MEANS ;
Stop ;
    ACCUM: NonMiss ++1 ; TotalWater ++ Water ;  Return ;
    MEANS: Average = Round ( TotalWater / NonMiss, 1 ) ;
              Put Average = Comma. ;
Run ;
```

In reality, RETURN closing a LINK module is only needed to separate it from other module defined right after it – otherwise they would be perceived by the compiler as one. That is why after MEANS, RETURN can be painlessly omitted, but after ACCUM, it is required.

4.3.2. 'Traditional'

It gets less attractive if the automatic loop is used. In this case, LINK module definitions must be placed after yet another RETURN, its role being no different from 'GO TO TOP' .

```
Data AwaterNmiss ( Keep = State Water )  ;
  If _N_ = 1 Then Do ;
    Time = Time() ;
    Put Time = Time. ;
  End ;
  If End_of_File Then Do ;
  Link MEANS ;
    Time = Round (Time() – Time, .001 ) ;
    Put Time = ;
  End ;
  Set States End = End_of_File ;
  If State NE: 'A' or Water = . Then Delete ;
  Link ACCUM ;
RETURN;
    ACCUM: NonMiss ++1 ; TotalWater ++ Water ;  Return ;
    MEANS: Average = Round ( TotalWater / NonMiss, 1 ) ;
              Put Average = Comma. ;
Run ;
```

Finally, a LINK module can be principally placed anywhere in the code if it is enveloped in a never-executable shell. In the case of the ACCUM routine, for example (either of the following two variants are workable):

```
If 0 Then Do;
   ACCUM: NonMiss    ++1        ;
             TotalWater ++ Water ;
End ;

Do While (0);
    ACCUM: NonMiss     ++       1 ;
              TotalWater ++ Water ;
End ;
```

Of course, if each module is defined this way, no RETURN is required *anywhere*. This method makes it possible to define LINK routines *before* they are called (analogous the SAS macro-style, for that matter). However, one should bear in mind that this kind of definition contains an extra comparison, so care should be taken to place any such definitions outside of any frequently iterating structure.

Whether this way of defining LINK modules can be seen as an advantage or not, depends on the personal programming style and one's propensity to split a program in a number of hierarchical paragraphs similar to the way COBOL programmers perceive as the 'structured programming'. (As a matter of course it has nothing to do with the structured programming whatsoever). There indeed may be situations where it may be more logical to arrange the program around several LINK modules, rather than macros, say. However, the functionality of a LINK routine is limited by the fact that it cannot be parameterized and shared all its variables with the caller.

IV. CONCLUSION

Even if you have had enough patience and courage to get here without using a GOTO CONCLUSION instruction, but rather by faithfully executing all instructions all the way from INTRODUCTION, you have only seen a shining tip of a giant SAS DATA step iceberg. Hopefully, it has given you a little bit of idea what kind of beauty dwells within.

V. REFERENCES

1. Rick Aster, Rhena Seidman. Professional SAS Programming Secrets, Wincrest/McGraw-Hill, 1991.
2. T.W. Pratt. Programming Languages. Design and Implementation, 2nd Edition, Prentice-Hall, Inc.
3. SAS Language. Reference. Version 6, 1st Edition, SAS Institute, Inc., Cary, NC.

VI. AUTHOR CONTACT INFORMATION

Paul M. Dorfman,
SAS Programmer

10023 Belle Rive Blvd. 817
Jacksonville, FL 32256

(904) 564-1931 (h)
(904) 905-5428 (o)

sashole@bellsouth.net
paul_dorfman@hotmail.com
paul.dorfman@bcbsfl.com