# SASdecoder™ 6.3 User Manual

SASdecoder™ is a utility that translates certain forms of SAS® code into Stata® dictionaries, Stata do-files, and Stat/Transfer™ schema files, facilitating the reading of raw data into various storage formats. This is useful when you are presented with SAS code for reading that raw data file, but you do not use SAS.

SASdecoder is a product of Essistant Software, LLC;
114 Hawthorne Road
Baltimore, MD 21210
www.essistantsoftware.com

For additional information or product support, or to report problems, write to
techsupport@essistant-software.com


Edition 1; October 2, 2011

# License Agreement and Limited Warranty

IMPORTANT: READ CAREFULLY BEFORE USING SASDECODER SOFTWARE.

By clicking the "I Agree" box during the installation process, or using any portion of the SOFTWARE, you indicate your acceptance of the following License Agreement.

As a USER of the SASdecoder™ SOFTWARE, you are granted the license to use it for translating SAS® System Program files, and for no other use. You, or any persons or entities in possession of the SOFTWARE, may not modify, translate, reverse-engineer, decompile, or disassemble this SOFTWARE. You may not incorporate this SOFTWARE into other products (commercial or otherwise) or create derivative works based on it. You may not sell, rent, or lease any rights to this SOFTWARE or accompanying DOCUMENTATION. You may not conduct a commercial enterprise involving the use of this SOFTWARE, other than to distribute files produced by SASdecoder that correspond to raw data files if your product is raw data.

Users are free to distribute or publish any output of SASdecoder, under the condition that the file shall retain the header lines that indicate that it is a product of SASdecoder and that mention Essistant Software – or that it includes a citation of SASdecoder and Essistant Software, LLC.

Essistant Software, LLC warrants that the SASdecoder SOFTWARE will perform in substantial compliance with the specifications set forth the in the **User Manual**, provided that it is not modified and that it is used on the hardware and operating systems for which it was designed. This warranty is limited to the 90-day period from your original purchase. If you report in writing within 90 days of purchase, a substantial defect in the SOFTWARE's performance, Essistant Software will attempt to correct it, or, at its option, issue a refund of the amount originally paid for the SOFTWARE. Essistant Software, LLC will not be held liable for consequential damages due to failure of SASdecoder to perform or to perform correctly; or any inaccuracies or shortcomings of the User Manual. Liability shall be limited to the purchase cost of the SOFTWARE.

Essistant Software, LLC does NOT warrant that the User Manual presents an accurate description of SAS functionality. The descriptions of SAS features contained herein are not intended to be instructions for using SAS.

# Contents

# Preliminary Notes

### What's New – Version 6.2 to 6.3

Version 6.3 introduces the following features:

- collapse_at_plus option
- pos1_linemove option
- string literals span lines, following SAS behavior
- comment lexing/parsing conforms to SAS behavior
- macro comments
- limited_simple_integer_expression for / + # pointercontrol
- variables in the INPUT statement need not be unique.
- ranges of existing variables, ranges of numerically-suffixed names, and predefined wildcards allowed in variable lists; a limited set of these are allowed in the INPUT statement
- grouped format lists in the INPUT statement; also, format multipliers
- corrections made to the handling of @ and + pointercontrol
- corrections made to the computation of implicit positions in the presence of format modifiers

### Updates

Users should check with Essistant Software for updates – www.essistant-software.com

### User Feedback

The author would appreciate hearing from users regarding how useful and helpful they find SASdecoder to be, what features they find most useful, and which interface they use most often. Please send your comments to techsupport@essistant-software.com , or write to this same address to report if…

- SASdecoder fails to perform, or crashes or hangs, or issues an internal error;
- you believe SASdecoder perfoms incorrectly;
- you believe there is a SAS feature that hasn't been accommodated but could and should be;
- you believe that this documentation is faulty or difficult to use or understand;
- you have any other suggestions for improvements or corrections to either the SASdecoder program or this documentation.

Additionally, the author seeks help in understanding the use of commas in grouped format lists in INPUT statements. If any users can clarify this matter, it would be greatly appreciated.

# Introduction

## Uses and Capabilities of SASdecoder

SASdecoder is a utility that can translate certain forms of SAS code into Stata dictionaries, Stata do-files, and Stat/Transfer schema files. The scope of SAS statements it can accept includes limited features of INFILE, INPUT, LABEL, PROC FORMAT and several other related statements that are used in specifying how raw data files are to be read.

SASdecoder runs on PCs under Windows. It has been tested on Windows XP, Windows Vista, and Windows 7. Its Stata do-file and dictionary output can be used on any system that has Stata installed. Its Stat/Transfer schema output can be used on any system that has Stat/Transfer installed. Thus, the output can be used on a broader variety of machines than can run SASdecoder (e.g., Macintosh and Unix-based). That is, given a successful run of SASdecoder, the output can be copied to and used on a broad variety of machines.

SASdecoder was created in response to a situation that is common among some data analysts: You are given a raw data (text) file, along with SAS code (a "SAS System Program") to read it into the SAS internal form, but you do not use SAS. It can provide a significant advantage when used on very large SAS source files – ones that are too large to translate clerically.

It is important to understand that SASdecoder does not read the data; it gives you the tools to read the data using Stata or Stat/Transfer. Furthermore, SASdecoder does not translate other SAS data-management operations such as MERGE, nor does it translate analysis procedures such as FREQ, UNIVARIATE or TABLE. (It does, however, translate VALUE statements to value label definitions.) Finally, it does not convert SAS data; users who need conversion of data should use a conversion facility such as Stat/Transfer (see www.circlesys.com).

It is important, also, to understand that, due to these limitations, SASdecoder cannot accept most SAS programs as given. You will need to edit the program down to the essential parts that are relevant for data-reading. See **More on What SASdecoder Can Do** for more on this topic.

## Stata and Stat/Transfer Output

For Stata users, it can generate a Stata dictionary, and optionally, a corresponding do-file. You must have Stata software to be able to make use of these files. See www.stata.com for more information.

For users of other data formats, it can generate a Stat/Transfer schema file, which can be used to convert the raw data into a multitude of formats (though the range of SAS input is somewhat restricted, as will be explained below). To use a Stat/Transfer schema file, you must have Stat/Transfer software, available from Circle Systems (www.circlesys.com), and your targeted data format must be among those generated by Stat/Transfer. Presently (in Version 11), Stat/Transfer supports these data formats; thus, with a valid schema file, the data can be read into any of these formats:

- 1-2-3
- Access (Windows version only)
- ASCII - Delimited
- ASCII- Fixed Format
- dBASE and compatible formats
- Data Documentation Initiative (DDI) Schemas
- Epi Info
- Excel
- FoxPro
- Gauss
- HTML Tables
- JMP
- LIMDEP
- Matlab
- Mineset
- Minitab
- MPlus
- NLOGIT
- ODBC (Windows and Mac versions only)
- OpenDocument Spreadsheets
- Paradox
- Quattro Pro
- R
- RATS
- SAS Data Files
- SAS Value Labels
- SAS Transport Files
- S-PLUS
- SPSS Data Files
- SPSS Portable
- Stata
- Statistica (Windows version only)
- SYSTAT
- Triple-S

See www.circlesys.com for more information.[1]

[1] Stat/Transfer also supports OSIRIS and SAS CPORT, but they are read-only. Data Documentation Initiative (DDI) Schemas, MPlus, OpenDocument Spreadsheets, and RATS are new additions as of version 11; they are not supported in version 10 or earlier.

# A Synopsis of What SASdecoder Can Do

The SAS INPUT statement allows these forms of input:

- column input
- formatted input
- list input
- named input

SASdecoder can handle all but the named input form.

Examples:

1, column input[2]:

```
DATA;
 INFILE "faminc01.dat";

INPUT
 ID01 1-4 FIPS_STN 5-6
 FAMINC01 9-15 TXHW01 16-22 TRHW01 23-29
 TXOFM01 30-35;

LABEL
 ID01="2001 INTERVIEW NUMBER"
 FIPS_STN="FIPS STATE NUMERIC CODE"
 FAMINC01="TOTAL FAMILY INCOME 2000"
 TXHW01="TAXABLE INCOME HEAD AND WIFE 2000"
 TRHW01="TRANSFER INCOME OF HEAD AND WIFE 2000"
 TXOFM01="TAXABLE INCOME OTHER FAMILY MEMBERS";
```

This would be translated into this Stata Dictionary:

```
dictionary using faminc01.dat {
_column( 1) int  id01 %4s "2001 INTERVIEW NUMBER"
_column( 5) byte fips_stn %2s "FIPS STATE NUMERIC CODE"
_column( 9) long faminc01 %7s "TOTAL FAMILY INCOME 2000"
_column( 16) long txhw01 %7s "TAXABLE INCOME HEAD AND WIFE 2000"
_column( 23) long trhw01 %7s "TRANSFER INCOME OF HEAD AND WIFE 2000"
_column( 30) long txofm01 %6s "TAXABLE INCOME OTHER FAMILY MEMBERS"
}
```

…or to this Stat/Transfer schema file:

```
file faminc01.dat
```

[2] This example is excerpted from the Panel Study of Income Dynamics Family Income-Plus supplement files: psidonline.isr.umich.edu .

```
variables
 id01 1-4 {2001 INTERVIEW NUMBER}
 fips_stn 5-6 {FIPS STATE NUMERIC CODE}
 faminc01 9-15 {TOTAL FAMILY INCOME 2000}
 txhw01 16-22 {TAXABLE INCOME HEAD AND WIFE 2000}
 trhw01 23-29 {TRANSFER INCOME OF HEAD AND WIFE 2000}
 txofm01 30-35 {TAXABLE INCOME OTHER FAMILY MEMBERS}
```

2, Formatted input[3]:

```
INPUT
 @1 CASEID $15.
 @18 V000  $3.
 @21 V001  8.0
 @29 V002  4.0
 @33 V003  3.0
 ;
```

This would get translated into these Stata Dictionary elements:

```
_column( 1) str15 caseid %15s
_column( 18) str3  v000 %3s
_column( 21) long  v001 %8f
_column( 29) int  v002 %4f
_column( 33) int  v003 %3f
```

…or into these Stat/Transfer schema elements:

```
Variables
  caseid 1-15 (A)
  v000 18-20 (A)
  v001 21-28
  v002 29-32
  v003 33-35
```

3, List input:

```
INPUT
 name $ age earnings;
```

gets translated into these Stata Dictionary elements:

```
 str8  name %s
 float  age %f
 float earnings %f
```

[3] Excerpted from the zwir31rt.sas file from Demographic and Health Surveys; www.measuredhs.com

…or to these Stat/Transfer elements:

```
name (A8)
age (F)
earnings (F)
```

In a typical SAS program that reads raw data, only one input form is used, but it is possible to have a mixture of the forms, in which case, the output, either a Stata dictionary or a Stat/Transfer schema, will be created with a corresponding mixture of specification types. But in the case of a StatTransfer schema, it will not be valid; a Stat/Transfer schema may contain list-input variables, provided that it is the only input type used – i.e., *all* variables are list-input. However, as noted, most real-world examples use only one input form, so this is not expected to be a serious limitation.

In typical use, the SAS code is a given entity – already written and tested by SAS programmers. It is not intended for you to write SAS code for SASdecoder to translate.[4]

## More on What SASdecoder Can Do

It is important to understand that SASdecoder accepts only a limited set of features of a small subset of SAS statements. Furthermore, there are limits to its capability to produce Stata or Stat/Transfer files that emulate the behavior of SAS. Partly, the limitations correspond to the intrinsic capability of Stata and Stat/Transfer to emulate SAS features, regarding the reading of raw data.

SASdecoder is designed to accept those SAS features that specify the reading of raw data files, and only a certain subset of these features can be translated to either a Stata dictionary or do-file or a Stat/Transfer schema. Furthermore, not every translatable SAS feature has been accommodated to date (see the **Notable Omissions** section), but much effort has been put into accommodating most of what you are likely to encounter.

Usually, a SAS program that was written for reading raw data cannot be used as-is by SASdecoder, but it still may be usable. You may need to edit- or comment-out certain parts of the file – features that SASdecoder doesn't understand, but which are not essential to the task of translation to Stata or Stat/Transfer. You may also need to rename some identifiers, such as those that are illegal as Stata variable names. It may take several iterations before this succeeds; you may want to use the "Dryrun" button or `dryrun` command (or omit the output file specifications) until you achieve a successful parsing of the SAS code. Also, you may want to do your editing in a separate copy of the SAS code file (saved under a distinct filename); you would do your editing in one file and keep the other in its original state for safekeeping and reference purposes.

Also, the output of SASdecoder may not always be precisely what you need, and you may want to adjust it as you see fit before using it. And it may need some tweaking to run satisfactorily. So you should consider it a starting point rather than a completed product.

[4] While that is not the intent, it is certainly possible to do. This might be instructive to someone with SAS skills to learn about Stata dictionaries or Stat/Transfer schemas.

Some SAS features cannot always be translated exactly into Stata or Stat/Transfer; some translate to code that is as close as possible, but may not produce exactly the same results. Sometimes this depends on the form of the raw data involved. Usually, warnings will be issued in these situations.

## Using the Output

If you are using this, then presumably you know to use a Stata dictionary and do-file, or a Stat/Transfer schema file. But, for instructions on these matters,

- Stata users should see "infile (fixed format)" in the Stata Data Management Manual, or type `help infile2` in the Stata command window. Stata users can also make use of the do-file option, which will include the appropriate `infile` command. Also, see the `do` command in the Stata Reference Manual, or type `help do` in the Stata command window.
- Stat/Transfer users should read about schema files in the Stat/Transfer documentation.

Briefly, for Stata users, once you have the dictionary, you subsequently use it in an `infile` command. Thus, assuming that the dictionary is named faminc01.dct, you might issue this command in Stata:

```
infile using faminc01
```

This is the simplest use of a dictionary; you may wish to add certain options or qualifiers as you see fit. Note that in case the dictionary does not name a raw data file, you must name it in the `using` option of the Stata `infile` command:

```
infile using faminc01, using(faminc01.dat)
```

Note, too, that SASdecoder can generate this code by using the do-file feature. Under this scenario, a do-file would be generated that contains this code; assuming that the do-file is named faminc01.do, you would issue this command in Stata:

```
do faminc01
```

Stat/Transfer users would use the "ASCII - Fixed Format (Stat/Transfer Schema)" option in the "Input File Type" specification in the "Transfer" tab on the menu. You can also use the Stat/Transfer Command Processor; specify the schema file as input to the `COPY` command. See the Stat/Transfer instructions for more information on this.

**IMPORTANT NOTE REGARDING Stat/Transfer:** The Stat/Transfer schema file (or the SAS file, prior to translation) may need some adjustment in order for Stat/Transfer to locate the raw data file. By default, Stat/Transfer expects the data file to be named *filename*.dat, where *filename* is the name of the schema (minus the .sts extension). Thus, if the schema is named jobhistory.sts, then it expects the data to be in jobhistory.dat. This can be overridden by a FILE command in the schema – which SASdecoder usually writes – but the FILE command apparently takes effect only if it includes the full path of the data file. If your data file is jobhistory.txt, and your file statement says,

```
file jobhistory.txt
```

then this file statement is not sufficient. The filename needs to be fully specified; it might read…

```
file c:\project22\testing\jobhistory.txt
```

This is true, even if your current directory is c:\project22\testing\. Note that Stat/Transfer will accept either forward- or back-slashes as directory separators; the prior statement could also be written as,

```
file c:/project22/testing/jobhistory.txt
```

Finally, note that there is a difference between the action of the Stata do-file and the Stat/Transfer schema file: the Stata do-file, as created by SASdecoder, does not include commands to `save` the dataset. It is left to the user to amend the do-file so as to save the dataset if desired, or to manually issue a `save` command. (For more on this, see the Stata `save` command in the Stata Data Management manual, or type `help save` in the Stata command window.) By contrast, Stat/Transfer naturally saves a data file when it successfully processes a schema file.

## More about the Do-File and the Stat/Transfer Schema File: Value Labels

The Stata do-file has two purposes. As mentioned earlier, it provides the `infile` command that makes use of the dictionary file. But most Stata users would know how to formulate this command, so this is not of great value. The other use of the Stata do-file is to provide value labels and the commands to assign them to variables. These value labels come from SAS VALUE statements. Thus, for example, the SAS code...

```
PROC FORMAT;
 VALUE SEXLABEL 1="Male" 2="Female";

DATA;
 INFILE "faminc01.dat";

FORMAT SEX SEXLABEL.;

INPUT
 ID01 1-4 SEX 5;
```

will result in a do-file that contains, in addition to the `infile` command, the following code to define and assign a value label:

```
#delimit ;

label def sexlabel
  1 "Male"
  2 "Female"
 ;

#delimit cr

label val sex sexlabel
```

The Stat/Transfer schema file will also include features to create value labels. Thus, this same example will generate the following Stat/Transfer schema code.

```
file faminc01.dat

variables
 id01 1-4
 sex 5 \sexlabel

value labels

 \sexlabel
  1 "Male"
  2 "Female"
```

Notice that for Stata, the data-reading specifications and the value label definitions (if any) are in separate files; for Stat/Transfer, they are all in one file.

## Sample Files

The installation package comes with a set of sample SAS file that you can try. The version 6.3 includes the following files:

- Sample01.sas – demonstrates column input
- Fam1971_short.txt – data for use with sample01
- Sample02.sas – demonstrates formatted input and value labels
- Sample03.sas – demonstrates list input and grouped format lists

Users should check the readme file sasdecoder6readme.txt (which is part of the installation) or examine the contents of the installation directory to see what sample files are actually be present.

These files should be found in the same directory as the SASdecoder software was installed, typically "c:\program files\sasdecoder6" ("c:\program files (x86)\sasdecoder6" on Windows 7).

**Important note:** you should **not** write your output to this directory when running these sample files. Instead, direct the output files to some other appropriate directory of you choosing or one which you have created for this purpose. (You should either specify the desired output directory explicitly, or control the current directory using the `cd` command or by browsing to it.) Possibly the best course of action is to copy these sample files to your chosen directory and to work entirely within that directory.

In case you should (accidentally) attempt to write your output to the installation directory under Vista or Windows 7[5], it will go to an alternative directory, which may cause confusion. This alternative directory will look like

[5] This behavior does not occur under Windows XP, but it is still a good idea to not write to or modify anything in the installation directory.

"c:\users\username\appdata\local\virtualstore\program files\sasdecoder6"

or

"c:\users\username\appdata\local\virtualstore\program files (x86)\sasdecoder6".[6]

Thus, under Vista or Windows 7, you may think you have written your output file into "c:\program files\sasdecoder6" or "c:\program files (x86)\sasdecoder6", and you will have every indication that you did, but when you go to look for it, it won't be there; it will, instead be in alternative directory shown above[7]. This information is provided to help you in case you have attempted to write a file into the installation directory; it is best to avoid the issue and use a distinct directory for testing the sample files.

Another important consideration: for Stat/Transfer users, you may need to adjust the filename in the FILE statement. Please see the section headed **IMPORTANT NOTE REGARDING Stat/Transfer** in the **Using the Output** section.

[6] The latter form is seen under Windows 7. Such a directory may get created at the time you first create a file located therein.

[7] What is nominally one directory is a pair of distinct directories: one for reading, one for writing.

# Activation and Running

## Activation

After installation, SASdecoder requires an activation step in order to enable its full functionality. Prior to activation, SASdecoder will run in "Demo Mode", which allows users to try SASdecoder – to see what it can do, and to evaluate whether to purchase an activation license. In Demo Mode, SASdecoder will omit some of the variable specifications and value label elements, but it functions the same as an activated edition in all other respects.

In Demo Mode, approximately one out of four randomly chosen variables and value label elements are omitted, and you are limited to 50 variables and 30 value label elements per data step. But you always get the first three variables in every data step and the first two value label elements per value label. Please see **Special Note about Variables that are Commented- or Edited-Out** for important cautions that apply when variables are omitted.

Every time SASdecoder starts, it checks to see whether it has been activated. If not, it presents you with the option to activate; if you decline, it will continue in Demo Mode. Within the same SASdecoder session, you can subsequently activate your SASdecoder by…

- pressing the "Activate" button on the "About" tabbed panel in the Windows Interface; or
- typing the `activate` command in the Command Interface.

You can also wait for the next time you start SASdecoder, when you will, once again, be presented with the option to activate.

Activation requires an activation code, which you should have obtained when you purchased SASdecoder. Or if you have been using a free demo copy of SASdecoder, you can purchase an activation code from your SASdecoder software source – either Circle Systems or Essistant Software; please refer to the contact information at the beginning of this document.

Activation requires an Internet connection at the time you are activating. Once your SASdecoder is activated, it should remain activated perpetually, and there is no need for an Internet connection thereafter. Users who do not have Internet access should contact their software source (Circle Systems or Essistant Software) to make other arrangements.

When you initiate the activation process, a separate screen will appear, asking for the activation code. (If you are activating from the Command Interface, you may need to "prod" your computer to make that screen appear, especially if you were running SASdecoder in full-screen mode.[8] You can switch in and out of full screen mode with alt-enter; you can switch processes by pressing the Windows key or alt-esc.)

During the activation process, you will be asked for a password – one that you are establishing at this time. Please retain this password for future reference. It is used in case you need to reactivate

[8] Full screen mode is apparently not supported in Vista and Windows 7.

your SASdecoder license at a future time – such as if you install SASdecoder on another computer, or if your operating system gets reloaded and you reinstall SASdecoder. During this process, you can also optionally provide an email address. This is useful in case you need to reactivate your software, but forgot your password; you can have it emailed to you.

Your activation code is valid for up to two activations – so you can, for example, have SASdecoder on your office and home computers under one license. Installing SASdecoder on a second computer is considered as a reactivation.

If you have trouble with the activation process, please contact techsupport@essistant-software.com for assistance.

## Running SASdecoder

There are two avenues for running SASdecoder:

- Via the Windows interface
- Via the Command interface (the "Console Interface" or "Console Application")

Either one is just as good as the other; the choice is a matter of your preference. They are two interfaces to the same underlying algorithms for reading SAS code and creating the translated output.

## Running SASdecoder via the Windows Interface

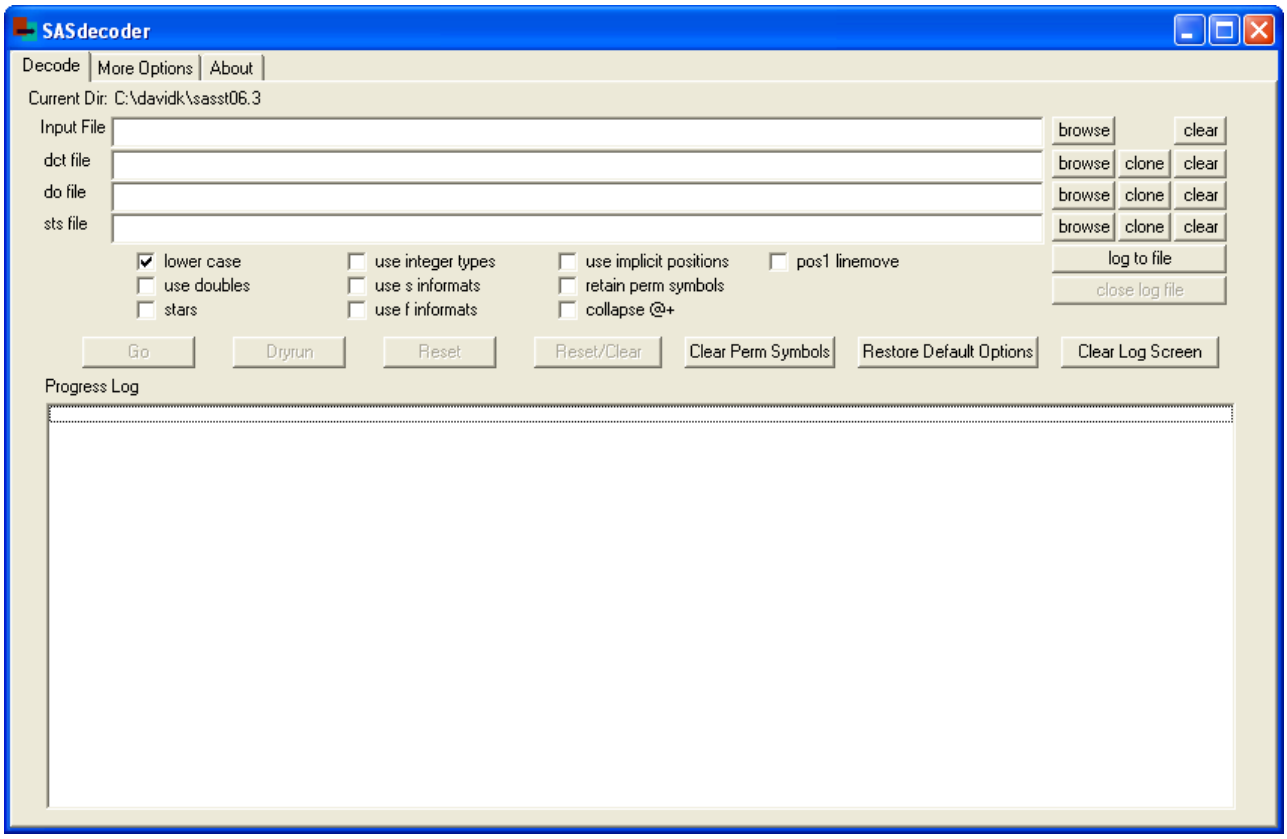
The Windows Interface can be launched by clicking the SASdecoder icon in the SASdecoder6 submenu of the system's Program Menu, or from a SASdecoder icon on the desktop

The SASdecoder Windows Interface has three tabbed panels with these labels:

- Decode
- More Options
- About

Most of the action will occur in the Decode panel. The More Options panel provides control of some rarely-needed options which can be ignored in most uses. The About panel reports some information about your edition of SASdecoder and its activation status. It also provides the option to activate, if your SASdecoder software is not already activated. (See the section on **Activating SASdecoder** for more on this subject.)

The Decode Panel appears as follows:




The Decode panel has four edit boxes for file names:

- Input File (i.e., the SAS file)
- dct file (the Stata dictionary file)
- do file (the Stata do file)
- sts file (Stat/Transfer schema file)

The Input File is the existing SAS file that SASdecoder is to read. The other files are those that are to be created: the Stata dictionary file, the Stata do file and the Stat/Transfer schema file. The Input (SAS) File is opened for reading when you press the Go button; the others are created when you press the Go button. The latter three files are all optional; SASdecoder will write whichever of these you specify – or none if you don't specify any of them.

For any of these file specifications, you can type them directly into the edit boxes, or you can browse for them by pressing the "browse" buttons to the right. For the output files, there are also "clone" buttons to the right, which invoke clone-source-file-name actions. This can help save you from having to retype a lot of text, assuming that your output names will be patterned after the input file name. E.g., if your input file is c:\bigproject\abc.sas, the "clone" buttons will create the names c:\bigproject\abc.dct, c:\bigproject\abc.do, and c:\bigproject\abc.sts. The name-patterning operations use the existing Input File name at the time the "clone" buttons are pressed.

There are also "clear" buttons on the far right which clear the edit boxes.

Above the Input File edit box is a line that displays the current directory. This is useful information if you type a file name into an edit box without its full name qualifications, e.g., abc, rather than c:\bigproject\abc. Note that you can change the current directory by browsing into the desired directory and selecting a file. Also note that the typical initial directory is the place where SASdecoder is installed, such as "c:\program files\sasdecoder6", which you should not use for you work.[9]

Further down on the right side are the "log to file" and "close log" buttons. The "log to file" button opens a file that will record whatever is written to the Progress Log screen. It opens a dialog box to select the file name; the default file extension is .log. If the file already exists, you have the option to overwrite it or append it. Also note that the log file is opened at the time you specify it, whereas, the others are opened when you press the Go button. The "close log" button closes the log file.

Filename will generally be appended with corresponding extensions (.sas, .dct, .do, .sts, and .log) if those extensions are not explicitly stated. (This action occurs at the time the files are opened.) If any of the filenames need to be taken "as-is", that is, without the extension, simply append a period to the name. The period will not used in the file name, but it signals that you want the rest of the name taken as-is.[10]

Below the filename edit boxes are a set of checkboxes for various options. These are, by default, unchecked (i.e. off), except for lower case.

- lower case – writes variable names in lower case; default is on, i.e., it is pre-checked. If this is unchecked, then all variable names will be written in upper case.
- use doubles – (Stata only) uses double, rather that float, as the general numeric type.
- stars – comments-out all variable specifications. Please see **Special Note about Variables that are Commented- or Edited-Out** for more information on this matter.
- use integer types – (Stata only) uses integer types (byte, int, long) for all numeric variables that are read from fixed-width fields with no implied decimals.
- use s informats – (Stata only) uses the %s %infmt for numeric variables with no implied decimals.
- use f informats – (Stata only) uses the %f %infmt for character variables.
- use implicit positions – calculates and uses implicit positions where possible, for variables without a specific starting position.
- retain perm symbols – specifies that permanent symbols (value labels, filename, libname) are retained between parses.
- collapse @+ – specifies that a sequence of `@` and + pointercontrol elements will be converted to a single `@`; thus `@7 +3` will be taken as `@10`. Control of this option is relevant for Stata only; it is always on for Stat/Transfer.
- pos1 Linemove – specifies that whenever a change of line occurs as a result of # or /, pointercontrol then a start position of 1 is imposed.

On the More Options tabbed panel there are also these options:

[9] In Vista and Windows 7, you will be prohibited from working there. More on this under **Sample Files**.
[10] Any combination of one or more trailing spaces or periods will be stripped; they are illegal at the ends of filenames in Windows.

- nostop – (Stata only); specifies that SASdecoder will not stop writing the .dct file if internal errors occur.
- lookupstr method, with a set of radio buttons labeled “none”, “exact”, “partial” and “full” – sets the lookupstr value; see the section headed **Options** for an explanation. The default/recommended value is “exact”.

Please refer to the **Options** section for more details on these options.

Returning to the Decode panel, below the options is a set of buttons that invoke actions:

- Go – parses the SAS (Input) file; potentially writes .dct, .do, and/or .sts files, if they were specified.
- Dryrun – parses the SAS (Input) file; does not write any files.
- Reset – resets the Decode window to enable another parse; retains file names.
- Reset/Clear – resets the Decode window to enable another parse; clears all file names except for the log file.
- Clear Perm Symbols – removes any permanent symbols (value labels, filename, libname) that may be remaining from a prior parse; this is relevant if you choose the retain perm symbols option.
- Restore Default Options – restores all options to their default values. This applies to the options on the More Options tabbed panel as well as the present (Decode) panel.
- Clear Log Screen – clears the Progress Log list box. Note that this has no effect on the copying of the Progress Log screen to a text file

Finally at the bottom there is the Progress Log list box, where various messages are displayed, reporting the actions completed and any error or warning messages generated in the parse. As noted above, you can have this saved to a file by using the “log to file” button. You can also select lines using standard mouse and keyboard actions; then press ctrl-C or ctrl-Insert to copy the selected lines to the System Clipboard, and subsequently paste it into another Windows application.

To use SASdecoder, you would specify (or browse for) a SAS file in the Input File edit box. You can then specify Stata dictionary and/or do files, or a Stat/Transfer schema file. (You could also specify Stata and Stat/Transfer files together, if you want.) You then can press Go; the output files will be created, and, if the SAS file is successfully parsed, the output files will be filled in.

In actual use, you are likely to encounter some errors in the first attempts at parsing a file, and you may need to alternately edit the input file and parse it until the errors are cleared up. In this scenario, it is appropriate to use Dryrun rather than Go, until the errors are cleared. The Dryrun button is equivalent to the Go button, but ignores any output files than may be specified; SASdecoder will still attempt to read the SAS file, but it won’t create any output. However, these file specifications are “waiting” for when you press Go. Alternatively, you could omit the file specifications until you get a successful parse. But the Dryrun facility allows you to set up the output specifications at the start. You then switch to Go, once you are getting an error-free parse.

The editing would need to be done in a separate window, using an editor of your choosing, as there is presently no built-in provision for editing. You can keep both the editor and SASdecoder processes active concurrently and switch between the two. Each time you press Go or Dryrun, the

latest saved edition of the Input File is opened, so this method will work as long as you save the file (from the editor) before switching to SASdecoder and pressing Go or Dryrun.

When creating the output files, if any of them already exist, you will be queried as to whether to overwrite them, or cancel the operation.

## Running SASdecoder via the Command Interface

The Command-Interface (or Console) version of SASdecoder runs in a Command Prompt window (or in full-screen mode, if you prefer, and if that option is available on your system). This allows users to invoke SASdecoder features by typing commands, which may be of value to some users who prefer that mode of controlling an application.

The Command-Interface can be launched by clicking the SASdecoder Command Interface icon in the SASdecoder6 submenu of the system's Program Menu. Or, you can run sasdecoder.exe from an existing Command Prompt window.

The Console Interface accepts a set of commands that tell SASdecoder what file to read, what files to produce, and what options to use in the process of doing its work – much the same as the Windows Interface, except that it is command-driven. The basic process is to specify the files and options; then give it the `go` or `dryrun` command to initiate a parse of the SAS file – equivalent to the go and dryrun buttons on the Windows Interface. You can then make changes and do another parse, or you can end the session by typing `exit`.

The commands are listed below. Entries in [square brackets] contain a list of items separated by the vertical bar (|), representing a set of items from which you can choose one. The square brackets and vertical bar are not to be typed. Where such a list occurs as the first syntactic element, it indicates command synonyms; where it appears as an option name, it indicates option synonyms. Items in *italic* font represent entities that are to be substituted by an actual value.

Some commands are for specifying filenames. Filenames can be quoted, and they must be quoted if they contain spaces or other specific characters not "traditionally" used in filenames.

Filenames will generally be appended with corresponding extensions (.sas, .dct, .do, .sts, and .log) if those extensions are not explicitly stated. (This action occurs when the files are opened.) If any of the filenames need to be taken "as-is", that is, without the extension, simply append a period to the name. The period will not be used, but it signals that you want the rest of the name taken as-is.[11]

Commands are case-sensitive; they are all in lower case.

Note that the edit keys – Insert, Delete, Home, End, Page Up, Page Dn, Backspace, F2, F3, F4, F7, F8, F9 – are available for recalling and editing commands, just as in the system Command Prompt; the list of recallable commands is separate from that of the Command Prompt (if you invoked the SASdecoder Command Interface from the Command Prompt).

[11] Any combination of one or more trailing spaces or periods will be stripped; they are illegal at the ends of filenames in Windows.

- `exit` – ends/exits the program. If you launched the Command Interface by clicking the shortcut, then the application will close.
- [`help` | `?`] – displays a brief help sequence.
- `log using` *logfilename* – opens a file to record the commands and resulting messages; essentially it copies what appears on your screen.[12] If the file already exists, you have the option to overwrite it or append it. Also note that the log file is opened at the time you specify it, whereas, the others are opened when you give the `go` command.
- `log [close cl]` – closes the log file if it is open.
- `log` – (no arguments) reports the currently open log file name.
- [`sasfile` | `sas`] *sasfilename* – specifies the SAS file.
- [`dctfile` | `dct`] *dctfilename* – specifies the dct (Stata dictionary) file.
- [`dofile` | `do`] *dofilename* – specifies the do (Stata do) file.
- [`stsfile` | `sts`] *stsfilename* – specifies the sts (Stat/Transfer schema) file.

Note: if any of these filename commands are given without an argument, then the existing value of the corresponding filename is reported. Also, for `dctfile`, `dofile`, `stsfile`, and `log using`, an argument of an asterisk has special meaning: it clones the sasfile name. Thus, for example, if the sasfile name is `abc.sas`, and you type `sts *`, then you get `abc.sts` as the stsfile name.[13]

The sasfile is opened when you issue the `go` or `dryrun` commands. The dctfile, dofile, and stsfile are created when you issue the `go` command. The latter three files are all optional; SASdecoder will write whichever of these you specify – or none if you don't specify any of them. The log is opened when you issue the `log using` command.

- `go` – parses the SAS file named by the sasfile command; potentially writes .dct, .do, and/or .sts files, if they were specified.
- `dryrun` – parses the SAS file named by the sasfile command; does not write any files.
- `clear` – clears all the filenames except the log file. To clear a single filename, give the corresponding command with a quoted null string argument, as in `dctfile ""`
- `files` – reports the names of the files you have specified, except the log file.
- `about` – reports information about the SASdecoder program.
- `activate` – invokes the license activation process. (See the section on **Activating SASdecoder** for more on this subject.)
- [`clear_perm` | `clearperm`] – removes any permanent symbols (value labels, filename, libname) that may be remaining from a prior parse; this is relevant if you choose the retain_perm option.
- `cd` [*directoryspecification*] – sets the current directory to *directoryspecification*, provided that the specified directory exists. Without an argument, it reports the current directory. For *directoryspecification*, the usual symbols for relative directories apply: `..` signifies the parent directory; `..\xyz` signifies a lateral move to directory xyz, assuming that it exists at

[12] The results of `help` and `activate` are omitted.
[13] This works if you have already declared a SAS file. Note that you can subsequently change the SAS file name, in which case the cloned name may no longer be appropriate.

the same level as the initial directory; `abc` moves to the abc subdirectory of the initial directory, assuming that it exists; `abc\xyz` moves down two levels to the xyz subdirectory of the abc subdirectory of the initial directory, assuming that they both exist. (Here, the "initial" directory refers to the directory that was in effect prior to issuing this command.) Essentially this is the `cd` of the system's Command Prompt, except that *directoryspecification* must be quoted if it contains spaces. Type `help cd` in the Command Prompt for more information. If the Command Prompt process remains open after SASdecoder exits, then, upon exit, the initial directory is restored.

- `over` [*directoryspecification*] – equivalent to `cd ..\`*directoryspecification* – makes a lateral move in the directory structure, provided that the specified directory exists. Without an argument, it reports the current directory. See the `cd` command for more details.
- `up` – equivalent to `cd ..` – makes an "upward" move in the directory structure. Takes no argument. See the `cd` command for more details.
- `option` *optionname optionvalue* – sets various options, as explained below.

Following are the options. This is a brief description of the options; they will be discussed in more detail in the **Options** section. Except as noted, all off/on options have a default value of off.

- `option` – (with no arguments) reports all option settings.
- `option` *optionname* – (with no *optionvalue*) reports the setting of the named option.
- `option default` – restores options to their default values.
- `option lower` [`off`|`on`] – sets the lower-case option; writes variable names in lower case; default: on. If this is set to off, then all variable names are written in upper case.
- `option doubles` [`off`|`on`] – sets the doubles datatype option (Stata only); uses double, rather that float, as the general numeric type.
- `option stars` [`off`|`on`] – sets the stars option; comments-out all variable specifications. Please see **Special Note about Variables that are Commented- or Edited-Out** for more information on this matter.
- `option inttypes` [`off`|`on`] – sets the inttype option (Stata only); uses integer types (byte, int, long) for all numeric variables that are read from fixed-width fields with no implied decimals.
- `option s_infmt` [`off`|`on`] – sets the s_infmt option (Stata only); uses the %s %infmt for numeric variables with no implied decimals.
- `option f_infmt` [`off`|`on`] – sets the f_infmt option (Stata only); uses the %f %infmt for character variables.
- `option implicit` [`off`|`on`] – sets the implicit-positions option; calculates and uses implicit positions where possible, for variables without a specific starting position.
- `option nostop` [`off`|`on`] – sets the nostop option (Stata only); specifies that SASdecoder will not stop writing the .dct file if internal errors occur.
- `option lookstr` [`none`|`exact`|`partial`|`full`] – sets the lookupstr value; see the section headed **Options** for an explanation. Default/recommended value is `exact`.
- `option more` *integer_value* – sets the more option; specifies how many lines are written to the screen before pausing. Default= 24. Setting it to 0 will turn off pausing.

- `option [retain|retain_perm] [off|on]` – sets the retain_perm option; specifies that permanent symbols (value labels, filename, libname) are retained between parses.
- `option [collapse_at_plus|collapse@+] [off|on]` – (Stata only) specifies that a sequence of `@` and + pointercontrol elements will be converted to a single `@`; thus `@7 +3` will be taken as @10. Control of this option is relevant for Stata only; it is always on for Stat/Transfer.
- `option [pos1_on_linemove|pos1_linemove] [off|on]` – specifies that whenever a change of line occurs as a result of # or / pointercontrol, then a start position of 1 is imposed.

The SASdecoder Command Interface displays the prompt "–>" to indicate that it is ready to accept a command. An example of a basic operation would be…

```
-> sas famdata
-> dct famdata
-> do famdata
-> go
-> exit
```

This sequence of commands would read famdata.sas, and, assuming no errors were encountered, would cause famdata.dct and famdata.do to be filled in with the translated output.

If you are a Stat/Transfer user, the basic operation would be…

```
-> sas famdata
-> sts famdata
-> go
-> exit
```

…and famdata.sts would be filled in with the corresponding translated output.

(You could also write Stata and Stat/Transfer output together, if you want.)

In actual use, you are likely to encounter some errors in the first attempts at parsing a file, and you may need to alternately edit the input file and parse it until the errors are cleared up. In this scenario, it is appropriate to use `dryrun` rather than `go`, until the errors are cleared. The `dryrun` command is equivalent to the `go` command, but ignores any output files than may be specified; SASdecoder will still attempt to read the SAS file, but it won't create any output. However, these file specifications are "waiting" for when you type `go`. Alternatively, you could omit the file specifications until you get a successful parse. But the `dryrun` facility allows you to set up the output specifications at the start. You then switch to `go`, once you are getting an error-free parse.

The editing would need to be done in a separate window, using an editor of your choosing, as there is presently no built-in provision for editing.[14] You can keep both the editor and SASdecoder

[14] Editing in a separate window is a natural consequence of using a Windows-based editor. You can also use a command-prompt-based editor in a separate Command Prompt window.

processes active concurrently and switch between the two. Each time you issue the `go` or `dryrun` command, the latest saved edition of the Input File is opened, so this method will work as long as you save the file within the editor before switching to SASdecoder and issuing `go` or `dryrun`.

(If you launched SASdecoder within an existing Command Prompt window, you could `exit` SASdecoder, edit the file in that same window, restart SASdecoder, and continue the process. However, this is not recommended, as it is slow and cumbersome, and would obliterate any choices of filenames and options that you have made.)

When creating the output files, if any of them already exist, you will be queried as to whether to overwrite them – or possibly append them, or cancel the operation.

As is true regarding filenames in general, you may include a directory specification, and you must include it if the file is not in the current directory. Thus, you could specify,

```
-> sas c:\bobsstuff\somesasfile
```

If you don't give a fully-specified file name (such as `somesasfile`, rather than `c:\bobsstuff\somesasfile`), then you need to be aware of the current directory, which you can see (or change) via the `cd` command. Also note that the typical initial directory is the place where SASdecoder is installed, such as "c:\program files\sasdecoder6", which you should not use for you work.[15]

## Options

Following are detailed descriptions of the options common to both the Command and Windows interfaces. These are controlled by the `option` command in the Command Interface and by various checkboxes, or radio buttons in the Windows Interface.

`option lower on` or the "lower case" checkbox: renders variable names in lower case. This is the default (and is pre-checked in the Windows interface).

Note that, since SAS variable names are case-insensitive, the rendering of names as either all-lower-case or all-upper-case does not present the possibility of collapsing distinct names.

If this option is deselected (`option lower off` or uncheck the "lower case" checkbox), then variable names will be rendered in upper case. SASdecoder does not presently have the capability of retaining the original case of variable names.

`option doubles on` or the "use doubles" checkbox: (Stata only) uses double rather than float for floating-point variables. Note that most numeric variables are potentially floating point, unless you specify `option inttypes on` or the "use integer types" checkbox.

`option inttypes on` or the "use integer types" checkbox: (Stata only) uses integer types (byte, int, or long) for numeric variables that are specified as fixed-width with no implied decimals.

[15] In Vista and Windows 7, you will be prohibited from working there. More on this under **Sample Files**.

This may or may not be appropriate, depending on the content of your data; you need to know the nature of your data before deciding whether this option is appropriate. It should be used only if you are certain that *all* numeric variables that are not specified as having implied decimals do not have any non-integer values in the data. Note that a decimal point in the data will override the no-decimals specification and generate a floating-point value. But such a value will be truncated if it the data type of the variable is integer. Thus, indiscriminate use of this option can be hazardous, and it is safer to not use this option – to use float or double, and to later `compress` the data in Stata. Also note that if the width is 9 or more, then the type will always be double, regardless of other considerations. This does not affect list-input variables. See more on this under **Content of the Output File**.

`option stars on` or the "stars" checkbox: comments-out all variable declaration lines in the .dct or .sts file. (The name is Stata-oriented, as it comes from the fact that it places asterisks at the start of each variable declaration line in the Stata dictionary. But this option also applies to Stat/Transfer schemas as well, using the "//" comment marker.)

This option is useful if there are many variables declared in the SAS INPUT statement, but you wish to read-in only a small subset of them. You can use this option, and then subsequently edit the dictionary or schema file and uncomment the lines for the desired variables.

An alternative method is to not use this option, but to cut and paste the desired lines from the output file into a separate file. This way, instead of using the dictionary or schema file directly, you keep it as a repository of all possible variables. Your functional dictionary or schema is another, smaller file, and you copy and paste the desired items from the repository into the functional dictionary or schema. (You would need to include any line-movement specifications that exist in the output file.)

IMPORTANT: Whether you are using the stars option or pasting from a full file to a sparse one, some cautions apply. These methods are appropriate for fixed-position variable specifications. When applied to list-input variables or formatted-input variables without pointer control, care must be taken to assure that the specifications apply to the correct field locations. Please see **Special Note about Variables that are Commented- or Edited-Out** for more information on this matter.

`option f_infmt on` or the "use f informats" checkbox (Stata only): uses the `%`*w*`f` infmt (rather than `%`*w*`s`) for character variables.

`option s_infmt on` or the "use s informats" checkbox (Stata only): uses the `%`*w*`s` infmt (rather than `%`*w*`f`) for numeric variables with no implied decimals.

The `f_infmt` and `s_infmt` options (or the "use f informats" or "use s informats" checkboxes) seem to specify the use of the "wrong" or "other" infmt type: `%`*w*`f` for character strings and `%`*w*`s` for numbers. (Here, *w* stands for a field width.) This may seem odd, but it is possible and sometimes desirable to use these infmts, especially `%s` for numbers; it actually may be common practice. See "String formats" under "infile (fixed format)" in the Stata Reference Manual for more on this matter.

`option implicit on` or the "use implicit positions" checkbox: specifies the use of calculated implicit starting positions. An implicit position is a situation where there is list input or formatted

input with no specified starting position, but the preceding variable has a fixed width and a fixed starting position (either explicitly, or implicitly). For example, in this input specification,

```
INPUT @3 v1 4.0 v2 6.0 v3 v4;
```

v1 is read from positions 3-6. v2 has no explicit starting position, but since v1 ends at position 6, v2 implicitly starts at position 7. Since v2 has a fixed width (6), it ends at position 12, and v3 implicitly starts at position 13. But since v3 does not have a fixed width (is list input), v4 does not have a definite starting position.

Also note that every INPUT statement that does not follow a hold-the-line specification (a trailing @), implicitly starts at position 1. Thus, in

```
INPUT v5 4.0 v6;
INPUT v7 6.0 v8;
```

v5 will be read from position 1 (assuming that this is the first input item, or that is not preceded by a trailing @) and v7 will be read from position 1, but on the next line of input; v6 is implicitly at position 5 and v8 is implicitly at position 7 (on different lines) . Furthermore, a line-movement specification, e.g., #2, also starts at position 1. See the section **Input Specifications for Pointer Control** for an explanation of these features.

This option will calculate implicit positions and assign them as if they were explicit. Thus, in the examples above, v2, v3, v6 and v8 will be assigned starting positions of 7, 13, 5 and 7, respectively. This can have two potential benefits: 1, as mentioned regarding the stars option (or the "stars" checkbox), this can reduce the possibility of a list-input variable being misread when some preceding variables are commented out; and 2, it can potentially make more variables usable in the .sts file. (That is, the .sts file is not valid if it has list input mixed with fixed-position variables; this option can convert some variables into fixed-position, potentially making the schema usable if was otherwise unusable.)

option [retain|retain_perm] on or the "retain perm symbols" checkbox: specifies that permanent symbols will persist from one invocation of the parser to the next . (By an invocation of the parser, we mean an instance of pressing the "Go" or "Dryrun" buttons, or issuing the go or dryrun commands.) Permanent symbols include user-defined informats (corresponding to value labels), libnames and filenames. The default action is to clear these symbols after every invocation of the parser. If this option is checked or turned on, then these symbols will be retained, rather than cleared. Thus, a set of value labels defined in one file could be used when parsing a subsequent file. If this option is checked or turned on, and, after parsing one or more files, you decide that you want to clear the permanent symbols before the next parse, then press the "clear perm symbols" button on the Decode panel or issue the command clear_perm or clearperm. (Unchecking the "retain perm symbols" checkbox or turning off this option will not clear the permanent symbols; it will only set the behavior from that point forward, and the permanent symbols will be cleared after the next parse.)

If you select this option and rerun the same SAS file, you may get messages indicating that some value elements are already defined. This is just a warning and is to be expected under the circumstances.

See the section on **Scope of Variables; Permanent vs. Local Symbols** for more on this matter.

`option [collapse_at_plus | collapse@+] on` or the "collapse @+" checkbox: specifies that input constructs of @ followed by + will be collapsed to a single @; thus @3 +5 is taken as equivalent to @8. With this option off, this example would yield Stata dictionary code of `_column(3) _skip(5)`; with this option on, the result is `_column(8)`. Control of this option is relevant for the Stata dictionary only; it is always in effect for the Stat/Transfer schema. See more on this under the **Input Specification – in Detail** section.

`option [pos1_on_linemove | pos1_linemove] on` or the "pos1 linemove" checkbox: specifies that whenever a change of line occurs as a result of # or /pointercontrol, and there is no specification of an explicit starting position, then a start position of 1 is inserted in the output. Note that when a line is specified by means of #, or a change of line is specified by / (and, in either case, there is no explicit starting position specified), then SAS will start reading from that line at position 1. The corresponding Stata or Stat/Transfer code, without any position specification, will also result in reading data starting from position 1, provided it is not a revisited line (see below). Thus, in these cases, it is not necessary for the output to explicitly include a position specification. This option makes that position specification explicit, inserting a position specification of 1 – so you can be more certain and clear about what is meant. But note that, for Stat/Transfer, the insertion of an explicit position can be a disadvantage if the rest of the items are list input; you will get a mixture of list input and specified-position items, which is not supported. Note, further, that when a line is "revisited" (see below), then SASdecoder will always insert an explicit starting location of 1; this is necessary because, under this scenario, SAS will read from position 1, whereas, Stata, in the absence of a `_column()` specification, or Stat/Transfer (due to how multiple lines are rendered), will proceed to read a revisited line from where it previously left off. A revisited line is one where the input specification causes a return to a line that had been previously read from, as in…

```
INPUT
#1 a #2 b #1 c;
```

In the above example, `c` constitutes a revisit to line 1. Also consider the following:

```
INPUT
a #1 b;
```

Assuming that `a` is on line 1, `b` constitutes a revisit to line 1, even though it is not a change of line; it is a "move", but not a change. Also, an additional INPUT statement can result in a revisit, as in the following:

```
INPUT
#2 a #1 b;
INPUT
c;
```

Here, `c` constitutes a revisit to line 2.

Finally, note that if an INPUT statement results in the first variable being read from a "fresh" line – one not yet encountered – then SAS reads from position 1, and so do Stata, and Stat/Transfer (in the absence of explicit position specifications). There is no need for SASdecoder to insert a position specification in these cases.

To recap, these considerations apply to list input only – variables that have no specific starting position. There are three classes to be considered:

- Class 1: An INPUT statement goes to a fresh line.
- Class 2: There is line movement resulting from the appearance of `#` or `/`, and it is not a revisit of a line;
- Class 3: There is line movement via any of the possible methods, and it is a revisit to that line.

In the translated output, Class 1 does not need and does not get an explicit starting position. Class 2 does not need an explicit starting position, however, one might want one anyway; this option turns on the insertion of explicit start positions. Class 3 needs and always gets an explicit starting position.

**Rarely-needed Options**

The following options should rarely, if ever, be needed by the general user. In the Windows Interface, these are located on the More Options tabbed panel.

`option lookstr [none|exact|partial|full]` or the "lookstr method" radio group (with the "none", "exact", "partial" and "full" radio buttons): invokes varying methods of saving space by allowing common textual values to share storage space. `exact` is the default and is appropriate for most uses. It would be rare to need to specify this option, as the default is suitable and adequate for most uses. But choosing one of these (particularly `partial` or `full`) may prevent a string-table-overflow error. Note that the textual values of interest include most textual values recorded while parsing the SAS code: variable names, variable labels, and many other entities.[16]

These options constitute a space/speed tradeoff. At one end of the scale, `none`, there is no space sharing, but it is the fastest; at the other end, `full`, there is maximal space sharing, but can take significant additional time, especially with large amounts of stored textual data. The others are somewhere in between these extremes. `none` specifies that no space-saving efforts are to be undertaken; every character string value is necessarily given a separate entry in the string table. `exact` specifies that only exact matches are sought. (Actually, it would take effect only for character string values in distinct namespaces, for example variables and value labels.) `partial` specifies that a newly encountered character string value can share space with an existing one,

[16] This excludes certain other textual values (value labels, filename, libname) that go in the permanent symbol table. See the section on **Scope of Variables; Permanent vs. Local Symbols** for more on this.

possibly as an initial substring. For example, “pen” can share space with “pencil”, if “pencil” was encountered first. `full` specifies that a newly encountered character string can share space anywhere in the existing stored characters. This has the greatest capacity for saving space, but is extremely time-inefficient, especially for large volumes of textual data (i.e., from large input files). (Its time usage is proportional to the square of the number of characters presently stored.)

Note that these space-sharing methods only work if there are character string values in common, which is data-dependent. They may save little space in actual practice, but may enable you to get past a string-table-overflow problem.

`option nostop on` or the “nostop” checkbox: tells SASdecoder to not stop writing the dictionary file if internal errors should occur. The default is to stop after the first internal error. This may enable you to obtain results where you otherwise might not, but the results should be used with caution, as they may be partly based on invalid or incomplete data.

## Special Note about Variables that are Commented- or Edited-Out

IMPORTANT: Whenever variable specifications in a dictionary or schema are omitted (as in Demo Mode) or commented out (as in the stars option), or cut and pasted, certain cautions apply. Generally, it is safe to do so for fixed-position variable specifications, though, even then, you must be sure to be referring to the correct line in multi-line input (i.e., when there are line-movement specifications). For input that does not specify fixed starting positions (list or formatted input without pointer control), you need to be sure that the correct starting position will be used.

Suppose you have used the stars option. It would be invalid to un-comment a variable if it does not have an explicit starting position (by column input or by pointer controls) or does not follow an uncommented variable that starts at a specific position[17]. That is, if a variable does not have an explicit starting position, then it must be preceded by a chain of one or more variables, none of which are commented out or otherwise omitted, that starts at a specific position or starts at the start of the data line as list input. (They don’t all need to be in a specific position; you just need one to start the chain.) If these conditions are not adhered to, then such a variable would likely be given an erroneous starting position (either fixed or not) because of a gap in the sequence of other variables that come before it. Commenting-out one or more of those preceding variables interrupts the sequence that would otherwise determine its proper start position.

Note that this problem can also occur if you manually comment-out some of the variable specifications, or if some are omitted due to Demo Mode. Furthermore, this principle extends to the “line number” for data formatted in multiple lines per observation. That is, you should not comment-out the line-movement specifications. (And you must keep variables within the section that pertain to their correct line numbers.)

[17] A specific position could be due to column input or pointer controls, or it may be at the start of an INPUT statement (with no preceding line-hold markers (trailing `@`)) or the first variable after line-movement controls; the latter two configurations are implicitly at position 1.

Note that the use of `option implicit on` or the "use implicit positions" checkbox can increase the number of variables that have specific starting positions, thereby possibly reducing the set of variables that are vulnerable to this problem.

# Fundamental Data-Reading Concepts

Before further describing the features of SASdecoder and the applicable features of SAS, Stata, and Stat/Transfer, it will be useful to briefly spell out some of the fundamental concepts of data and the reading of raw data.

## Raw Data vs. Internal Format

Data-handling and analysis facilities such as SAS and Stata typically hold data in their own proprietary "internal" formats. On the other hand, data sets are often delivered in textual "raw" form – in text files using printable character representations of all values. "Delivered" may mean the initial electronic form after the data-collection process, or it may be a form used to store and transmit data between applications. The process of reading raw data is that of converting from the textual form to a particular proprietary internal format. For some facilities, such as Stata, there is an in-memory data-storage scheme and a corresponding proprietary file format. In this scenario, the process of reading raw data can be viewed as "loading" or "populating" the internal data storage scheme with values found in the raw data file. You can subsequently write the data to a file, as you would by using the `save` command in Stata. Another scenario is where the values in the raw data file are read and then written directly to file in a proprietary format; this latter method is what Stat/Transfer does.

## Relational Tables

The internal format usually holds data organized in a relational table, that is, a rectangular arrangement of values. In abstract terms, it is a set of tuples, where a tuple is an ordered list of values, and where each tuple has the same form: each tuple has the same number of elements (values), and the corresponding elements in each tuple share a defined meaning in terms of what they represent. Thus, the tuples can be regarded as repeated instances of a fixed set of identifiable quantities; each of these quantities can therefore be given a distinct name. With this in mind, each tuple can be seen as a named list of values, rather than an ordered list; the order of the elements becomes an irrelevant low-level detail. These identifiable quantities are known as the variables of the table. Conversely, each variable is a set of values, one for each tuple. The whole set of values can be viewed as a rectangular array; typically, the tuples are regarded as the rows, and the variables are regarded as the columns.

Typically, each variable has a fixed data type; more on this later.

In data-analysis parlance, each tuple is an "observation" or a "case" or a "record"[18]. Each tuple typically represents some real-world entity – the "unit of observation" of the table. Each tuple contains particular values for each variable, corresponding to specific attributes of the entity that the tuple represents.

[18] "Record", particularly "physical record" sometimes refers to a line of the raw data file.

## The Raw Data File; Fields

A raw data file is typically a text file, that is, it is a sequence of textual character divided into lines (though other arrangements are possible[19]). More precisely, a text file is a sequence of lines, where a line is a sequence of characters.

In the raw data file, each value occupies a contiguous segment of characters, known as a field. It is almost universal that a field is located within a line; it does not cross line boundaries. The number of characters in the field is the "field width" (though "field length" would also be appropriate). For each variable, there are multiple instances of fields holding its values – one for each observation. It is important to make the distinction between field (in the raw data) and the corresponding value as stored in the internal form, or even the variable. Sometimes, "field" is used to refer to any of these entities, but in this document, we will use "field" only for the segments of the raw data file.

The raw data file is usually set up such that either...

- each line corresponds to one data observation, or
- every *m* lines (where *m* is an integer constant) corresponds to one data observation.

However, other possibilities exist.

For any given variable, its fields may have either a fixed or variable starting position within the lines, and either a fixed or variable width. That is, the starting position and width may be the same for each observation, or they may vary. Typically, a raw data file is set up so that it has either fixed-position fields or variable-position (and variable-width) fields for all its variables, but it is actually possible to mix the two, though that would be unusual.

With variable-width fields, the end of the field is determined by the content of the raw data, typically, by the presence of a space or other designated character. And often (though not always), the start of the next field is the first non-space character after that; thus, variable width usually implies variable positions. More precisely, if there is a variable-width field, then any following field on the same line, without absolute pointer control, will have a variable starting position. This variable-width/variable-position form of raw-data placement is known as "list input" in SAS terminology, or "free format" in Stata.

SAS also has the concept of "formatted input", which has fixed-width fields, though it does not, by itself, necessarily have fixed starting positions. But it is often used in combination with pointer control, thus facilitating another way of specifying fixed-position/fixed-width fields

## Storage Length

When the data are converted to the internal form, each value occupies a certain amount of storage space – the "storage length". Typically, each instance of a given variable has the same storage length, and thus we speak of storage length as an attribute of a variable.[20]

---

[19] Some older data files may appear as one long sequence of characters that are implicitly divided into fixed segments; these require the `lrecl` feature.

[20] This is the case with SAS and Stata, though it is not necessarily so in some other facilities such as Excel.

In SAS, the storage length of a variable can be set with the LENGTH or ATTRIB statement, or may be fixed at the first occurrence of that variable within the data step. This can apply to both character and numeric variables, though SASdecoder does not pay attention to the length setting of numeric variables.

It is important to distinguish between field width and storage length. To reiterate: field width refers to the raw data file; storage length refers to the internal data format. They may easily be confused, especially for character data, since there is a natural correspondence between the characters in the raw data field and those in the internal data form. (This natural correspondence does not exist for numeric variables.) In many typical usages, character variables have equal field width and storage length (if the field width is fixed). That is, it is appropriate to set the storage length equal to the field width, assuming fixed-width fields. But this is not necessarily the case in general. There can be situations where the storage length is greater, in which case the internal storage location is only partially filled, or where the field width is greater, in which case, the value is truncated (the latter part of the field value – that which can't fit into the storage location – is discarded). Potentially, both of these conditions may occur together in the same instance of reading a dataset – possibly in the same variable, though in different observations. (That could only happen with variable-length fields, i.e., list input.)

## Data Types

In SAS, Stata, and most data-handling facilities, each variable has a fixed data type – the set of values that may be stored. There are two broad classes of types: character[21] (or "character string" or "string") and numeric. Character types can hold textual values, and are distinguished by a fixed maximal length, which is usually the same as the storage length. Numeric types hold numbers. Often, numeric types can be divided into integer and floating-point, and possibly further disaggregated into types such as byte, word, etc.; SAS does not provide for these distinctions, but does allow some control over the storage length of numeric variables.

[21] In some programming contexts, "character" means a single character. In SAS usage, it seems to include character strings.

# SAS Language Features Accepted by SASdecoder – the Lexical Level

This section is intended to explain the lexical structure of SAS features that SASdecoder accepts – the ways that characters are assembled, at the lowest level, into meaningful units, plus some of the basic features of the structure of statements.

## Tokens

A SAS program is a sequence of SAS Statements; a SAS statement is a sequence of "tokens" – the smallest meaningful segments of text above the level of a single character.

There are several broad types of tokens: identifiers, character string and numeric literals, special characters, formats and informats. Some, such as the special characters (e.g., `$`, `+`, `/`) are single characters; others consist of sequences of several characters.

All of this occurs in the context of text files: files that are composed of characters and are "organized" into lines. There is no significance to the end of a line; a statement ends with a semicolon, which is not necessarily the end of a line. Thus, a statement continues, possibly spanning several lines, until the semicolon is encountered. Indeed, certain statements are typically long and are customarily spread over many lines. There is only one almost-absolute rule regarding how a statement's components must be arranged on the lines: a token must be entirely within a line; it may not be split across lines. There is one exception: character string literals can span lines. The only other considerations are aesthetics and legibility by human readers.

Practically speaking, SASdecoder has no limit on the line lengths in the source file. (There is a limit of about 2 gigabytes – practically negligible as limits go. The author is unaware of what, if any, is the limit for SAS. In any case, this should not be an issue.)

As mentioned earlier, SASdecoder, in accordance with SAS protocols, interprets identifiers – statement names, option specifiers (e.g. `LRECL`), and variable names, among others – as case-insensitive. Variable names will, by default, be rendered in lower case in the output; you can change this with `lower` option (`option lower off`), or by unchecking the "lower case" checkbox. Note that SASdecoder does not preserve the original case of variable names, but this should not be a significant issue, since SAS is case-insensitive regarding variable names. Thus, SAS code cannot have two variable names distinguished only by case.

The cases of variable labels or any other character string literals are preserved.

In the following examples and explanations, the words that have special meaning to SAS (statement names such as `INFILE`, `INPUT`, etc., as well as some options) will be written in upper-case letters, as is traditional in SAS documentation, though, as mentioned, they may be of either or of mixed case.

### Tokenizing

The term *tokenizing* refers to the lowest level of analyzing a program. It is the process of reading a stream of characters and deciding where one token ends and where the next one begins, as well as determining the class of each token. Thus, in the following example,

```
LABEL ID01="2001 INTERVIEW NUMBER";
```

The tokens are, with each one listed on a separate line,

```
LABEL
ID01
=
"2001 INTERVIEW NUMBER"
;
```

The first two tokens are identifiers; then there is an equals sign, a character string literal (quoted text), and finally a semicolon.

## Whitespace

Tokenizing is the process deciding where one token ends and where the next one begins, but the next one does not always begin immediately after the end of the present token. Often there is some "open space" between them. This may consist of one or more instances of space characters, tab characters, comments (of the `/* */` variety), and newlines. (A newline is the concept of the transition from the end of one line to the start of the next.) These characters and constructs are collectively known as "whitespace".

Generally, whitespace may appear (may be inserted) between any two consecutive tokens. There are some contexts where it is required, as between two identifiers, such as in `LABEL ID01` in the above example. In other contexts, whitespace is optional, such as between `ID01` and the equals sign in the example. Whitespace is not allowed inside a token; by definition, whitespace is what may appear *between* tokens. (Space within a character string literal, that is, within quotation marks, is part of the token and is not considered whitespace.) Where it is allowed, whitespace may be as little or as much as you care to insert.

## Comments

SASdecoder knows how to skip over comments in the SAS source code, in these four forms (believed to cover all forms used in SAS):

`*` *any_text_except_a_semicolon* `;`
`COMMENT` *any_text_except_a_semicolon* `;`
`/*` *any_text_except_an_asterisk-slash_combination* `*/`
`%*` *any_text_except_a_semicolon_token* `;`

Note that all of these forms may span lines.

The `* ;` and `COMMENT ;` forms can go wherever a statement is permitted; they function as statements, and the second form is known as a COMMENT statement. The two are almost equivalent – *almost* equivalent because there is a subtle difference. For a comment that begins with an asterisk, the signal that ends it is simply the first semicolon that appears. The COMMENT statement, on the other hand, tokenizes the first token of the statement after the introductory "`COMMENT`" command name identifier – and then it searches for a semicolon character. Consequently, if the first token after "`COMMENT`" happens to be a character string literal (begins with a quotation mark or apostrophe), then any semicolons that might appear within that character string literal will *not* end the comment; they are not "seen" in the process of searching for the ending semicolon. Within the COMMENT statement, beyond that first token, no more tokenizing is performed, and the character stream is scanned for the first semicolon character.[22] Thus, you could have,

```
COMMENT "a character string literal as the first token past
the command name; with ; embedded; semicolons;" then comes
more text;
```

The semicolons inside the character string literal are not "seen" in the process of searching for the ending semicolon. But if such a token appears as the second or later token after "`COMMENT`", then the semicolons within it are seen, and they act to end the comment:

```
COMMENT some stuff and a "character string literal with an
embedded ; semicolon" and more stuff;
```

This is not a proper COMMENT statement; the first semicolon – inside the character string literal – ends the token, because it is "seen"; it is not in the first token past "`COMMENT`". Consequently, the text,

```
semicolon" and more stuff;
```

is also seen and is invalid code.

The `/* */` style comment can go between tokens – anywhere that a space is permitted. It is not a token, but is part of the whitespace that is skipped when looking for the next token.[23]

The final form,

```
%* your comments go here;
```

is a macro comment; it is part of the SAS macro facility, which, except for the macro comment feature, is not accommodated by SASdecoder. This comment form is distinct in that the entire contents of the statement are tokenized, and consequently, the ending symbol is a semicolon *token* (a semicolon not within a character string literal). Thus, what was said about semicolons inside a character string literal if it is the first token past `COMMENT` in the COMMENT statement applies here to the *all* character string literals in the macro comment.

---

[22] If the first non-whitespace character after `COMMENT` is a semicolon, the statement ends there.

[23] From a formal perspective, in the other comment forms, the asterisk and the COMMENT identifier are tokens, but the content of the comments are not.

To recap,

- a comment beginning with an asterisk is not tokenized
- a comment beginning with `COMMENT` has the first token past `COMMENT` tokenized
- a macro comment is fully tokenized
- this form `/* your comments go here */` is whitespace

It is worth noting that, for any part of a comment that is tokenized, any character string delimiter (quotation mark or apostrophe) must be balanced by another delimiter of the same type. Note too, that, since character string literals can span lines, a spurious or missing quotation mark or apostrophe in a COMMENT statement or macro comment may throw off the reading of the subsequent text of the program. See **Character String Literals** for more on this matter.

All forms of comments may span multiple lines.

Comments may not be nested.[24]

The `/* */` type of comment ends at the first occurrence of */, regardless of any attempted nesting or enclosure within quotation marks.[25]

However, a comment using the `/* */` form should also not contain a semicolon. Actually, SASdecoder has no problem with this, but it may cause peculiar behavior in SAS.

The content – what's inside the comment – is completely skipped by SAS and SASdecoder. There are two uses for this: 1, to insert comments, i.e., notes to any human reader of the program, and 2, to hide segments of code – to make them virtually invisible to the code processor, but to not delete them outright. (You may want them back at a later time, or they are of historical significance.) The latter use, "commenting out" code, is something you can expect to do in preparing a SAS file for use by SASdecoder, as there may be features in the original SAS program that are not accepted by SASdecoder.

## Identifiers and Name Space

Identifiers are the "words" used in SAS code. They are used for a variety of purposes, most notably as variable names, but also as statement names, filenames, and other entities. An identifier is an

---

[24] If you attempt to nest `/* */` type comments, you will cause the following situation. If you write...

`/* a /* b */ c */`

(with *a*, *b*, and *c* not containing any more instance of `/*` or `*/`) then the comment actually ends with the `*/` after *b*, and *c* `*/` will be "seen" by SASdecoder (and may cause an error). If either *c* is accepted as legal, or it is absent; and if the context is such that a command is expected, then the final `*/` will actually be the start of a `* ... ;` type comment.

It is believed that this behavior mimics SAS, though certain other peculiar behaviors may occur in SAS with these situations.

[25] Enclosing comment delimiters in quotation marks may invoke some peculiar behavior in SAS.

unbroken sequence of letters, underscores and digits, but the initial character may not be a digit. Identifiers are case-insensitive (though note that in Stata and some other data-handling software, identifiers – notably, variable names – are case-sensitive).

Each instance of an identifier is interpreted in a particular "name space". There are several name spaces, and the same identifier may occur in different name spaces, each with a distinct interpretation of the same identifier. Thus, for example, a variable name may be the same as a statement name; you can use "input" as a variable name. (This may be considered poor programming practice, as it may confuse human readers, but it is possible and causes no confusion to SAS or SASdecoder.) The context in which an identifier is used determines the name space in which it is interpreted.

The name spaces recognized by SASdecoder are...

- Statement names (e.g., INFILE, INPUT, LABEL, FORMAT, DATA, RUN)
- proc names (e.g. FORMAT; note that FORMAT is both a statement and a proc)
- variable names
- formoids ("base names" for formats)
- informoids ("base names" for informats)
- infile options (LRECL, N, FIRSTOBS)
- dataset names
- librefs (the first part of a two-part dataset name)
- filenames (as defined in a FILENAME statement)
- libnames (as defined in a LIBNAME statement)
- character string literals

Some of these, in particular command names, infile options, informoids and proc names, are not at the user's control; all the others are names that are created by the user's code. The important thing to know is that each of these categories is a distinct "space" in which identifiers exist independently of the other name spaces.[26]

Character string literals are not identifiers, but the class of character string literals functions as a name space for the purpose of parsing SAS code. They also differ from identifiers in that the case is preserved and they may span lines.

While it is possible to use any identifier as a variable name in SAS, in many data-handling facilities, certain names are reserved and are unacceptable as variable names and possibly other features. Thus, you need to be wary of this possibility in the course of reading or transferring the data. SASdecoder will flag variable names that are Stata-reserved names, which are the following:

[26] The author is not aware of whether this is exactly the same set of name spaces used by SAS, though it seems to be a reasonable emulation of SAS behavior within the set of statements that SASdecoder recognizes. The terms "formoid" and "informoid" were created by the author for the purpose of this discussion. They are the base names – the part before the width, the period, and decimal – of formats and informats, respectively. For example, in the informat F10.2, F is the informoid. (There is also an F formoid as well.) Note that some formoids are pre-loaded, while others – the values established in VALUE statements – are user-defined.

_all _b byte _coef _cons double float if in int long _n _N _pi _pred _rc _se _skip using with

See the Stata User's Guide for more information – [U] 14.3 Naming conventions, in both the Stata 7 and 8 User's Guides, [U] 11.3 in the Stata 9 and 10 User's Guides, or see "reserved names" in the Stata User's Guide index.

In accordance with Stata lexical rules, this checking for reserved names is case-sensitive and is performed on the names as they appear in the output. Thus, this name checking is dependent on whether the output is in upper or lower case. Rendering variable names in upper case will eliminate the possibility of a problem with all reserved names except _N (though many users will prefer to not use upper-case variable names). These names are believed to not be prohibited from use in other Stata entities such as value names, and are of no consequence in such SAS entities as libnames and filenames.

If you are outputting a Stata dictionary, and any of these identifiers appear, you will need to modify the name(s) – either in the resulting dictionary or in the SAS file (and rerun the translation). In the Stat/Transfer schema file, these Stata-reserved names are also flagged, just for informational purposes. They may or may not pose a problem, depending on your ultimate targeted data format. (And there is no checking for names that might be illegal in any other data format.[27] You may need to look for variable names that are illegal, if there are any, in your targeted data format.)

SASdecoder has no particular limit to the length of identifiers, though SAS, Stata, and many other data facilities have a limit of 32 characters for variable names. If a variable name appearing in an INPUT statement exceeds 32 characters, you will get a warning in the output file, but since SAS has the same limit, a valid SAS specification should never invoke that warning.

## Character String Literals

Many types of statements (in SAS and most other programming languages) can include components that are textual values. Such a value needs some way to mark its beginning and end[28], and to distinguish it from other lexical features, since a textual value could consist of just about any sequence of characters, which could be confused with some other token type or a meaningless stream of text. This is accomplished by surrounding the value by a designated delimiter character – a quotation mark or apostrophe. Such a construct is called a character string literal, and its value (or "content") is the text between the delimiters. The whole thing should be regarded as a textual representation of a textual value.

In SAS, the delimiter may be either a quotation mark or an apostrophe. But whichever one is used for introducing the character string value must be the same one that ends it. That is, the same delimiter must be used at both ends. Thus,

```
"faminc01.dat"
```

and

```
'faminc01.dat'
```

[27] Stat/Transfer may in fact create a variable that is illegal in the given target data format; it can create a variable that would be impossible to create within the chosen data software, such as "using" in Stata.

[28] --since, for one thing, it may contain spaces.

are equivalent.

Within the content of a character string literal, a double occurrence of the delimiter serves to represent a single occurrence of that character. Thus `'nine o''clock'` represents nine o'clock.

Given a character string that is introduced by one of the delimiters, the other delimiter has no special significance; it is like any other character. Thus, the value represented in the previous example could also be represented by `"nine o'clock".`

In all of what follows, character string literals will be delimited by quotation marks, but it should be remembered that either form is acceptable.

Character string literals in SAS may span lines. The “pieces” will be concatenated. Thus,

```
"hello
 world"
```

and

```
"hello world"
```

are equivalent.

## Numeric Literals

Another feature found in SAS code is the numeric literal – a textual (decimal) representation of a number. In SASdecoder this is limited to nonnegative integers, e.g., `3`.

There are structures that may look like floating point numbers, e.g., `6.2`, however, within SASdecoder, the only occurrences of such a constructs are in formats and informats.

## Formats and Informats

Another important set of lexical features comprise formats and informats. These are specifications for how values are represented textually; formats apply to output, informats apply to input (the reading of raw data). Their use will be described elsewhere; the present discussion is about their lexical form. Formats and informats are unbroken strings of letters, underscores, or digits, along with a single period, which is either within the string or at the end, but never at the beginning. It may look like a decimal number, e.g., `8.2`, or it may have letters or underscores, in which case it may not start with a digit. If the period is not at the end, then what follows the period must be a sequence of digits. Usually, there are also digits preceding the period as well, as in `F5.1` . The digits to the left of the period specify a field width; the digits to the right specify implicit decimal places. Thus, for example, an `F3.1` informat specifies a width of 3 and 1 implied decimal. It will cause a field value of 142 to be interpreted as 14.2. The decimal specification may be absent, in which cases it is equivalent to 0.

If there are non-digits present in the format or informat, they form the "base name"; different formats and informats may share a base name, with the width and decimals forming different variations. Here are several variants of the F informat, where F is the base name:

- `F8.0`
- `F12.2`
- `F12.4`

Formats and informats may also have a preceding dollar sign (`$`) to indicate a character string variable or character string format. For example, `$CHAR14.` The `$` may be attached or separated: `$CHAR14.` or `$ CHAR14..`

(Actually, these are not precisely the same, but the effect may be equivalent.)

SAS allows specific formats and informats, and there are many that serve as both formats and informats. To SASdecoder, formats and informats look the same, though where informats are used, only certain ones are accepted.

In addition to formats and informats that are predefined (more precisely, have predefined base names), there is the possibility of user-defined formats, as defined in the `PROC FORMAT` statement, which will be explained later.

## Special Characters

There are occasions that use special characters such as `/`, `$`, `+`, `@`, `#`, `&`, `*`. These will be described as needed. The thing to remember is that, from a lexical standpoint, they are generally each a one-character token.

# SAS Language Features Accepted by SASdecoder – Structure

This section will explain the structural features of SAS programs that SASdecoder accepts.

## SAS Steps

SAS programs operate along the concept of "steps" – segments of code that are processed as a unit. SASdecoder also follows this concept of steps.

There are two types of steps: data steps and proc steps. Every point within a SAS program either initiates a step or is in a data step, a proc step, or a neutral zone (a region that is not in a data or proc step). Neutral zones occur before the first step, or between steps, and only a few statements are valid in a neutral zone. In terms of what SASdecoder accepts, only `FILENAME`, `LIBNAME`, `COMMENT`, and `RUN` are allowed in a neutral zone. (`RUN` is allowed but has no effect in a neutral zone. Also, `DATA` and `PROC` are allowed, as they initiate steps.)

Most statements are valid only in some steps. A statement occurring in the wrong context will usually result in a warning or may invoke a fatal error.

A data step begins with a `DATA` statement.

A proc step begins with a `PROC` statement.

Either kind of step ends whenever...

- a new step is started (i.e., a `DATA` or `PROC` statement occurs);
- a `RUN` statement occurs;
- the end of the file is encountered.

It is helpful to consider a data step as a code loop that runs through the set of all observations. Also, variables are meaningful only within the data step that defines them. See the section on **Scope of Variables; Permanent vs. Local Symbols** for more on this.

SASdecoder recognizes only one type of proc step: `PROC FORMAT`, and within that proc, it only recognizes `VALUE` statements (and a limited form of them). These set up a correspondence between numeric values and text – what are known as value labels in Stata and some other data-handling facilities.

The significant value of SASdecoder lies in two areas:

- its capacity to translate the elements of `INPUT` statements that appear in `DATA` steps;
- its capacity to translate `PROC FORMAT` and `VALUE` statements.

In either case, these statements are tied to the steps in which they are located.

Usually a data step represents a read-in operation of a single raw data file. But note that SAS can operate on multiple files in one data step, though SASdecoder does not handle this possibility. This will be explained further in the section on **The INFILE statement**.

In a typical use, the SAS program you are translating would have just one data step. But SASdecoder can also accommodate multiple data steps, and many existing SAS files involve multiple data steps. Some adjustment to the output will be necessary in that case. The reason is that each data step – each file read-in operation – needs to have its own corresponding Stata dictionary or Stat/Transfer schema file, since each Stata dictionary or Stat/Transfer schema file performs a single read-in operation. But SASdecoder is set up to produce just one Stata dictionary and/or one Stat/Transfer schema file in each run. Thus, in the cases where you have multiple data steps, the resulting Stata dictionary or Stat/Transfer schema file will need to be broken up into several separate files, one for each data step.

The Stata do-file, on the other hand, could be used for multiple read-in operations. This would require these steps: (1) Break up the dictionary to several files, one for each data-step. Each dictionary will naturally have a distinct name. (2) In the do-file, substitute the corresponding dictionary names in each `infile` command. (3) In the do-file, place `save` and `drop _all` commands between each data step. There should be commented text indicating where to place these. You will need to specify distinct file names with each `save` command. Note that the use of drop _all (rather than clear) will preserve any value labels that may be needed from one `infile` to the next.

An alternative way to handle this situation would be to break up the SAS code into separate files, one for each data step. (You would need to retain a copy of any common VALUE definitions that may be used in each of the files – or run the files in sequence, using the "retain perm symbols" or `retain_perm` option.)

## Scope of Variables; Permanent vs. Local Symbols

An identifier's scope is the range of statements in which it is meaningful. In SAS, a variable's scope runs from its first appearance within a data step through the end of that step; it ceases to exist past the end of the step. Thus, variables are "local" to the steps they inhabit; a variable defined in one data step is unknown to other data steps, and the same variable name may appear in different data steps, but they are completely independent.

Certain other identifiers are "permanent"; they exist from the point they are first defined, and remain in existence, independent of step boundaries, until they are explicitly deassigned, or to the end of the SAS session. Some are predefined, such as the `F` and `CHAR` informoids. The permanent symbols that SASdecoder recognizes are…

- value labels
- filenames
- libnames
- formoids (base names for formats)
- informoids (base names for informats)

As mentioned in the section on **Options**, the default action of SASdecoder is to deassign all user-defined permanent symbols at the end of each invocation of the parser. In SAS, on the other hand, permanent symbols persist between runs of SAS code within the same SAS session. Thus, a run of code can make use of any permanent symbols (value assignments, libnames, etc.) from an earlier run, and a subsequent rerun of SAS code that includes assignments of permanent symbols will actually cause the symbol assignments to replace the earlier ones, even they are the same as before.

By using `option [retain | retain_perm] on` or the "retain perm symbols" checkbox, this is changed so that permanent symbols persist between invocations of the parser, and your SASdecoder session will behave like a SAS session in this regard. Then you can use the "Clear Perm Symbols" button or the `clearperm` or `clear_perm` commands to clear these symbols, if desired. Please note that this pertains to user-defined permanent symbols; predefined permanent symbols remain perpetually defined; they are unaffected by these options and actions.

## Data Types Revisited

Every variable has a data type: the types of values it may contain – most significantly, whether it is numeric or character string. Nearly every statement that deals with variables will set the type if it is not already set, and for character string variables, the length can usually be set as well. Thus, you may have seen or heard, "a variable's type is determined in its first appearance."

There is one exception among the statements accepted by SASdecoder: the `LABEL` statement (or an `ATTRIB` statement that assigns only a label) does not set the type.[29]

As mentioned, character string variables usually get their length defined as well. But if no length is defined, it will default to 8.

While many different types of statements can define variables, only those variables that appear in an `INPUT` statement will show up in the Stata dictionary or Stat/Transfer schema.

[29] If a variable is first mentioned in a LABEL (or equivalent ATTRIB) statement, it is untyped at that point. It can acquire a type subsequently, or will eventually default to numeric.

## SAS Statements Accepted by SASdecoder

SAS statements begin with a statement name such as DATA or FILENAME. SASdecoder accepts the following SAS statements, that is, statements beginning with these identifiers:

- COMMENT
- DATA
- RUN
- FILENAME
- LIBNAME
- INFILE
- INPUT
- LABEL
- PROC
- VALUE
- FORMAT
- INFORMAT
- LENGTH
- ATTRIB

This is a small subset of the statements that SAS uses, but these are the ones which are essential to the task of reading raw data, plus a few other related statements. In what follows, there will occasionally be a reference to what SAS accepts – as contrasted with what SASdecoder accepts.

In the following syntax descriptions, elements in UPPERCASE, FIXED-WIDTH FONT represent segments of text that are to be written as given, though lower-case is accepted as well. Elements written in *italic font* are symbolic entities that are to be substituted by actual values. For example in the description for FILENAME,

FILENAME *identifier* "*file_name*";

"FILENAME" is written as part of the statement, whereas *identifier* is to be substituted by an identifier and *file_name* is to be substituted by a filename, as in

```
FILENAME inp "C:\AHS\AHS1975RAW.TXT"
```

Elements written [inside square brackets] are optional. The square brackets are symbols used here for describing the syntax; there are no actual square brackets used in any of the SAS language features that SASdecoder understands. Thus, for example, a description such as

*varname* [$] *position_spec*

indicates that the dollar sign is optional; the square brackets do not appear in actual instances of the form being described.

An ellipsis (...) indicates that the preceding construct may be repeated an arbitrary number of times. Only the one preceding element may be repeated. Thus, in

VALUE *name value_element ...* **;**

one or more *value_element* entities may follow *name*.

Following is a description of the SAS statements that SASdecoder understands. Subject to the limited set of SAS features that SASdecoder allows, SAS grammar is followed as best as is practical, but may differ slightly from actual SAS grammar.

## Variable_Lists

Several of the statements accept a list of one or more variables, with the additional possibility of certain wildcard forms. Wherever such a list is accepted it will be denoted as a "variable_list". Such a list consists of a sequence of one or more (in some contexts, it might be zero or more) instances of the following types of entities, separated by whitespace.

- a variable name (an identifier)
- a "colon wildcard" – a possible prefix of variable names, followed by a colon, as in `ABC:`
- a construct consisting of two items separated by a hyphen, denoting a range of numerically-suffixed identifiers such as `M13-M24`
- a range of variable names, indicated by pair of existing variables, with a double hyphen between the two names. Example: `EMPID--SALARY`
- a range of existing variables, with a limiting qualifier between two hyphens:
  - `EMPID-CHARACTER-SALARY`
  - `EMPID-CHAR-SALARY`
  - `EMPID-NUMERIC-SALARY`
- one of the predefined wildcard symbols:
  - `_ALL_`
  - `_CHARACTER_`
  - `_CHAR_`
  - `_NUMERIC_`

Some of these forms refer to "existing variables" or variables that "already exist". By this, we mean the set of variables that have been established – that is, mentioned in *previous* statements – prior to the statement in which the variable_list appears.

A colon wildcard indicates all existing variables that start with the indicated prefix. Thus, `CAT:` represents all existing variables that begin with `CAT`; it may, for example, include `CATALOG`, `CATEGORY`, and `CATAPULT`, assuming that these already exist as variable names.

A range of numerically-suffixed identifiers represents a list of names formed by concatenating the common prefix with the set of integers in the indicated range. Thus, `M13-M24` represents

```
M13 M14 M15 M16 M17 M18 M19 M20 M21 M22 M23 M24
```

Note that if the numeric suffixes have differing widths, the generated names use the lesser width. For example,

- `ZZ8-ZZ12` generates `ZZ8 ZZ9 ZZ10 ZZ11 ZZ12`
- `ZZ08-ZZ12` generates `ZZ08 ZZ09 ZZ10 ZZ11 ZZ12`
- `ZZ008-ZZ12` generates `ZZ08 ZZ09 ZZ10 ZZ11 ZZ12`

You can also write the range in backward order; `ZZ12-ZZ8` generates the same set of names as `ZZ8-ZZ12`, though in reverse order.

The names generated by such a construct can be new or already-existing variables. If they are new, then they are being established by the present statement

A range of variable names is indicated by pair of existing variables, with a double hyphen between the two names[30]. This represents the set of variable names, spanning the two names as found in the list of existing variables *in the order that they have been established*. Thus, if prior statements have established the variables `CASENO`, `EMPID`, `EFFD`, `JOBCODE`, `JOBGRADE`, `SALARY`, `STATUS`, in that order, then `EMPID--SALARY` represents `EMPID EFFD JOBCODE JOBGRADE SALARY`.

Note that the two "boundary names" must exist and appear in the given order. In the example, `EMPID` must precede `SALARY` in the list of existing variables.

The next form is a range of existing variables, with a limiting qualifier between two hyphens. This is just a variation of the basic range of variable names, but limited to either character or numeric types. Thus, `EMPID-CHARACTER-SALARY` is the same as `EMPID--SALARY`, but limited to character variables. (`CHAR` is a synonym of `CHARACTER`.)

Finally, the general wildcard symbols represent all variables, or all variables of one type.

- `_ALL_` represents all existing variables.
- `_CHARACTER_` represents all existing character variables.
- `_CHAR_` also represents all existing character variables.
- `_NUMERIC_` represents all existing numeric variables.

Any of the wildcard forms (other than the unqualified name range) might yield an empty set of variables, if no existing variables match the given criteria.

A varliable_list component of the form a-numeric-b will pick up all variables, in the range, that are not yet defined as character. This is true, even if it's for a statement that makes it character. For example,

```
length a $4;
label b = "hello";  /* b is untyped at this point */
length c $5;
length a-numeric-c $12;
```

[30] The two hyphens can be separated.

That will pick up b as numeric, because it is not character, and then define it as character. This behavior has been observed in SAS and has been emulated in Sasdecoder.

## Limited_Variable_Lists

In certain contexts, in INPUT statements, in particular, a limited form of a variable_list is used. This encompasses all the possible features of a variable list, *except* for…

- a "colon wildcard" – a possible prefix of variable names, followed by a colon, as in `ABC:`
- the predefined wildcard symbols: `_ALL_`, `CHARACTER_`, `_CHAR_`, `_NUMERIC_`.

## Limited_simple_integer_expressions

Some syntactical elements take what could be called a limited_simple_integer_expression. These are the values that can occur after `@`, `+`, or `#` in an INPUT statement. They are usually just an integer literal, such as `15`, but other forms are possible. SAS allows a general expression (except for a missing value literal) in these contexts, however, SASdecoder will accept only a limited set of forms.

To explain what a limited_simple_integer_expression is, we will first define a simple_integer_expression. This comprises…

- an integer literal
- an identifier
- (*simple_integer_expression*)
- +*simple_integer_expression*
- −*simple_integer_expression*

This is a recursive definition, allowing for nesting of parentheses and multiple leading unary plus and minus signs. The minus sign yields the negative of the value it precedes, as you would expect; the plus sign has no effect. Though this may resemble the definition of a generalized expression, it does not allow (that is, the present capability of SASdecoder does not allow) any binary operations; the construction of such an expression must terminate with a single integer literal or identifier.

Having established that definition, then the definition of a limited_simple_integer_expression is…

- an integer literal
- an identifier
- (*simple_integer_expression*)

That is, a leading unary plus or minus sign may not appear at the outermost level of such an expression (that is, a plus or minus sign may not be the starting character), but may appear deeper within it. In other words, if you want to use a leading unary plus or minus sign, then it, along with its following subexpression, must be enclosed in parentheses.

Note that in the contexts in which these expressions are used in SASdecoder, the use of an identifier yields a non-determinable value. This will be explained further in the various places that refer to these kinds of expressions.

## The COMMENT statement

Syntax:

> `COMMENT` *nearly_anything_except_a_semicolon* `;`

The COMMENT statement ends at the first semicolon – except that any semicolon embedded within a character string literal that is the first token past `COMMENT` does not count as ending the command.

The content of the statement – that is, all text up to the ending semicolon – is skipped over.

The statement name `COMMENT` can be substituted by an asterisk, though there is a subtle difference; the remark above about semicolons embedded within a character string literal that is the first token past `COMMENT` does not apply in this case.

See the **Comments** section under the major heading of **SAS Language Features Accepted by SASdecoder – the Lexical Level**, for more details on the syntax of these statements and for other methods of denoting comments.

## The DATA Statement

Syntax:

> `DATA` [*dataset_name…*]`;`

The DATA statement begins a data step; it will also end a step of either type if one is already in progress.

The dataset_name, if present, may be a simple one-word name:

```
DATA mydata;
```

or it may be in a two-word name – two identifiers joined by a period:

```
DATA cats.mydata;
```

(No spaces are allowed around the period.) The first component is a libref; the second is the actual dataset name. But these distinctions have no significance in SASdecoder.

The only effect of the DATA statement is to introduce a data step. The optional dataset name and libref are parsed but subsequently unused.

Note that SAS allows a great variety of options on the DATA statement, which are not currently accepted by SASdecoder. If your SAS file has such options, you will need to edit or comment them out.

## The RUN statement

Syntax:

```
RUN;
```

This serves to end a data step or proc step. (It is also allowed in a neutral zone, with no effect.)

## The FILENAME Statement

Syntax:

FILENAME *identifier* ["*file_name*"];

This establishes an association between the identifier and the file name, which can later be used in an INFILE statement. (See **The INFILE Statement** for an example of this.) This association must be made before it can be used; that is, the FILENAME statement must come before the use of the identifier in an INFILE statement.

The FILENAME statement can be used anywhere, including neutral zones. The scope of the identifier is from its definition forward through the end of the program, or until it is redefined by another FILENAME statement. That is, filenames are permanent symbols. See the section on **Scope of Variables; Permanent vs. Local Symbols** for more on this matter.

In SAS, within a given session, the assignment made by this statement will persist beyond the running of a given SAS file, into the next run – analogous to a global macro in Stata. Thus, in SAS, if the SAS file neglects to assign a filename or if the FILENAME statement fails due to a grammatical error, but you assigned the filename (correctly) earlier in your session, then you may have a correct run anyway. SASdecoder will emulate this behavior if the "retain perm symbols" checkbox is checked or the `retain_perm` option is on.

You can reassign a filename identifier:

```
FILENAME inp "C:\AHS\AHS1975RAW.TXT";
FILENAME inp "C:\AHS\AHS1980RAW.TXT";
```

The latter value prevails.

You can deassign (erase) a filename identifier by omitting the filename:

```
FILENAME inp;
```

Inp now has no assigned value.

Note: In SAS, the FILENAME statement may have grammatical errors that are non-fatal (though there may be consequential fatal errors elsewhere). In SASdecoder, grammatical errors in these statements are fatal. This difference should not likely be very inconvenient.

Older SAS documentation indicates that you can put multiple definitions in one FILENAME statement: `FILENAME fileid1 "name1" fileid2 "name2";` This, apparently, is no longer valid, and is not accepted by SASdecoder.

## The LIBNAME Statement

Syntax:

> `LIBNAME` *identifier* [*lib_specification*]`;`

In SAS, there are several forms of *lib_specification*. SASdecoder accepts only one form: a character string literal. Thus, the syntax in SASdecoder is limited to…

> `LIBNAME` *identifier* ["*lib_name*"]`;`

This has no effect in SASdecoder, but it is accommodated because it typically occurs in the SAS files that read raw data. The *identifier* is known as a libref. The assignment of the libref to the specification is recorded, but no further use is made of it.

SAS also accepts other forms and options, including the possibility of concatenating libraries by writing a space-separated list of one or more lib_specifications enclosed in parentheses:

> `LIBNAME` *identifier* `(`*lib_specification...*`)``;`

where each *lib_specification* is either a character string literal or an existing libref. This form is not accommodated by SASdecoder.

The LIBNAME statement can be used anywhere, including neutral zones. The scope of the identifier is from its definition forward through the end of the program, or until it is redefined by another LIBNAME statement. That is, libnames are permanent symbols. See the section on **Scope of Variables; Permanent vs. Local Symbols** for more on this matter.

(As with the FILENAME statement, in SAS, the effect of this statement will persist beyond the running of a given SAS file within the same session. SASdecoder will emulate this behavior if the "retain perm symbols" checkbox is checked or the `retain_perm` option is on. However, since there is no real effect within SASdecoder, this is of no consequence.)

As with FILENAME, you can reassign and deassign libname identifiers; refer to the section on FILENAME for examples.

Note: In SAS, the LIBNAME statement may have grammatical errors that are non-fatal (though there may be consequential fatal errors elsewhere). In SASdecoder, grammatical errors in these statements are fatal. This difference should not likely be very inconvenient.

Older SAS documentation indicates that you can put multiple definitions in one LIBNAME statement: `LIBNAME libref1 "name1" libref2 "name2";` This, apparently, is no longer valid, and is not accepted by SASdecoder.

## The INFILE statement

xSyntax:

```
INFILE "file_specification" [option ...];
```

or

```
INFILE identifier [option ...];
```

The INFILE statement is used in a data step to specify the raw data file. In the first form, *file_specification* is the name of the raw data file. In the second form, *identifier* is an identifier that was previously defined in a FILENAME statement, which names the raw data file. Thus, you may write...

```
INFILE "faminc01.dat";
```

or, equivalently,

```
FILENAME familyincomefile "faminc01.dat";
INFILE familyincomefile;
```

The options will be described below.

The effect of the INFILE statement is to place the file specification into the dictionary header line in the Stata dictionary:

```
dictionary using faminc01.dat {
```

or in a FILE statement in the Stat/Transfer schema file:

```
file faminc01.dat
```

Without such an INFILE statement, you will get a note in the output alerting you to the absence of a file name.

The INFILE options accepted by SASdecoder are

- `LRECL=` *n*
- `N=` *n*
- `FIRSTOBS=` *n*

where *n* stands for a positive integer. Thus, you may have, for example,

```
INFILE "faminc01.dat" LRECL=124 FIRSTOBS=9;
```

The `LRECL` option adds a brief note to the dictionary or Stat/Transfer schema file, alerting you to the fact that an `LRECL` option was present in the SAS file. This may just indicate that the raw data file has lines that are all of the same length as specified in the option (124 in this example), and is of no consequence to Stata or Stat/Transfer. In possibly rare circumstances, the raw data file may be "unformatted" – having no end-of-line markers. If so, and you would need to inspect your raw data to determine this, you may require the `_lrecl()` feature in the Stata dictionary; Stat/Transfer does not have such a feature. (Apparently, SAS takes this feature to mean "impose an

end-of-line after *n* characters if it isn't explicitly there, but respect the end-of-line if it is." Stata has a same-named feature, but it behaves more strictly and is much more rarely needed; see the Stata help for infile2 for more on this.[31]).

Most raw data files nowadays are formatted into lines, and the `LRECL` feature is irrelevant to either Stata or Stat/Transfer (and SAS). But if you do have an unformatted raw data file, insert `_lrecl(`*n*`)` into the Stata dictionary. If you use Stat/Transfer, you would need to do some conversion operation to format the raw data.

The `N` option has no effect; its use in SAS has no corresponding feature in Stata or Stat/Transfer. Note that this *n* in the `N` option is not necessarily the number of lines per observation. The number of lines per observation is determined by the maximal line specified with "line number movement" directives. See the explanation of #*n*, below, under **Input Specifications** for more on this matter.

The `FIRSTOBS` option will generate a

```
_firstlineoffile(n)
```

directive in the Stata output, or

```
first line n
```

in the Stat/Transfer schema file. These directives indicate that the data actually does not begin until line *n* in the raw file.

No other INFILE options are currently accepted. If your SAS file has other options, you should edit or comment them out.

IMPORTANT NOTE: SAS uses a dynamic interpretation for the INFILE statement, whereas SASdecoder uses a static interpretation, which is a natural consequence of the static interpretation used by both Stata and Stat/Transfer. Operationally, this means that, in SAS, an INFILE statement specifies which file to read the next data field from; you can think of it as an introductory clause to an INPUT statement. Thus, you can, in one data step, switch back and forth between different raw data files:

---

[31] The `_lrecl()` directive in Stata would be needed if the raw data were a stream of characters with no intrinsic formatting into records or lines – that is, no "line breaks". Thus, you will need to know the low-level structure of your raw data file to make the decision as to whether an `_lrecl()` directive is needed. In actual use in modern computing environments, it is very rarely needed. If you suspect that you do need it, you may want to test your dictionary with and without it. For more information, see "Dictionary directives" under "infile (fixed format)" in the Stata Data Management Manual.

Note that Stata and SAS use `_lrecl(`*n*`)` and `LRECL=`*n*, respectively, in somewhat different ways, if the file has built-in line breaks (as do normal text files). In Stata, it causes the file to be reformatted into records of length *n*, beginning every *n* characters, ignoring (skipping over) any built-in line breaks. SAS, on the other hand, seems to be sensitive to the line breaks, and will truncate lines to *n* characters if they are longer than *n*. In other words, Stata requires an `_lrecl(`*n*`)` in the absence of line breaks, and will trip if both the `_lrecl(`*n*`)` and line breaks are present; SAS is not bothered if both are present. So the behavior of Stata and SAS may differ in this regard, given the same raw data files and these seemingly equivalent directives or options.

```
DATA;
INFILE "abc.txt";
INPUT v1 v2 v3;
INFILE "def.txt";
INPUT v4 v5;
RUN;
```

In SAS, the effect of this would be that v1, v2, and v3 are read from abc.txt, and v4 and v5 are read from def.txt, creating a single dataset which is effectively a merge of two datasets. Stata and Stat/Transfer cannot do this in one step.[32]

SASdecoder interprets such code by scanning the entire data step, and retaining the last INFILE statement. Thus, the example above would result in a Stata dictionary or Stat/Transfer schema that references only def.txt. You would, however, get a warning alerting you that multiple INFILE statements were encountered; this warning appears on the screen, or in the Progress Log list box of the Windows Interface, as well as in the output files.

Other odd situations are possible. Consider this example:

```
DATA;
INFILE "abc.txt";
INPUT v7 v8;
INFILE "def.txt";
RUN;
```

In SAS, the effect of this would be that v7 and v8 are read from abc.txt. Then the designated file for reading raw data is switched to def.txt, but that has no effect, since it is immediately switched back to abc.txt the next time through the loop. So the outcome is that all data are read from abd.txt, and def.txt is effectively ignored. Meanwhile, SASdecoder would interpret this as reading from def.txt *only*. This is an unfortunate consequence of SASdecoder's "use the final infile mentioned" policy, but it would be particularly complicated to include an algorithm to detect these sorts of situations. Fortunately, these situations are rare; this example might be considered pathological. In realistic situations, reading from multiple files corresponds to some sort of join or merge of datasets. The practical thing to do is to arrange to read the files separately and then do an appropriate `merge` in subsequent processing.

One final note on INFILE: SAS requires that the file exist; SASdecoder does not make that check.

## The INPUT statement

Syntax:

`INPUT` *input_spec ...*`;`

[32] Stat/Transfer cannot do this; Stata can possibly do it in several commands, e.g., with a subsequent `merge`, but not within the read-in operation (`infile`) by itself.

The INPUT statement is one of the essential parts of SASdecoder, and will be described below under the heading **Input Specifications**.

## The LABEL statement

Syntax:

```
LABEL varlabel_assignment … ;
```

where *varlabel_assignment* is

```
varname = "label"
```

The LABEL statement assigns variable labels to variables. (Variable labels are distinct from value labels; see **The VALUE Statement** and **The PROC Statement** (PROC FORMAT) for how to create and assign value labels.)

If a variable is mentioned multiple times in one or more LABEL statements (in the same or in separate LABEL statements), the final one will be used.

In the Stata dictionary, variable labels will appear quoted. If the label contains an embedded quotation mark, then it is put into compound quotes (`` `" "' ``) to enable Stata to correctly read the label. For example,

```
_column(403) byte lahca32 %1f `""Old age" causes limitation"'
```

But no testing or special action will be taken for embedded unbalanced compound quotes, which could cause Stata to misread the dictionary. While it would be possible to deliberately construct an example, it is unlikely for such a configuration to occur “naturally”.

In the Stat/Transfer schema, variable labels appear in braces (`{ }`). There is presently no special action taken if there is an embedded brace in the label, which might cause Stat/Transfer to misread the schema.

Labels can also be assigned with an ATTRIB statement (q.v.).

Note that, whereas several other statements (FORMAT, INFORMAT, LENGTH, ATTRIB) take variable_lists, LABEL takes only a single variable in each label specification.

## The PROC statement

Syntax:

```
PROC procname;
```

The PROC statement begins a proc step; it will also end a step of either type if one is already in progress.

In SAS, options may follow *procname*, but SASdecoder does not accept any. Also, while SAS has a great multitude of procs available, the only *procname* accepted by SASdecoder is FORMAT. Thus, in SASdecoder, the only syntax accepted for PROC is...

`PROC FORMAT` [*procformatoptions* …];

Once this step is initiated, a VALUE statement may be used. (In SAS, every PROC has a specific set of allowable statements. For PROC FORMAT, in SAS, these are VALUE and PICTURE, but only VALUE is accepted by SASdecoder.)

For *procformatoptions*, the possibilities are…

`LIBRARY` = *identifier*

`DECK`

In SAS, *identifier* must be a valid libref or fileref (presumably defined in a `LIBNAME` or `FILENAME` statement). SASdecoder does not impose that restriction; thus it can be any identifier. Both the `LIBRARY` and `DECK` options have no effect on the output.

## The VALUE statement

The VALUE statement creates a user-defined format (or more precisely, a formoid), which sets up a correspondence between (extended) integer values and textual values – what are known as value labels in Stata and Stat/Transfer. It is allowed only within a PROC FORMAT.

Syntax:

`VALUE` *name value_element ...* **`;`**

where *name* is a valid SAS name, not ending in a digit, and *value_element* is

*extended_integer* = **`"`***label*`"`

where *extended_integer* is either an integer (including negatives) or one of the missing values

`.`, `._`, `.a`, `.b`, `.c`, etc.

Typically, there are many value elements, as in

```
VALUE region
 1 = "Northeast"
 2 = "South"
 3 = "Midwest"
 4 = "West";
```

*name* is typically "new" – not yet seen in the namespace of formoids. But if it has been seen before, then the present VALUE statement overwrites the old. *name* may not be a built-in formoid name, such as `F`.

The (extended) integer values must be distinct. If this condition is violated, then the VALUE statement fails; the set of value_elements is not assigned to the name, and any previous assignment remains in effect.

SAS allows ranges in place of the extended_integer (e.g, `VALUE zscale 1-2 = "Low"`, and other, more complex specifications). SASdecoder does not accept this. SAS also allows formats to be defined for character data (with a $ prefix on name); this is also not accepted by SASdecoder. And SAS allows the keyword `OTHER` in place of a value, to signify "all other values not covered"; this is not accepted by SASdecoder either. Most of these restrictions correspond to limitations in what can be translated to Stata or Stat/Transfer.

The values that can appear as *extended_integer* include the missing values `.` (sysmis) and `._` . Both of these are accepted, but are not allowed in Stata and are flagged with a notice in the Stata output (do-file). In Stat/Transfer, they are accepted, but may cause unexpected results, and a warning is included in the output. (`.` may be ignored; `._` may get interpreted as 0).

The format (or formoid) created by a VALUE statement can subsequently be associated with a variable using a FORMAT or ATTRIB statement. When doing this, a period must be appended to indicate a format – in this case a user-defined format. For example, if the above definition of region has been made, you can then assign it to a variable as follows:

```
FORMAT reg01 region.;
```

(You can also include width and decimals, as in `FORMAT reg01 region6.3;`, however, these features are ignored.)

The format (or formoid) created by a VALUE statement corresponds to a value label in Stata and Stat/Transfer. For Stata, a `label def` command is written in the do-file, and if an assignment was made (in a FORMAT or ATTRIB statement), then a `label val` command is written. In Stat/Transfer, both the definition and assignments (if any) are written within the schema file.

The format (or formoid) created by a VALUE statement is a permanent symbol. See the section on **Scope of Variables; Permanent vs. Local Symbols** for more on this matter.

Note that SASdecoder does not support the PICTURE statement, a related statement. But this is not something that can be translated to Stata or Stat/Transfer.

## The FORMAT statement

Syntax:

`FORMAT` [ *format_assignment* … ] `;`

where *format_assignment* is

*variable_list* [*format*]

and *variable_list* is composed of one or more (space-separated) variable names or wildcard elements. See the section on **Variable_lists** for more information about this.

*format* is a sequence of letters, underscores, or digits, along with a single period that appears after the start of the sequence. The period can be surrounded by digits, indicating a field width and decimals; one common form consists of only digits surrounding a period. The format may also be preceded by a dollar sign (with or without intervening space), indicating that the variables are character string. See the section on **Formats and Informats** for more details on the form of formats.

The FORMAT statement assigns a format to one or more variables.

In SAS, formats specify how values are represented in textual output. In SASdecoder, they are accepted but mostly ignored, except if the format is a value label that was previously defined in a VALUE statement.

If a variable in *variable_list* already has a format assigned, that assignment is replaced with the new one. If *format* is absent then the variables are assigned no format; any previously existing format assignments are removed.

If *format* includes a field width and the variable is character and has not yet had its storage length set, then the storage length will be assigned as the given field width.

As mentioned, in SASdecoder, the only significance to a format assignment is if the format is one previously defined in a VALUE statement. In that case, code is written into the Stata do file or the Stat/Transfer schema to assign the value label to the variables.

Note that in the VALUE statement, the format is written without the period; in the FORMAT statement, the period is required. For example,

```
PROC FORMAT;
VALUE sexfmt
 1 = "male"
 2 = "female";

DATA;
FORMAT sex sexfmt.;
```

Note that format assignments can also be made in an ATTRIB statement (q.v.).

## The INFORMAT statement

Syntax:

INFORMAT [ *informat_assignment ...* ] ;

where *informat_assignment* is

*variable_list* [*informat*]

and *variable_list* is composed of one or more (space-separated) variable names or wildcard elements. See the section on **Variable_lists** for more information about this.

*informat* is a sequence of letters, underscores, or digits, along with a single period that appears after the start of the sequence. The period can be surrounded by digits, indicating a field width and decimals; one common form consists of only digits surrounding a period. The informat may also be preceded by a dollar sign (with or without intervening space[33]), indicating that the variables are character string. See the sections titled **Formats and Informats** and **Informats as Input Specifications** for more details on the forms and uses of informats.

The INFORMAT statement assigns an informat to one or more variables. In SAS, informats specify how values are read in an INPUT statement. They can either be mentioned in an INPUT statement, or assigned prior to use in the INPUT statement. Such an assignment is made in an INFORMAT statement or an ATTRIB statement.

If a variable in *variable_list* already has an informat assigned, that assignment is replaced with the new one. If *informat* is absent then the variables are assigned no informat; any previously existing informat assignments are removed.

If *informat* includes a field width and the variable is character and has not yet had its storage length set, then the storage length will be assigned as the given field width.

If a variable is assigned an informat and an informat also appears for that variable in an INPUT statement, the one in the INPUT statement takes precedence. If a variable is assigned an informat and appears without an informat in an INPUT statement, then it is taken as having list input. That is, the fieldwidth mentioned in the assignment (in the INFORMAT or ATTRB statement) is ignored. However, the decimals specification, if any, from that informat assignment, does get used. (That last point is in contradiction to some SAS documentation; however, it seems to be actual SAS behavior, and SASdecoder has been made to conform with this.)

SAS uses assigned informats (if informats are absent in the INPUT statement) to apply informat-style data-reading to list input. SAS documentation refers to this as mixing list and formatted input. In SAS, certain specialized informats can be applied this way, however SASdecoder ignores them for the most part, except to warn you of unsupported informats, and, as mentioned, to make us of any decimals specification, that may be present. One use of this feature that SASdecoder does accommodate is to pre-assign character informats and then to omit them in the INPUT statement:

```
INFORMAT lastname $15.;

INPUT lastname;
```

This is equivalent to…

[33] The effect of an intervening space, in say `INFORMAT abc $ 12.;` is that the dollar sign causes abc to have its type defined as character; abc is then assigned the `12.` informat, which is properly numeric, but can safely serve for character data. Without the space, the `$` also defines abc as character type, but its assigned informat is $12. – a true character informat.

```
INPUT lastname: $15.;
```

Note, however, the use of the colon (`:`) format modifier; this will be explained under **Format Modifiers**.

Also note that informat assignments can also be made in an ATTRIB statement (q.v.).

## The LENGTH statement

Syntax:

`LENGTH` *length_declaration ...* `;`

where *length_declaration* is one of the following:

*variable_list* [`$`] *integer*
`DEFAULT =` *integer*

where *variable_list* is composed of one or more (space-separated) variable names or wildcard elements. See the section on **Variable_lists** for more information about this.

The presence of a dollar sign signifies that the variables are of character type.

The LENGTH statement sets the storage lengths of variables.

For the *variable_list* form of the *length_declaration*, the integer must be in the range of 1-200 for character variables, and 2-8 for numeric variables. For character variables, only the first occurrence of a LENGTH statement (or equivalent ATTRIB statement) is used; any subsequent occurrence is flagged and then ignored.

In SASdecoder, the length setting for numeric variables has no effect.

The DEFAULT = *integer* form is intended for changing the default storage length for numeric variables (it is otherwise 8). SASdecoder parses this but does nothing with it.

Storage length can also be made set in an ATTRIB statement (q.v.).

## The ATTRIB statement

Syntax:

`ATTRIB` *specification_segment …* `;`

where *specification_segment* is

*variable_list attribute_specs*

where *attribute_specs* is one or more of these:

`FORMAT =` *format*
`INFORMAT =` *informat*

```
LABEL = "label"
LENGTH = [$] integer
```

The ATTRIB statement sets various attributes for one or more variables; it is virtually a FORMAT, INFORMAT, LABEL, and LENGTH statement all rolled into one[34], though you don't need to set all of these attributes. It is functionally equivalent to a set of corresponding FORMAT, INFORMAT, LABEL, and/or LENGTH statements. Thus, for example,

```
ATTRIB clname LENGTH = $ 20 LABEL = "client name";
```

is equivalent to...

```
LENGTH clname = $ 20;
LABEL clname = "client name";
```

See the **FORMAT**, **INFORMAT**, **LABEL**, and **LENGTH** statements for more on how each of these statements function.

Typically, *variable_list* is a single variable name; however a (space-separated) list of variable names or wildcard elements is permitted. See the section on **Variable_lists** for more information about this. (The possibility of multiple names and wildcards allows the assignment of the same variable label to multiple variables – probably not something you would want to do.)

SAS may exhibit a peculiar restriction. It allows the wildcards _all_ and _numeric_ (and possibly _char_ and _character_) in the first segment of an ATTRIB statement, but not in later segments. Thus,

```
ATTRIB _all_ length = $20 abc label = "hello";
```

is accepted, but

```
ATTRIB abc label = "hello" _all_ length = $20;
```

is not.

This behavior is not emulated in SASdecoder.

[34] It does not have the capability of DEFAULT = *integer* of the LENGTH statement.

# Input Specifications

This section describes what may occur under an INPUT statement.

## Input Specifications – Introduction

Before proceeding, it will be useful to consider some concepts relevant to data-reading.

SAS accepts four types of data input: column, formatted, list, and named. Of these, SASdecoder accepts three types: column, formatted, and list. (Named input cannot be done by either Stata or Stat/Transfer; thus it is out of the scope of what SASdecoder attempts to cover.) To understand these data-input methods and how they differ, it is helpful to review the concepts of field, field width, and storage length. These were explained under **Fundamental Data-Reading Concepts**, but, to reiterate briefly, a field is a segment of text in the raw data file that holds a textual representation of the value of a variable for a particular observation. Every field has a particular starting position and number of characters, the latter being the "field width". These characteristics may or may not be the same for each observation, i.e., they may be fixed or they may vary.

Column and formatted input can be characterized as having a fixed field width. Column input has a fixed starting position as well as a fixed field width. List input implies no particular field width; the field width is determined by the content of the raw data. In particular, for list input, reading of the data field begins by skipping spaces; then the value is read until a space or end-of-line is encountered. These characteristics are summarized in the following table.

| | **Starting Position** | **Field Width** |
|---|---|---|
| **Column Input** | Fixed | Fixed |
| **Formatted Input** | Can vary | Fixed |
| **List Input** | Can vary | Can vary; read until a space or end of line. |

For list or formatted input, the starting position is not intrinsically fixed by the input specification. But it is common practice for formatted input to be combined with pointer control (to be explained below) to fix the starting position, in which case, the result is equivalent to column input. Formatted input, without pointer control, can also establish implicitly fixed positions in subsequent variables. (See the `implicit` option, or the "use implicit positions" checkbox in the **Options** section for more on this matter.)

## Input Specifications – Briefly Enumerated

The features accepted in an INPUT statement will be described briefly here, and in more detail in a later section. This two-stage explanation will create some redundancy for the reader, though it will

be hopefully tolerable, and possibly beneficial. If any features seem to be absent or unexplained here, they should be covered in the later **Details** section.

As mentioned earlier, the INPUT statement accepts this form:

INPUT [*input_spec* ]...;

The forms of *input_spec* that SASdecoder accepts are:

*pointercontrol*
*variable_spec*
*grouped_format_list*

where…

*pointercontrol* consists of…

@*n*
@
+*n*
#*n*
/

where *n* is usually a non-negative integer literal (such as 12), but other forms are allowed, as will be explained later. Pointercontrol features specify the location of the next variable to be read.

*variable_spec* consists of

*limited_variable_list* [*fmt_modifier*] [$] [*position_spec_or_informat*]

where *limited_variable_list* is one or more variables or other forms that symbolically denote sets of variables. It is usually a list of one or more variable names, but certain other forms are accepted. See the section on **limited_variable_lists** for more on that matter.

*fmt_modifier* is an optional format modifier: either a colon (:) or an ampersand (&). The format modifiers will be explained below.

The optional $ indicates that the variable is a character string; it is necessary if the variable's type has not already been determined and you intend it to be character string.

*position_spec_or_informat* is either a *position_spec* or an *informat*, where…

*position_spec* specifies column input; it specifies the location of a fixed-position/fixed-width field. For example, 13-20 signifies column positions 13 through 20.

*informat* is an allowable SAS informat (example 10.) – for formatted input, which implies a fixed field width (in the absence of a colon format modifier), but not necessarily a fixed starting position. This will be described in detail in the **Formats and Informats** and **Informats in Input Specifications** sections.

Either a *position_spec* or an *informat* may appear, but not both.

*position_spec_or_informat* can also be entirely absent; this specifies list input, corresponding to "free format" in Stata, or "format delimited spaces" in Stat/Transfer. (List input is supported by Stat/Transfer, provided that the schema consists exclusively of list input.)

*grouped_format_list* consists of two lists, each in parentheses; first a list of variables, followed by a list of informats or pointercontrol elements. For example,

```
(name idno yearborn) ($12. $6. 4.)
```

The variables in the first list get matched with the informats in the second list, in sequence. Thus, the prior example is equivalent to

```
Name $12. idno $6. yearborn 4.
```

## Input Specification – in Detail

### Input Specifications for Pointer Control

Though the *variable_spec* is the central feature of input specifications (and *grouped_format_list* can be seen as a variant form of *variable_spec*), it is worthwhile to cover *pointercontrol* first, as it is often used in conjunction with all the other forms and it important that it be well understood.

As stated earlier, the specifications for *pointercontrol* are the following:

`@`*n*
`@`
`+`*n*
`#`*n*
`/`

Usually, *n* is a non-negative integer literal (such as `12`), but other forms involving parentheses, +, -, and identifiers are allowed. In particular, *n* is a limited_simple_integer_expression, which includes integer literals among other things. See the section on **limited_simple_integer_expressions** for more on these forms. Note that identifiers are allowed, but their use will result in indeterminate positions or line numbers – rendered as "###" in the output. Thus, while there is some latitude in what is accepted, usually only integer literals are meaningful to SASdecoder, and in most cases, only positive integer literals are meaningful and accepted.

Pointercontrol features specify the location (or "position") of the next variable to be read; a pointercontrol entity can be thought of as a preface to the variable that comes net. If there are multiple variables that follow (or a *grouped_format_list*[35]), only the first variable is affected directly.

The `#`, `@`, and + symbols may have whitespace between them and the *n*. Thus, you may have

```
@12
```

[35] However, any pointercontrol features embedded within that list will also be applied – after those that precede the list are applied.

or

```
@  12
```

Those are equivalent.

(See the section on **Whitespace** for more on that subject.)

The possibility of parenthesized expressions enables the use of negative numbers, at least grammatically. However, under the semantic rules, only the +*n* form allows negatives. Thus, you may have `+(-3)`, though the resulting translation is usually invalid. More on this later.

The `@`*n* feature specifies a location within the current data line; it means "move to position *n* in the current input line". (Position *n* is often referred to as "column *n*".) This is useful with list and formatted input. Thus, you can have...

```
input @12 def;
```

and the variable def will be read starting at position 12. This example uses list input. You can also have formatted input, which will read from a fixed-length field:

```
input @12 def 9.;
```

reads def as a 9-character field starting at position 12. It has the same effect as

```
input def 12-20;
```

*n* must be positive.

Note that if you combine pointer control with column input, the column input overrides the pointer control. Thus, in

```
input @12 def 14-22;
```

def is read from position 14-22; the `@12` is irrelevant.

Note that there is a SAS construct where a variable name appears directly after an `@` (or + or #):

```
input @k v;
```

In this form, SAS takes k to be a variable containing a value to be used as pointer control. This might be a data variable set by an earlier appearance in the INPUT statement:

```
input k 1-2 @k v;
```

or it might be set some other way, such as by an assignment:

```
k=20;
input @k v;
```

It could also be undefined:

```
input @k v;
```

(with no prior assignment to k), in which case a missing value is assigned to v.

This construct has no equivalent in Stata or Stat/Transfer. SASdecoder will accept these forms[36], but the result is an indeterminate position, and a warning is issued in the output.

The @ without a following number is a "trailing @", and is typically the final element in the INPUT statement.[37] Such a construct "holds the line"; a subsequent INPUT statement reads from the same line. For example,

```
input v1 1-4 @;
input v2 7-12;
```

would take v1 and v2 as on the same line (it would otherwise read v2 from the next line), and is equivalent to...

```
input v1 1-4 v2 7-12;
```

Note that, while the "hold the line" semantics are implemented by SASdecoder, its usual purpose in SAS is to create more complex specifications that cannot be directly translated to Stata dictionaries or Stat/Transfer schemas. That is, an occurrence of a trailing @ is likely to be part of a construct that doesn't translate.

A trailing @ on the final INPUT statement in a data step is possible, though its effect may be unpredictable (or data-dependent) and has no corresponding feature in Stata or Stat/Transfer.[38] If such a construct is found, a warning is issued on screen and in the output files.

There is also a SAS construct that uses a trailing double @ (@@ – with no space between the two @'s)[39], which is similar to a single trailing @; it holds the line for additional observations, and has no Stata or Stat/Transfer equivalent.[40] SASdecoder will recognize this form, but will raise a fatal error condition.

The +*n* feature specifies is another type of pointer movement. This says to go forward *n* places within the line, and will be mirrored by a _skip(*n*) entry in the Stata dictionary file. For

---

[36] It parses the @k form – not the assignment statement, as in k=20;.

[37] It is permitted to have multiple occurrences of @, with and without numbers, such as...

```
input v1 1-4 @3 @ @7 v2 @5 @ @4;
```

But only when there is a "lone" @ (one not followed by a number) that is not followed by anything else but possibly another @ (lone or not), is the statement regarded as having a trailing @. This is intended to mimic observed SAS behavior. Thus, the above example does qualify as having a trailing @. (The first lone @ does not count, as it is followed by v2. The second lone @ does qualify, even though it is followed by @4.)

[38] The effect, as presently understood by the author, is that at the end of each time through the loop of reading the data, SAS will not go to a new line. But then, depending on the raw data file content and the types of data items that occur subsequently (in reiterating the INPUT specifications), SAS may go to a new line because it runs out of data on the current line. The results can be peculiar and difficult to predict; they depend on the raw data. The author believes that this has no Stata or Stat/Transfer equivalent.

[39] Two @'s with space between them is permitted, as they are two separate single @'s – though this is not particularly useful.

[40] Stata's -infile- without a dictionary works that way, but -infile- with a dictionary cannot do it.

Stat/Transfer, there is no explicit translation, but it will contribute to the implicit position calculation, if that is used.

As with the `@` form, an identifier can be used in place of an integer literal, but the result is an indeterminate position.

Multiple + specifications in sequence will be combined and reduced to one specification. Thus,

```
+3 +7 +2
```

will be processed as if it were a single instance of `+12`.

Note that `+0` will be accepted but ignored. [41]

A negative number can be used as well; parentheses are necessary:

```
+(-4)
```

In SAS, this means to go backwards by 4 places. There is no equivalent Stata or Stat/Transfer feature. (`_skip(-4)` is not allowed in Stata.) However, if it is combined with other + entities such that the total will reduce to a non-negative value, then you will have a usable result. Thus,

+9 +(-4)

is reduced to `+5`.

Note that there is a "collapse@+" option. This will reduce an `@`*n* +*n* sequence to a single equivalent `@`*n*. Thus, when this option is activated, a sequence such as `@3 +4` will be treated as if there had been just `@7`. This can also help render a negative + specification as usable. Under this option, the sequence `@12 +(-4)` will be treated as `@8`. (This option is always "on" for Stat/Transfer output.) Under this option (or always for Stat/Transfer output), if the resulting position is non-positive, it will be coerced to 1.

The #*n* form specifies a move to another line of the input data. This is used where you have multiple lines of the raw data file corresponding to each observation – sometimes called "multiple lines per case" or "multiple physical records per logical record". In this situation, the number of lines per observation is a fixed quantity. For example, if it is 5, then lines 1-5 are for observation 1, lines 6-10 are for observation 2, lines 11-15 are for observation 3, and so on. Furthermore, and continuing this example, you usually expect specific variables on lines 1, 6, 11, etc., and other specific variables on lines 2, 7, 12, etc., and so on. (We are putting aside the possibility of the jumping to the next line that can occur with list or formatted input.) With this style of data layout, we refer to line 1, 2, 3, etc. of the raw data – meaning the first, second, third, etc. lines within each group of *m* lines, where *m* is the number of lines per observation (5 in this example).

Thus the *input_spec*

[41] SAS accepts +0, but, as expected, this has no effect. Stata does not accept _skip(0), so nothing is output.

```
#3
```

means that whatever follows, until the next line-number-control specification is encountered, is to be read from the third line of the raw data file – and every third line within groups of *m* lines.

Without any such line-number-control specifications, it is assumed that each observation is on one line of the raw data file. I.e., *m*=1.

As with the `@`*n* form, the semantic rules require that n be positive.

Again, as with the `@`*n* form, an identifier can be used in place of *n*, but the result is an indeterminate line number; the result is a "phantom" line number. (In this event, there is a first phantom line, and possibly a second and so forth – one for each instance of an identifier used in place of a n integer literal. They will appear as "###1", "###2", etc. in the output.)

`/` provides another means of specifying line-number-control; it signifies a move to the next line.

Still another method is to have multiple input statements; each one refers to the next line number (except where a trailing `@` is used; see below). Thus, in this example,

```
INPUT abc 3-7 def 12-17 / ghi 1-9 #4 jkl 1-7;
```

assuming that this is the first INPUT in the data step, then it is equivalent to the sequence,

```
INPUT abc 3-7 def 12-17 / ghi 1-9 / / jkl 1-7;
```

and is also equivalent to the sequence,

```
INPUT abc 3-7 def 12-17;
INPUT ghi 1-9;
INPUT;
INPUT jkl 1-7;
```

Assuming that this is the first set of INPUT statements in the data step, then this specifies that abc and def are on line 1, ghi is on line 2, and jkl is on line 4 – of every group. The number of lines per observation (*m* in the discussion above) is computed as the maximal line number indicated by any line-number-control specification – either explicitly in `#`*n* or implicitly using `/` or multiple INPUT statements, and regardless of whether any actual variables are specified on that line. In this example, *m* is 4, provided that this is the entirety of all input statements in the data step or that no additional lines are referenced.

If *m* turns out to be greater than 1, then SASdecoder will write

```
_lines(m)
```

in the Stata dictionary; in the Stat/Transfer schema file, sufficient `/` specifications are written to create the equivalent effect (i.e., *m*-1 occurrences of `/`). (In the Stat/Transfer schema file, all line-1 variables appear together, then a `/`, followed by all line-2 variables, and so forth. In the Stata dictionary, variables appear in the same order as in the SAS INPUT statements.)

This *m* is presumably the correct number of lines per observation; you should verify that it is correct by inspecting the raw data and possibly running the dictionary or schema; adjust as necessary.[42]

**Input Specifications for Variables**

Before proceeding, note that the input specifications will include a set of variable names – explicitly or implicitly. Typically, these names will be unique within a data step, but uniqueness is not required – in SAS, at least. (In SAS, if a variable name is mentioned multiple times in the INPUT statements of a given data step, then it will be filled-in with values multiple times within each data-reading cycle; the final value will prevail.) If SASdecoder encounters a name mentioned multiple times in the INPUT statements of a given data step, it will simply write the name a corresponding number of times in the output, and a warning will be issued[43]. Stata does not accept multiple occurrences of a variable name in a dictionary. Thus, in this circumstance, you should modify some of the names to distinguish them. (You can modify either the SAS code or the Stata dictionary.) In Stat/Transfer, multiple occurrences of a variable name are accepted; when Stat/Transfer runs the schema, some of the names will be automatically modified to insure uniqueness.

The specification for variables (*variable_spec*) consists of

*limited_variable_list* [*fmt_modifier*] [`$`] [*position_spec_or_informat*]

where *limited_variable_list* is one or more variables or other forms that symbolically denote sets of variables. It is similar to a variable_list, but with certain restrictions[44]. Briefly, it is one or more elements that are either variable names or ranges of variables or ranges of numerically-suffixed variable names such as `age1-age4`. See the section on **limited_variable_lists** for more on that matter.

*fmt_modifier* is an optional format modifier: either a colon (`:`) or an ampersand (`&`). The use of format modifiers will be explained below.

The optional `$` indicates that the variable is a character string; it is necessary if the variable's type has not already been determined and you intend it to be character string.

Actually the `$` and format modifiers may appear in any order. Thus, fmt_modifier and the `$` can be considered together as one optional grammatical unit. Also, `&` and `:` may appear repeatedly, though once is enough. But the `$` is allowed only once. If an informat, beginning with `$`, follows (in the

[42] In Stata, in the absence of a `_lines()` directive, can guess *m*, based on the maximal line number it sees. But unless SASdecoder sees an actual variable on that maximally-enumerated line, no mention of it will be made in the output except for this `_lines()` directive. Thus, it is generally necessary for the Stata dictionary to have the `_lines()` directive.

[43] Prior to version 6.3, SASdecoder required uniqueness of variable names in the INPUT statements of a data step.

[44] One restriction is that a colon wildcard is not allowed. But actually, a variable may be followed by a colon, but it is not taken to be a wildcard, but rather as a format modifier. The other prohibited features are the predefined wildcards: _ALL_, _CHARACTER_, _CHAR_, and _NUMERIC_.

*position_spec_or_informat* segment), then a separate `$` is not allowed. I.e., at most one `$` is allowed, including that which begins the informat.[45]

If *limited_variable_list* contains multiple variables, then the format modifiers or `$` will affect only the last variable mentioned (or implicitly mentioned).

*position_spec_or_informat* is either a *position_spec* or an *informat*, where…

*position_spec* specifies column input; it specifies the location of a fixed-position/fixed-width field. For example, `13-20` signifies column positions 13 through 20. The full syntax of *position_spec* will be described below.

*informat* is an allowable SAS informat – for formatted input, which implies a fixed field width (in the absence of a colon format modifier), but not necessarily a fixed position. This will be described briefly below, and in greater detail in the **Formats and Informats** and **Informats in Input Specifications** sections.

Either a *position_spec* or an *informat* may appear, but not both.

*position_spec_or_informat* can also be entirely absent; this specifies list input, corresponding to "free format" in Stata, or "format delimited spaces" in Stat/Transfer. (List input is supported by Stat/Transfer, provided that the schema consists exclusively of list input.)

*position_spec* is used for column input; it can be either of these forms:

*start_position* [*decimal_spec*]
*start_position* – *end_position* [*decimal_spec*]

where *start_position* and *end_position* are non-negative integers, with *start_position* <= *end_position*. If - *end_position* is absent, it is assumed to equal *start_position* (i.e., the field is of width 1). If *start_position* is 0, then both start_position and end_position are incremented by 1, following SAS behavior.[46]

*decimal_spec* is optional and can be either of these forms:

**.***decimal*
*decimal*

where *decimal* is a non-negative integer, specifying the number of implicit decimal places in the values to be read from the raw data field. Note that this is overridden if the data field contains an explicit decimal point.

Space is required before **.***decimal* or *decimal* (without the period), but is not allowed between **.** and *decimal*. That is, if a period is present, then *decimal* must follow immediately. The form without the period may be an archaic form, rarely seen nowadays.

[45] SAS allows multiple `$`, but exhibits bad behavior.
[46] This is behavior that was observed in SAS, though not seen in official documentation.

*decimal* can have leading zeroes. Thus,

```
14-20 .2
```

and

```
14-20 .002
```

are equivalent.

A decimal may be specified with a character variable, but it is ignored. (SASdecoder issues a warning; SAS ignores it silently.)

SAS accepts decimals that are greater than or equal to the width, such as...

```
input var1 2-4 .4 @12 var2 3.4;
```

but the corresponding Stata %infmts (`%3.4`) are not acceptable to Stata. (In this example, var1 uses column input; var2 uses formatted input.) If such a construct is encountered, the %infmts will be written anyway, and a warning will be written in the output. It is up to the user regarding how to handle the situation. (You can read without decimals, and then divide by a power of ten. But properly, you will need to detect cases that have explicit decimal points in the raw data, and bypass that division for those cases. Such an operation would require reading the field as a character string value and inspecting that value for the presence of a decimal point.)

This reveals a difference in how SAS and Stata operate. SAS apparently reads the number, and then divides by 10^*decimal* if no explicit decimal point occurs in the raw data. Stata, on the other hand, attempts to insert a decimal point into a textual copy of the raw data item if no explicit decimal point occurs. Then it attempts to read that result as a number. This can lead to problems (in Stata) if there are "short" numbers written with leading spaces.

One consequence of this difference is that, in SAS, you can have a field one character wide, which is read as a non-integer value. (One way to get this is to have a *decimal_spec* as part of a *position_spec* that omits - *end_position*.) This is impossible in Stata. This construct is, however, acceptable to Stat/Transfer; it can handle decimals that are greater than or equal to the width, including the reading of a non-integer value from a field with a width of 1 – apparently the same as SAS. The example shown above will result in a variable type of `(F3.4)` in the schema file.

*informat* is for formatted input. They are short constructs such as `F9.2` or `F6.` that specify a fixed width (9 and 6 in these examples), and other features about the data and how it is encoded. SAS Informats comprise a vast set of possibilities, though only a small subset is accepted by SASdecoder. This subject will be explained under **Informats in Input Specifications.**

### Grouped_format_lists

A *grouped_format_list* is a shorthand way of notating a set of *variable_specs* and possible *pointercontrol* and *informats*. It consists of a list of variable names and a corresponding list of other feature that specify how the variables are to be read. Each list is set within parentheses; thus, a grouped_format_list has the form:

(*variable_list*) (*format_list*)

See the section on **variable_lists** for a complete description of what may appear as *variable_list*. Briefly it is a list of one or more variable names, with the possibility of ranges and wildcards. Note that this includes the full set of variable_list forms – not the limited set. Thus, forms that are not allowed in a variable_spec (wildcards and ranges) are allowed here. Except for numeric-suffix ranges, it is doubtful that you would want to use these additional forms in a grouped_format_list, however, numeric-suffix ranges are commonly used, as in

```
(age1-age15) (4.)
```

*format_list* comprises a variety of things including informats; thus the name does not suggest the full range of what goes in there. Roughly, it includes all the things that may precede and follow a variable name in the "regular" part of an INPUT statement: *pointercontrol*, *fmt_modifier*, `$` *position_spec_or_informat*. There are some other differences, as will be explained.

Formally, *format_list* is a sequence of one or more *format_list_element*, where *format_list_element* is…

[*preceders*] *follower*

*preceders* is optional and comprises the same features as *pointercontrol* (q.v.) with the exception that the trailing (or lone) `@` is not allowed.

*follower* consists of the following form:

[*fmt_modifier1*] [`$`] [*multiplier* [*fmt_modifier2*]] [*position_spec_or_informat*]

This should look like a *variable_spec*, minus the variable (or the *limited_variable_list*, to be precise), plus the insertion of the optional *multiplier* and possible second *fmt_modifier*. As with a variable_spec, the `$` can actually be before *fmt_modifier1* (the first one) or intermingled with it, but only one `$` is allowed. It indicates that the variable is character type, in case it is not already established as such. *fmt_modifier2* (the one after *multiplier*) is allowed only if there is a multiplier (to be explained shortly), and a $ is not allowed in that position. *position_spec_or_informat* is the same as described above under variable_spec. If a multiplier is present, then *position_spec_or_informat* is required.

*multiplier* is a distinct feature. It consists of an integer literal followed by an asterisk, as in `4*`, and it indicates that the subsequent *position_spec_or_informat* is to be repeated as many times as indicated by the integer literal. Thus, `(4* 8.)` is equivalent to `(8. 8. 8. 8.)`. If a multiplier is present, a *fmt_modifier* (but not `$`) may follow, but the multiplying effect applies to the subsequent *position_spec_or_informat*. Note that, while a `$` is not permitted after a *multiplier*, an informat that begins with `$` is permitted as is commonly seen:

```
(jobcode1-jobcode4) (4* $8.)
```

If *multiplier* is present, then *fmt_modifier1* affects only the first instance of the expanded *position_spec_or_informat*; *fmt_modifier2* affects all instances of the expanded

*position_spec_or_informat*. That is, *fmt_modifier2* is subject to the *multiplier*; *fmt_modifier1* is not[47]

(The prohibition against a `$` after a multiplier is a SASdecoder feature; SAS allows it, however a variety of odd behaviors – read bugs – occur. Note also, that, while it is permitted to have a multiplier followed by a *position_spec* (e.g., `4* 12-19`), it would be rare to actually use such a construct, as it would require that a multitude of variables are to be read from the same positions – that is, they would get the same values. SAS also allows a form such as `2*3`, which is interpreted as `3 3`, which specifies a single-character field at position 3 with 3 implied decimals. SASdecoder does not follow this behavior. Finally note that SAS actually accepts multipliers is a wider variety of places such as in the *variable_list* and in the regular part of an INPUT statement (i.e., outside a grouped format list); SASdecoder does not accept these features, as they are probably unintended and may exhibit odd behavior.)

The reason for denoting the *format_list* components as *preceders* and *followers* is that they are the features that precede and follow a variable name in the regular part of an INPUT statement. The preceders denote the location of the field, and the followers modify the reading of the characters. This is important in determining how these elements align with the variable names in the first part of the grouped_format_list, as a grouped_format_list gets interpreted as an equivalent sequence of regular input components. You can think of the process of interpreting a grouped format list as that of taking the variables and inserting them between the appropriate preceders and followers. For example, in…

```
(housenumber street city) (@14 4. $25. @40 $20.)
```

`housenumber` is matched with `@14 4.`
`street` is matched with $25.
`city` is matched with @40 $20.

But note that the pointercontrol features apply as if they precede the variable names. Thus, this example is equivalent to…

```
@14 housenumber 4. Street $25. @40 city $20.
```

Another important concept is that the *format_list* functions as a loop; its elements get cycled through as often as needed. Thus, in matching the *format_list* elements to the variables, if you reach the end of the *format_list*, (that is, there are fewer *format_list_elements* than variables) then you start over at the beginning. Thus,

```
(street city county) ($25. $20.)
```

is equivalent to

```
(street city county) ($25. $20. $25.)
```

Though not explicitly stated above in the description of *format_list*, a "bare" set of *preceders* may appear at the end of the *format_list* – with no *follower*. Such *preceders* will apply to the first *format_list_element* if the *format_list* gets cycled. Thus,

```
(street city county) ($25. $20. +4)
```

[47] For example, `(: 3* $5.)` is equivalent to `(: $5. $5. $5.)`, whereas `(3* : $5.)` is equivalent to `(: $5. : $5. : $5.)`.

is equivalent to

```
(street city county) ($25. $20. +4 $25.)
```

But any such *preceders* have no effect beyond the bounds of the *grouped_format_list*. Thus, in

```
(street city) ($25. $20. +4) county $25.
```

the `+4` has no effect on `county`. More generally, anything remaining in the *format_list* once the final *follower*[48] is used will have no further effect.

A final note: SAS allows commas to be inserted in certain places in the *format_list*, supposedly to disambiguate certain constructs. However, the author has not been able to determine where they are supposed to be allowed, and what effect they actually have; they seem to have no effect. To date, no attempt has been made to accommodate commas, and the author would appreciate if any users can provide some feedback regarding how commas are supposed to be used and in what situations they have a real effect.

## Informats in Input Specifications

One possible component of the *input_spec* is an informat.

Informats describe how raw data fields are to be read.

As mentioned elsewhere, an informat is recognized by the fact that it either ends with a period or has a single period within it; it does not begin with a period. What lies to the left of the period may be a number –alone, or preceded by alphabetic characters. This number, if present, is known as the width specification, or *w* for short.[49] What lies to the right of the period, if present, must be a number, known as the decimal specification, or *d* for short. The informat may also be preceded by a `$` (either directly or with some space) to indicate a character variable. (The `$` is optional if the type of the corresponding variable is already established as character.)

Examples:

```
9.
9.2
F9.2
$12.
```

Informats can be used in two contexts: in an INPUT statement, or as assigned to a variable (in an INFORMAT or ATTRIB statement).

The use of informats within an INPUT statement (following a variable name), usually implies fixed-width fields; it does not, by itself, imply a fixed starting position. Within an INPUT statement, informats are often combined with the `@`*n* pointercontrol to fix the starting position, as in...

---

[48] That's the final one in terms of it being matched to a variable – not necessarily the same as the final one in the list.
[49] In terms of lexical rules, what lies to the left of the period may have digits other than those that lie directly behind the period, as long as the informat starts with an alphabetic or a $, for example ABC12DEF6.2. But no actual informats of this kind exist.

```
@20 myvar $4.
```

which is equivalent to

```
myvar $ 20-23
```

(column input).

Note, however, that it is permitted, to omit the `@`*n* (or other) pointercontrol, in which case, the starting position may or may not be fixed; the starting position is wherever the preceding variable left off, which may or may not be in a fixed position. Such a variable may have an implicitly fixed position – if it is the first variable to be read from the given line (first item in the INPUT statement, or first item after a line-movement specification), or if the preceding variable has a fixed starting position (implicit or explicit) and fixed field width. See the `implicit` option or the "use implicit positions" checkbox in the **Options** section for more on this subject.

While the use of informats is often equivalent to column input, some informats allow additional features not available in column input.

In most cases, informats read a fixed number of characters. The exception is if you omit the width part of the informat, e.g., `$BZ.`, rather than, say, `$BZ4.` Then, for some specific informats, the result is list input. But this should be regarded as a deviant case.

The following informats are accepted by SASdecoder. This is a small subset of what is possible in SAS:

| **Informat** | **Interpretation** |
|---|---|
| *w*`.`*d* | standard numeric |
| `F`*w*`.`*d* | standard numeric[50] |
| `BZ`*w*`.`*d* | numeric, blanks as zeros |
| `$`*w*`.` | standard character |
| `$CHAR`*w*`.` | character, not trimming leading blanks |
| `$CHARZB`*w*`.` | character, binary zeros as blanks |

In these descriptions, *w* and *d* stand for integers – the width and implicit decimals, respectively. Thus, some examples are `8.2`, `F8.2`, and `$12.`

In all cases, the *d* is optional and defaults to 0. You can specify *d* as 0, which is equivalent to not specifying the *d*. Thus, for example, `8.` and `8.0` are equivalent.[51]

[50] This was not found in the documentation available to the author. SAS apparently accepts this as equivalent to the *w*`.`*d* format.

The *w* is optional for all but *w*.*d* and $*w*., though the results of omitting it are idiosyncratic. In particular, if the variable is character, and the informat is F, CHAR, or CHARZB, then the field width is taken to be the same as the storage length (which should already be established, possibly taking a default of 8); if the variable is numeric, and the informat is F or BZ, then the field width is taken to be 1. In all other cases, no field width is assigned, and the variable is read using list input.[52]

For the character informats, the $ may be separated from the rest of the informat.[53] Otherwise, no space is permitted within the informat.

The standard informats (*w*.*d*, F*w*.*d*, $*w*.) are the only ones that properly translate to Stata or Stat/Transfer. The others (BZ*w*.*d*, $CHAR*w*., $CHARZB*w*.) are accepted but are just translated as if they were a corresponding standard informat. In these cases, warnings are inserted into the output. But note that the $CHAR*w*. is not too far off, in that its only deviant feature is that it does not trim leading spaces. Stata cannot do that, but the difference is probably of minor significance. The BZ*w*.*d* and $CHARZB*w*. informats, on the other hand, have no proper Stata or Stat/Transfer equivalents, and their behavior can deviate significantly from their corresponding standard informats, depending on the raw data. Nonetheless, they are allowed, since the differences only occur on unusual forms of raw data.

SAS has a multitude of additional informats, but it is the understanding of the author that none of them are applicable to Stata or Stat/Transfer.

Informats will translate to these Stata %infmts:

| **SAS informat** | **Stata %infmt** |
|---|---|
| *w*.*d* | %*w*.*d*f |
| *w*. | %*w*f |
| F*w*.*d* | %*w*.*d*f |
| F*w*. | %*w*f |
| $*w*. | %*w*s |
| BZ*w*.*d* | %*w*.*d*f (with a warning) |
| $CHAR*w*. | %*w*s (with a warning) |
| $CHARZB*w*. | %*w*s (with a warning) |

The choice of s or f is, however, affected by the f_infmt and s_infmt options, or the "use f informats" and "use s informats" checkboxes. The warning is that the SAS informat is unsupported and the %infmt is the closest possible equivalent.

---

[51] The *d* part may have leading zeroes. Thus, F6.0002 is equivalent to F6.2, which is also equivalent to 6.0002 and 6.2. Note that while the *w*.*d* informat may look just like the textual representation of a floating point number, it differs in this respect. Also, *d* is permitted on the character informats as well, but 0 is the only value allowed there.

[52] This is observed SAS behavior, but not seen in documentation.

[53] The $ is really a separate token, indicating, character data. In effect, the informat itself is *w*., CHAR*w*., or CHARZB*w*., but these tokens must be preceded by $ if the variable is not already defined as character string.

In Stat/Transfer, a simple fixed-width field is specified.

### Format Modifiers

The `:` and `&` format modifiers may occur after *limited_variable_list* in an *input_spec*. They are used with formatted input, though `&` can also apply to list input. If they appear after a multitude of variables, they apply to the last variable only.

`:` specifies that the field is to be treated as list input, even if there is a format specification. Thus, blanks are skipped before reading data, and the field extends until a blank is encountered or the end of the line is reached. Again, as in list input for character data, the actual number of characters stored is fixed – and it might possibly be fixed in the informat that follows. Consider…

```
INPUT lastname: $15.;
```

This specifies that lastname is to be read as list input. If it hasn't been established already, then the type is character of length 15.

`&` specifies that character values may include single blanks; this requires that the end of the field is marked by two or more blanks to distinguish it from embedded blanks. This input scheme is not supported in Stata or Stat/Transfer (to the author's knowledge). SASdecoder will take note of it and issue a warning that it is unsupported.

## Storage Length Revisited

An important concept in data management is storage length: the space allocated to the variable in your data-handling facility (SAS, Stata, etc.). This was covered under **Storage Length** in the **Fundamental Data-Reading Concepts** section, but there are additional issues that should be considered in the context of input specifications.

For character string variables, the storage length is usually a fixed quantity that you declare, such as in

```
LENGTH myvar $ 20
```

in SAS, or

```
str20 myvar
```

in Stata. For numeric quantities in SAS, the storage length is usually coupled with the data type and may or may not be under user control; we won't say much about that at this point (though see more about this under the LENGTH and ATTRIB statements).

In SAS, the storage length of a character string variable may be set in several ways, such as with the LENGTH, ATTRIB or INFORMAT statements, and is set the first time a character string variable is mentioned in one of these statements. Thus, these statements should be used prior to the INPUT statement. If no storage length has yet been set for a given variable by the time you reach an INPUT statement, it will be set as the length implied by the field width in a column or formatted *input_spec*. On the other hand, with list input, given no previously-set storage length, a character

string variable will be given a default storage length of 8. Thus, a list-input character string variable specification such as

```
somevar $
```

where somevar has not previously had its length specified, is given a length of 8, and translates to the Stata construct,

```
str8 somevar %s
```

or the Stat/Transfer schema construct,

```
somevar (A8)
```

Note that in this situation, 8 is the storage length – the maximum number of characters that can be stored in somevar. The number of characters read from the raw data will depend on the data itself; it may be less than or greater than 8. If another variable follows this in the INPUT statement and it does not have a fixed starting location, then its starting location is wherever somevar left off, and may differ from one observation to the next.

It may be tempting to try to adjust the length using informats in the *input_spec*, but informats imply a fixed-length field. In other words, a lone `$` (to indicate a character string variable) is not the same as a `$8.` informat. Thus,

```
somevar $8.
```

specifies a fixed-width (not list input) 8-character field; the storage length will also be 8 if it has not already been set to something else. (The starting position may or may not vary, depending on other circumstances.)

# Content of the Output Files

This section will briefly describe the content of the various output files.

## Content of the Output File – Stata

The output Stata dictionary will start with a `dictionary` line, which will mention the raw data file name if it was specified. This is followed by some commented information regarding how the file was created.

If the filename contains special or illegal characters, then it is bound in quotation marks. And if it contains illegal characters, a warning message is written. The special characters are those that are allowed in filenames but, due to possible ambiguity in the command syntax, must be bound in quotation marks. They are...

```
, ^ & = ; \ : /
```

plus space and tab.

The illegal characters are those that are not allowed at all in filenames. They are...

```
" * | < > ?
```

These classifications are based on Windows practices. They may not exactly be correct in other systems.

Further note that both `\` and `/` are classified as special. In Windows, `/` is actually illegal, but is regarded as merely special, due to the fact that Stata treats it as a directory separator. `\` is not allowed in the filenames per se, but does occur in file specifications as a directory separator. (In some Windows utilities, `/` will function as a directory separator.) In Stata, the directory separator can be either `\` or `/`; they are equivalent. But also note that in Stata, `\` can often cause problems (though not in dictionaries) which can be remedied by switching to `/`. This is a Stata feature, not an OS feature. (In some Stata contexts, problems with `\` can be fixed by substituting `\\` , but `/` is generally safe and effective.)

Also note that the directory separator is `\`, while it is `/` in other systems (Unix and Mac). But within Stata, it does not matter which you use.

The `:` is, similarly, not allowed in filenames per se, but is allowed in a full file specification, to indicate a device, as in `d:\myfile` – and only in such a construction. SASdecoder does not go so far as to distinguish this use of `:` from illegal ones.

If the `"` (quotation mark) character were to occur within a filename, then properly, compound quotes would be needed. This is not done, but there is no real need for them since the quotation mark is illegal in filenames anyway.

After this header sections, there is the section containing the variable specifications, each of which is on a separate line. Variables will appear in the same order as in the SAS INPUT statements.

There may be other relevant entries such as `_lines()`, `_line()`, `_firstlineoffile()`, and `_skip()`.

For the character types, a corresponding Stata `str` type will be specified, e.g., `str24`; there is no checking regarding the length limit, as this depends on your version of Stata. In general, there is no guarantee that you will generate a valid Stata dictionary. But for reasonable and valid SAS code, you will usually get a valid Stata dictionary.

For numeric types, the default type is float. However, this is subject to control by certain switches or options. See the discussion of the `doubles` option or the "use doubles" checkbox, and the `inttypes` option or the "use integer types" checkbox in the **Options** section for information on how these operate. Additionally, under the `inttypes` option or the "use integer types" checkbox, the type will be byte, int, or long, according to the field width, using the smallest type possible for the given width. This option is for situations where you know that the raw data contain only integers (no actual decimals) in these sorts of fields. There is generally no guarantee that this is the case, but in many common situations, you may know, from inspecting the data or your knowledge of the nature of the data, that it is true. On the other hand, it is possible that some of your items will fall under this category, while others will not. In that case you will need to adjust the types on a variable-by-variable basis. You can always just use float or double, and subsequently issue a `compress` or `recast` command.

There are two overriding rules, however:

1. if the field width is 9 or more (with or without implied decimals), then the type will always be double.
2. if the field width is 1 (with or without implied decimals), then the type will always be byte.[54]

Note that the `inttypes` option or the "use integer types" checkbox applies only to fixed-width fields – column or formatted input. It does not apply to list input, as it has no particular width.

In addition, you may get commented warnings alerting you to various conditions that you may need to attend to. Generally these warnings follow the data item (or other condition) that they refer to.

The do-file will contain an `infile` command to refer to the dictionary (if one was written during the same run that generated the do-file). In addition, if there were any value labels defined (via PROC FORMAT … VALUE statements in the SAS code), there will be the commands to define these and to attach them to the appropriate variables.

[54] That is, for numeric field of width 1, the `inttypes` option is virtually always "on". This rule makes sense, as, in Stata, a 1-character field will never be read as a floating-point value. SAS, on the other hand, can read a 1-character field as a floating-point value, using, for example, a 1.1 or 1.2 informat. Such informats will be translated by SASdecoder, but will yield illegal %infmts in the Stata dictionary.

## Content of the Output File – Stat/Transfer

The Stat/Transfer schema file begins with some commented header text, then a FILE statement and a VARIABLES statement, followed by a list of variable specifications. Finally, if appropriate, there will be a VALUE LABELS statement followed by value label definitions.

All line-1 variables will appear together, then a `/`, followed by all line-2 variables, and so forth (if there are multiple lines per observation.

As in the Stata output, you may get commented warnings alerting you to various conditions that you may need to attend to. Generally these warnings follow the data item (or other condition) that they refer to.

# Errors

In running SASdecoder, you may generate errors and warnings. When this happens, an error message will be issued, and in most cases, the location of the offending input text will be indicated; the line with the error will be written to the screen (or the Progress Log in the Windows Interface), and a caret indicator will appear pointing to the place in question, as in...

```
** Semantic error; [10,1]
inpiut
^
existing identifier of type command name expected
```
[55]

The pair of integers in the square brackets (`[10,1]`) indicates the line number and position of the error in the source file – in this case, line 10 and position 1. The final line (`existing identifier`…) is an explanation of the error.

Some conditions generate warnings; others are fatal. After a fatal error, processing stops, so only one fatal error can be detected at a time; however, you may get two or three error messages as a result of the same error.

The output file may include some warning messages (as commented text) – some in the header, and others in the body mixed in with variable declarations. For the latter type, the warnings occur immediately after the variable to which they refer. You can search for "`**--Warning:`" in the output to locate these.

[55] If there are any non-printing characters in the text, the indicator may not appear in the correct place. (Tabs may be okay.) There is a remote (probably impossible) situation where the line in question is no longer in memory, in which case the line and indicator will not be written. But in any case, the line and position will be reported, if it is an error in the form of the SAS file.

# Limitations

SASdecoder uses a fixed amount of memory for storing the information it gathers, so it is possible to overwhelm its capacity. (You will get a message indicating some sort of overflow.) It is believed that the capacity is generous enough to handle most SAS inputs, but if you do run into this problem please contact Essistant Software about it.

If this problem does occur, one possible interim solution would be to divide up the SAS file into pieces (or selectively comment-out parts of it), and later edit the results back together.

If you get “string table overflow”, you can try one of the `lookstr` options. (See the **Options** section.)

There is a vast range of possibilities in the few SAS statements that SASdecoder accepts, some of which do not translate to Stata or Stat/Transfer. But SASdecoder should have enough capability to be useful in most cases that are translatable.

## Notable Omissions

There are some SAS features that might be possible to include, but have not yet been accommodated. Prior to version 6.3, there were some notable omissions which have since been addressed (grouped format lists, variable ranges). Other possibilities include allowing more options on the INFILE statement (even if they have no effect on the resulting output).

Users are invited to suggest additional features for future development.

It is worth noting that the IF THEN and DO WHILE constructs can appear in conjunction with INPUT statements, but they do not translate into Stata dictionaries or Stat/Transfer schemas. So they are unlikely candidates for incorporation into SASdecoder. Some simple instances of these constructs, however, would have a corresponding action in a Stata do-file. But it is beyond the present goal of SASdecoder to include this capability. Note that where there is an IF THEN construct that merely filters the observations (determines which one to include), then you can comment-out the IF THEN construct in the SAS code and perform some filtering during the data-reading process. For example, you can place a corresponding `if` qualifier on your Stata `infile` command, or in the Stat/Transfer **Observations** panel using a case-selection ("where") statement. Or you can perform a corresponding filter at a later stage.