SAS Publishing

The FCMP Procedure

The Power to Know®

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2003. *The FCMP Procedure*. Cary, NC: SAS Institute Inc.

The FCMP Procedure

Copyright © 2003, SAS Institute Inc., Cary, NC, USA

All rights reserved. Produced in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19, Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, September 2003

SAS Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at **support.sas.com/pubs** or call 1-800-727-3228.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

The FCMP Procedure

Contents

USING THE FCMP PROCEDURE 3
- Introduction 3
- Function/Subroutine Declarations 4

SAS GLOBAL OPTION CMPLIB 5

FCMP SYNTAX 6
- PROC FCMP Statement 6
- Declaration Statements 8
- Program Statements 13

SPECIAL FUNCTIONS 15
- SOLVE 15
- DYNAMIC_ARRAY 20
- C Helper Functions 21

FUNCTION/SUBROUTINE PACKAGE VIEWER 24
- Getting Started 24
- Function/Subroutine Package Viewer Details 26
- Subroutine Finder 29
- Function/Subroutine Reader 30

EXAMPLES 32
- Example 1 - Creating a Subroutine and a Function 32

The FCMP Procedure

Using the FCMP Procedure

Introduction

The SAS Function Compiler (FCMP) Procedure allows users to create, test, and store SAS functions and subroutines for use by other SAS procedures.

The following SAS procedures allow the use of the functions and subroutines created by FCMP:

- CALIS
- COMPILE
- DISTANCE
- GA
- GENMOD
- MODEL
- NLIN
- NLMIXED
- NLP
- PHREG
- RISK DIMENSIONS
- ROBUSTREG
- SIMILAR
- SYLK

The FCMP procedure accepts a slight variant of the SAS DATA step language. Most features of the SAS programming language can be used in functions and subroutines processed by PROC FCMP. Additionally, this procedure enables the user to test functions and subroutines before using them in other procedures.

Function/Subroutine Declarations

Functions are equivalent to "routines" as used in any other programming language. They are independent computational blocks that require zero or more arguments. The subroutine is a special type of function that has no return value. All variables declared (created) within a function/subroutine block are local to that subroutine. PROC FCMP enables the user to declare, compile, and save subroutines and/or functions to SAS catalogs. Multiple subroutines and functions can be declared in a single usage of PROC FCMP.

For example, the following program defines a subroutine called **inverse**, which calculates a simple inverse; and the function **day_date**, which converts a date to a numeric day of the week. The subroutine begins with the SUBROUTINE statement, the FUNCTION begins with the FUNCTION statement, and each is completed with an ENDSUB statement.

```
proc fcmp outlib= sasuser.MySubs.MathFncs;

   subroutine inverse( in, inv );
      outargs inv;
      if in = 0 then inv = .;
      else inv = 1/in;
   endsub;

   function day_date( indate, type $ );
      if type = "DAYS" then wkday = weekday(indate);
      if type = "YEARS" then wkday = weekday(indate*365);
      return( wkday );
   endsub;

run;
```

These routines follow the SAS DATA step syntax. Any functions/subroutines already defined in the current FCMP procedure step as well as most SAS Datastep functions can be called from within these routines as well. In this case, the SAS Datastep function **weekday** is called by **day_date**.

In this example, the routines are saved to the catalog **sasuser.MySubs**, inside a package called **MathFncs**. A package is any collection of related routines as specified by the user. It is simply a way of grouping related functions/subroutines within the catalog. The OUTLIB= option in the PROC FCMP statement tells PROC FCMP where to store the subroutines it compiles, the LIBRARY= option tells it where to read in libraries (C or SAS) for use.

Note: Subroutine and function names must be unique within a package. However, different packages can have subroutines and functions with the same names. To select a specific subroutine when there is ambiguity, prefix the subroutine name with the package name and a period (.) For example, to get the **MthFncs** version of **inverse**, use **MthFncs.inverse**

SAS Global Option CMPLIB

The SAS Global option **CMPLIB** specifies where to look for previously compiled functions and subroutines. All procedures (including FCMP) that supports the use of FCMP functions and subroutines utilize this global option.

Instead of specifying the **LIBRARY**= option on every procedure statement that support subroutines and functions, the global option can be set and used by all procedures. This option follows the same syntax as the procedure **LIBRARY**= option:

CMPLIB = *library*
CMPLIB = (*lib1 lib2 lib3 ...)*
CMPLIB = *list**n**-list**n2***

Example of setting the CMPLIB option is as follows:

```
cmplib = sasuser.funcs;
cmplib = (sasuser.funcs work.functions mycat.funcs);
cmplib = sasuser.func1 - sasuser.func10;
```

In the following example, PROC FCMP is used to compile and store the **transform_log** and **simple_scale** subroutines in the **sasuser.similar** catalog. Then the CMPLIB global option is set and the functions are used by PROC SIMILAR:

```
proc fcmp outlib=sasuser.similar.simple;

   subroutine simple_normalize( sequence[*] );
      datalen = DIM(sequence);
      do i = 1 to datalen;
          sequence[i] = sequence[i] / 2;
          sequence[i] = .;
      end;
   endsub;

   subroutine simple_scale( targ[*], input[*] );
      datalen = DIM(targ);
      do i = 1 to datalen;
          input[i] =  input[i] / targ[i];
      end;
   endsub;
run;

options cmplib= sasuser.similar;

proc similarity data=workers out=out;
  id date interval=month;
  input  electric masonry  / normalize=simple_normalize
                             scale=simple_scale;
  target e_target m_target / normalize=simple_normalize
                             measure=simple_measure;
```

```
run;
```

FCMP Syntax

The statements used within the PROC FCMP step are

PROC FCMP *options*;

declaration statements;

program statements;

PROC FCMP Statement

PROC FCMP *options;*

The following options can be used in the PROC FCMP statement:

LIBRARY | INLIB = *library*
LIBRARY | INLIB = (*lib1 lib2 lib3 ...)*
LIBRARY | INLIB = *list**n**-list**n2***

specifies that previously compiled libraries are to be linked in for use. These libraries are created by a previous PROC FCMP step or by using PROC PROTO (for external C routines). Libraries are created by the OUTLIB= option and stored as members of a SAS catalog that have the type specified as CMPSUB. Currently, only subroutines and functions are read in when the LIBRARY= option is used.

Use the *libref.catalog* format to specify the two-level name of a library. The *libref* and *catalog* names must be valid SAS names that are not longer than 8 characters. (The *catalog* name is restricted to 7 characters or less on some operating environments, such as CMS.)

You can specify a list of files in the LIBRARY= option, and you can specify a range of names by using numeric suffixes. When you specify more than one file, you must enclose the list in parenthesis, except in the case of a single range of names.

```
Examples:
   proc fcmp library= sasuser.exsubs;
   proc fcmp library= (sasuser.exsubs work.examples);
   proc fcmp library= lib1-lib10;
```

OUTLIB | OUTCAT = *libname.catalog.package*
specifies the name of an output catalog package to which the compiled subroutines and functions are written when the PROC FCMP step ends. You specify the three-level name of a library entry using the formats *libref.catalog.paackage*. This option is mandatory when you use the OUTLIB= option and subroutines or functions are to be saved.

Note that only those subroutines that are declared inside the current PROC FCMP step are saved to the output file. Those loaded by using the LIBRARY= option are not saved to the output file. If no OUTLIB= option is specified, no subroutines that are declared in the current PROC FCMP step are saved.

```
Examples:
   proc fcmp outlib= sasuser.fcmpsubs.pkt1;
   proc fcmp outlib= sasuser.mysubs.math;
```

LISTCODE
requests that the compiled program code be printed. LISTCODE lists the chain of operations generated by the compiler. The LISTCODE output is somewhat difficult to read. For a more readable listing of the compiled program code, use the LISTPROG option.

LISTPROG
specifies that the compiled program be printed. The listing for assignment statements is generated from the operation chain. The source statement text is printed for other statements. Note that the expressions printed by the LISTPROG option do not necessarily represent the way that the expression is actually calculated because intermediate results for common subexpressions may be re-used, but they are printed in expanded form by the LISTPROG option. To see how the expression is actually evaluated, refer to the listing from the LISTCODE option.

LISTSOURCE
requests the printing of the source statements for the program.

LIST
specifies both the LISTSOURCE and LISTPROG options. Printing both the source and the compiled code and comparing the two listings of assignment statements is one way to verify that the assignments were compiled correctly.

LISTALL
specifies LISTCODE, LISTPROG, and LISTSOURCE.

Declaration Statements

ARRAY Statement

> **ARRAY** *arrayname* **[** *dimensions* **]** < */NOSYMBOLS | variables | constants | (initial-values)* >*;*

The ARRAY statement is similar to, but is not the same as the ARRAY statement in the SAS DATA step. The ARRAY statement associates a name with a list of variables and constants. The array name can then be used with subscripts in the program to refer to the items in the list.

The ARRAY statement that is used in PROC FCMP does not support all the features of the ARRAY statement in the DATA step. Implicit indexing of variables cannot be used; all array references must have explicit subscript expressions. Only simple array dimensions are allowed; lower-bound specifications are not supported. A maximum of six dimensions is allowed.

However, the ARRAY statement that is used in PROC FCMP does allow both variables and constants to be used as array elements. (Constant array elements cannot be assigned to). Although dimension specification and the list of elements are optional, one of them must be given. When the list of elements is not specified, or fewer elements than the size of the array are listed, array variables are created by suffixing element numbers to the array name to complete the element list.

Initial values for the array can be specified inside paranthesis.

The /NOSYMBOLS options specifies that the array of numeric values should be created without the associated element variables. In this case the only way to access elements is through array sunscripting. This can save a lot of memory if the user does not need to accesss the indivual array element varibles by name.

```
Examples:
   array spot_rate[3] 1 2 3;
   array spot_rate[3] (1 2 3);
   array y[4] y1-y4;
   array xx[2,3] x11 x12 x13 x21 x22 x23;
   array pp p1-p12;
   array q[1000] /nosymbols;
```

ATTRIB Statement

> **ATTRIB** *variables* **FORMAT=** *format* **LABEL=** *'label'* **LENGTH=** *n . . .;*

The ATTRIB statement (like the ATTRIB statement in the SAS DATA step) specifies format, label, and length information for variables. The ATTRIB statement supports a list of variables and their attribute definitions.

```
Examples:
   attrib x1 format= date7. label= 'variable x1' length= 5;
   attrib
      x1 format= date7. label= 'variable x1' length= 5
```

```
x2 length= 5
x3 label= 'var x3' format= 4.
x4 length= $2 format= $4.;
```

FORMAT Statement

FORMAT *variables format . . .* **DEFAULT=** *format;*

The FORMAT statement operates like the FORMAT statement in the SAS DATA step. The FORMAT statement controls the format that is used when printing variables.

```
Examples:
   format date monyy.;
   format gen 10.6 bignum e7.2 mychar $8.;
```

FUNCTION Statement

FUNCTION *funcname(arg1, arg2, ..., argn)*

OUTARGS *outarg1, outarg2 ... outargN;*

program-statements;

RETURN(*expression* **);**

ENDSUB;

The FUNCTION statement is a special case of the subroutine declaration that returns a value. The definition of a function begins with a FUNCTION statement and ends with an ENDSUB statement. Note that the CALL statement is not used to call a function.

Use the OUTARGS option to specify arguments from the argument list that the function should update.

Use the RETURN statement to specify the returned value of the function.

```
Example:
   function inverse( in );
      if in = 0 then inv = .;
      else inv = 1/in;
      return( inv );
   endsub;
```

LABEL Statement

LABEL *variable= 'label' . . .;*

The LABEL statement specifies a label of up to 255 characters for variables used in the program.

```
Examples:
   label date='Maturity Date';
   label bignum='Very very large numeric value';
```

LENGTH Statement

LENGTH *variables* **$** *n* . . . **DEFAULT=** *n;*

The LENGTH statement operates like the LENGTH statement in the SAS DATA step. The LENGTH statement controls the number of bytes that are used to store variables and specifies the length of the character variables.

```
Examples:
   length zzzz $ 200 zz $ 50;
   length c1a $3 c1b c1c $12 c1d $3 c1e c1f $12;
   length x14-x21 7;
```

STRUCT Statement

STRUCT *structure-name variable;*

The STRUCT statement declares (creates) structure types. The structures are defined in the C Language packages and when used, are declared in PROC FCMP.

```
Examples:
   struct DATESTR matdate;
   matdate.month = 3;
   matdate.day = 22;
   matdate.year = 1999;
```

SUBROUTINE Statement

SUBROUTINE *subr_name(arg1, arg2, ..., argn)*

OUTARGS *outarg1, outarg2 .... outargN;*

program-statements;

ENDSUB ;

The SUBROUTINE statement enables you to declare (create) an independent computational block of code that is callable from any program statement. To call a subroutine use the CALL statement.

Use the OUTARGS option to specify arguments from the argument list that the subroutine should update.

```
Example:
   subroutine inverse( in, inv ) group= "generic";
      outargs inv;
      if in = 0 then inv = .;
      else inv = 1/in;
   endsub;
```

Program Statements

Program statements are used in the main body of the PROC FCMP step to test the subroutine and functions declared. They are also used within the subroutine and function.

Most of the program statements that can be used in the SAS DATA step can be used in PROC FCMP. The following program statements are supported. Refer to *SAS Language: Reference* for the basic documentation of SAS program statements.

variable = *expression* **;**

variable + *expression* **;**

arrayvar[*subscript* **]** = *expression* **;**

CALL *subroutine-name (expression, expression, . . .)* **;**

STOP ;

ABORT ;

IF *expression* **THEN** *program-stmt;* <**ELSE** *program-stmt;* >

DO *program-statements;* **END;**

DO variable = *expression* **TO** *expression* <**BY** *expression*> **;**
    program-statement **; END;**

DO WHILE *expression* **;**
    program-statement **; END;**

DO UNTIL *expression* **;**
    program-statement **; END;**

GOTO *statement-label* **;**

RETURN ;

DELETE ;

SELECT <*(expression)*> **;**
    WHEN *(expr-1 <, expr-2, . . .>) program-statement* **;**
    <**WHEN** *(expr-1 <, expr-2, . . .>) program-statement* **;** >
    <**OTHERWISE** *program-statement* **;**>

PUT < *variable(s)* > < **@** | **@@** > **;**

Most of the preceding program statements work the same as they do in the SAS DATA step (as documented in *SAS Language: Reference*). However, there are some differences that should be noted.

- The DO statement does not allow a character index variable, and the IF statement does not allow a character test. Therefore, the following forms of statements are supported.

  ```
  do i=1,2,3;
  if 'this' < 'that' then ... ;
  ```

  But the following forms of statements are not supported.

  ```
  do i='one','two','three';
  if 'this' then ...;
  ```

- The PUT statement, which is typically used for program debugging in PROC FCMP, supports only some of the features of the PUT statement in the DATA step, and it has some new features that the DATA step PUT statement does not.
  - The PUT statement in PROC FCMP does not support line pointers, factored lists, iteration factors, overprinting, _INFILE_ , the colon (:) format modifier, or the special character $.
  - The PUT statement in PROC FCMP does not support subscripted array names are not supported unless they are enclosed in parenthesis. For example, the statement PUT (A[i]); prints the i-th element of the array A, but the statement PUT A[i]; results in an error message.
  - The PUT statement in PROC FCMP does not allow the asterisk (*) subscript, but an array name can be used in a PUT statement without subscripts. Therefore, the statement PUT A =; (when A is an array) is acceptable, but the statement PUT A* =; is invalid. The statement PUT A; prints all the elements of the array A. The statement PUT A=; prints all the elements of the array A with each value labeled with the name of the element variable.
  - The PROC FCMP PUT statement does support expressions inside of parentheses. For example, the statement PUT (SQRT(X)); prints the square root of X.
  - The PROC FCMP PUT statement does support the print item _PDV_ to print a formatted listing of all the variables in the program data vector. The statement PUT _PDV_ ; prints a much more readable listing of the variables than is printed by the statement PUT _ALL_ ;.
- The ABORT statement does not allow any arguments.
- The WHEN and OTHERWISE statements allow more than one target statement. That is, DO/END groups are not necessary for multiple WHEN statements, for example, SELECT; WHEN(exp1) stmt1; stmt2; WHEN(exp2) stmt3; stmt4; END;

Special Functions

There are a few special purpose functions automatically provided by the FCMP Procedure for convenience.

SOLVE

The SOLVE function computes implicit values of a function. The general form for using the SOLVE function is

```
answer = solve( "function_name",
                options_array,
                expected_value,
                arg1, arg2, ..., argn );
```

where

function_name	is the name of the function of interest.
options_array	is the array of options to the solve functions (See details below).
expected_value	is the expected value of the function of interest.
arg1, arg2, ..., argn	is a list of arguments.

The **solve** function finds the value of the specified argument that makes the expressions of the following form equal to zero.

```
expected_value - function_name ( arg1, arg2, ..., argn )
```

The argument of interest is indicated by a missing value (.), which appears in place of that argument in the parameter list. If successful, the returned value for this function is the implied value.

Options Array

The options array is used to control and monitor the root finding process. The options array can be missing (.) or can have up to five elements. The five elements in order are

initial value	What to use as a starting value for the implied value. The default for first call is 0.001. If the same line of code is executed again, the previously found implied value is used.
absolute criterion	The absolute value of the difference between the expected value and the predicted value must be less than this value for convergence. The default is 1.0e-12.
relative criterion	When the change in the computed implied value is less than this criterion, then convergence is assumed. The default is 1.0e-6.
maximum iterations	The maximum number of iterations to use to find the solution. The default is 100.
solve status	• 0 - Successful • 1 - Could not decrease the error • 2 - Could not compute a change vector • 3 - Max number of iterations exceeded • 4 - Initial objective function missing

An example of options array is as follows:

```
array opts[5] initial abconv relconv maxiter ( .5 .001 1.0e-6 100 );
```

where

- initial value (“initial”) = .5
- absolute criterion (“abconv”) = .001
- relative criterion (“relonv”) = 1.0e-6
- maximum iterations (“maxiter”) = 100

The solve status is the fifth element in the array, which can be displayed by specifying opts[5] in the output list.

Note: The names of the elements do not have to be those that were used in this example. For example, instead of “initial”, you could use “a”.

SOLVE Function Examples

This first SOLVE function example computes the *x* that satisfies the equation
$y = 1/sqrt(x)$.

```
proc fcmp;

      /* define the function */
   function inversesqrt( x );
      return( 1 / sqrt(x) );
   endsub;

   y = 20;
   x = solve( "inversesqrt", {.}, y, . );

   put x;
run;
```

Note: Functions and subroutines must be defined, then they can be used in the SOLVE function. In this example, the function **inversesqrt** is defined and then used in the SOLVE function.

In this second SOLVE function example, the subroutine **gkimpvol** calculates the **Garman-Kohlhagen** implied volatility for FX options by using the SOLVE function on the **garkhprc** function.

- The options_array is **SOLVOPTS**, which requires an initial value.
- The expected value is the price of the FX option.
- The missing argument in the subroutine is the volatility (sigma).

```
subroutine gkimpvol( n, premium[*], typeflag[*], amt_lc[*],
                     strike[*], matdate[*], valudate, xrate,
                     rd, rf, sigma );
   outargs sigma;

   array solvopts[1] initial ( 0.20 ) ;
   sigma = 0;
   do i = 1 to n;
      maturity = (matdate[i] - valudate) / 365.25;
      stk_opt = 1./strike[i];
      amt_opt = amt_lc[i] * strike[i];
      price = premium[i] * amt_lc[i];

      if typeflag[i] eq 0 then type = "Call";
      if typeflag[i] eq 1 then type = "Put";

      /*--- solve for volatility -------------------*/
      sigma = sigma + solve( "GARKHPRC", solvopts, price,
                             type, "Buy", amt_opt, stk_opt,
                             maturity, xrate, rd, rf, . );
   end;
   sigma = sigma / n;
endsub;
```

This third SOLVE function example defines the function **blksch**, by using the built-in SAS function, **blkshclprc**. The SOLVE function uses the **blksch** function to calculate the Black-Scholes implied volatility of an option.

- The options_array is **OPTS**.
- The missing argument in the function is the volatility (VOLTY).
- PUT statements are used to print the implied volatility (BSVOLTY), the initial value, and the solve status.

```
proc fcmp;
   opt_price = 5;
   strike = 50;
   exp = '01jul2001'd;
   eq_price = 50;
   intrate = .05;
   time = exp - date();
   array opts[5] initial abconv relconv maxiter
                 ( .5 .001 1.0e-6 100 ) ;
   function blksch( strike, time, eq_price, intrate, volty );
      return (blkshclprc( strike, time/365.25,
                          eq_price, intrate, volty ));
   endsub;
   bsvolty = solve( "blksch", opts, opt_price, strike,
                              time, eq_price, intrate, .);

   put 'Option Implied Volatility:'  bsvolty
       'Initial value: ' opts[1]
       'Solve status: ' opts[5];
run;
```

Note: SAS functions and external C functions cannot be used directly in the SOLVE function. They must be enclosed in a PROC FCMP function. In this example, the built-in SAS function BLKSHCLPRC is enclosed in the PROC FCMP function **blksch**, then **blksch** is called in the SOLVE function.

DYNAMIC_ARRAY

The DYNAMIC_ARRAY subroutine allows an array declared within a function to change size in a efficient manner:

```
array scratch[1];
length = 200;
call DYNAMIC_ARRAY( scratch, length );
```

This is very useful if a function needs a scratch area to do work in, but the size of that area depends on parameters passed to the function. In the following example a scratch array is created called *temp*:

```
function avedev_wacky( data[*] );

   length = DIM(data);
   array temp[1];
   call DYNAMIC_ARRAY(temp, length);

   mean = 0;
    do i=1 to datalen;
       mean += data[i];
       if i > 1 then temp[i] = data[i-1];
       else temp[i] = 0;
   end;
   mean = mean/length;

   avedev = 0;
   do i = 1 to length;
      avedev += abs((data[i]-temp[i])/2-mean);
   end;
   avedev = avedev/datalen;

   return( avedev );

endsub;
```

C Helper Functions

Several helper functions are provided to handle C-language constructs in PROC FCMP. Most C-language constructs must be defined in a catalog package created in PROC PROTO before the constructs are referenced or used in PROC FCMP.

The following helper functions have been added to extend the SAS language to handle C language constructs that do not fit naturally into the SAS language.

ISNULL C Helper Function

ISNULL determines whether or not a pointer element of a structure is NULL.

double ISNULL (*pointer-element* **);**

For example, the LINKLIST structure and function **get_list** are defined using PROC PROTO as shown below. The **get_list** function is an external C routine that generates a linked list with as many elements as requested.

```
struct LINKLIST {
    double value;
    struct LINKLIST * next;
};

struct LINKLIST * get_list( int );
```

The following FCMP code segment demonstrates using the ISNULL() helper function to loop over the linked list created by get_list() and print out the elements:

```
struct LINKLIST list;

list = get_list( 3 );
put list.value=;

do  while ( ^ISNULL(list.next ));
    list = list.next;
    put list.value=;
end;
```

The following output is produced by this run:

```
              The SAS System
              FCMP Procedure

LIST.value=0
LIST.value=1
LIST.value=2
```

SETNULL C Helper Function

SETNULL sets a pointer element of a structure to NULL.

call SETNULL (*pointer-element* **);**

Assuming that the same LINKLIST structure described above is defined using PROC PROTO, the SETNULL subroutine could be used to set the 'next' element to NULL:

```
struct LINKLIST list;
call SETNULL(list.next);
```

STRUCTINDEX C Helper Function

STRUCTINDEX enables access to each structure element in an array of structures.

call STRUCTINDEX (*struct_array, index, struct_element* **);**

The *index* parameter is a 1-based index as in most other SAS arrays.

Given that the following structures and function are defined using PROC PROTO:

```
struct POINT {
    short s;
    int   i;
    long  l;
    double d;
};

struct POINT_ARRAY {
    int             length;
    struct POINT * p;
    char            name[32];
};

struct POINT *  struct_array( int );
```

The following FCMP code segment demonstrates using the STRUCTINDEX() helper function to get and set each 'POINT' structure element of an array 'p' in the 'POINT_ARRAY' structure:

```
struct POINT_ARRAY pntarray;
struct POINT pnt;

/* call struct_array() to allocate array of 2 'POINT' structs */
pntarray.p = struct_array( 2 );
pntarray.length = 2;
pntarray.name = "My funny structure";
```

```
/* get each element using the STRUCTINDEX() call and set values */
do i = 1 to 2;
   call STRUCTINDEX( pntarray.p, i, pnt );
   put "Before setting the" i "element: " pnt=;
   pnt.s = 1;
   pnt.i = 2;
   pnt.l = 3;
   pnt.d = 4.5;
   put "After setting the" i "element: " pnt=;
end;
```

The output produced by this run is:

```
            The SAS System
            FCMP Procedure
Before setting the 1 element:  PNT {s=0, i=0, l=0, d=0}
After setting the 1 element:  PNT {s=1, i=2, l=3, d=4.5}
Before setting the 2 element:  PNT {s=0, i=0, l=0, d=0}
After setting the 2 element:  PNT {s=1, i=2, l=3, d=4.5}
```

Function/Subroutine Package Viewer

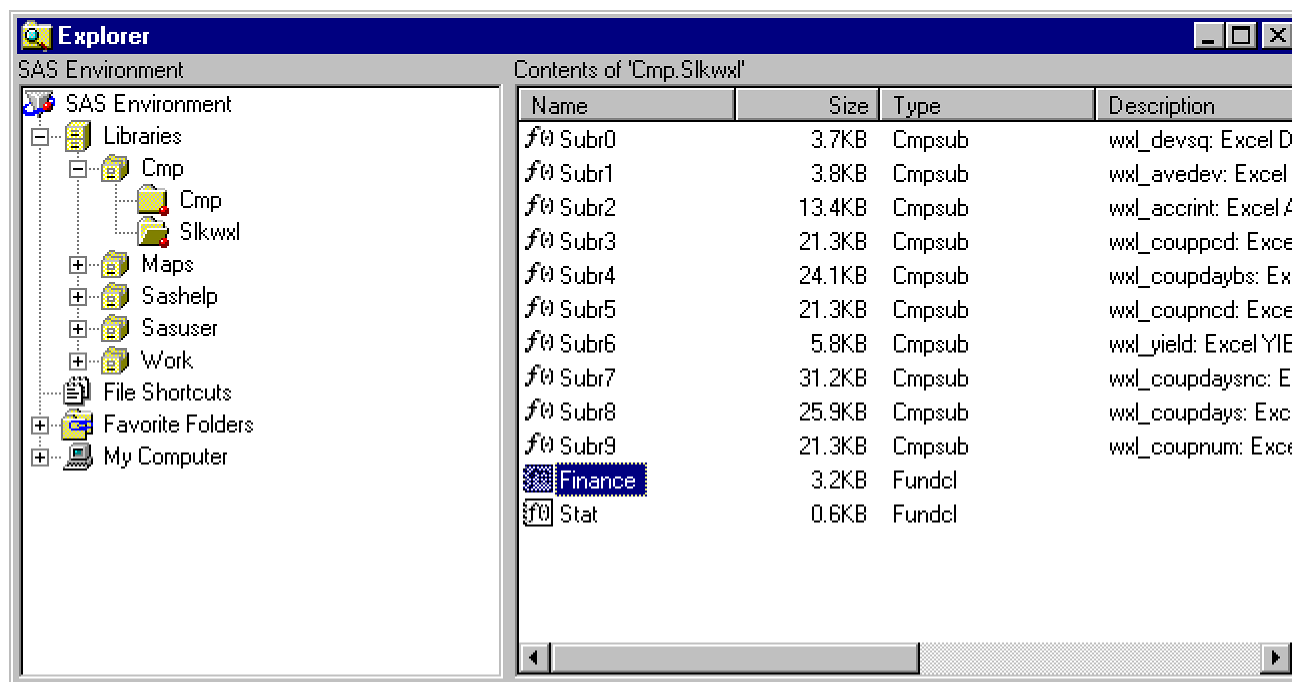
SAS Language functions and subroutines created in PROC FCMP are stored in SAS catalogs in *package declarations* (entry type FUNDCL). Each package declaration contains any number of functions and/or subroutines (entry type CMPSUB) as specified by the user. The Package Declaration Viewer displays all the routines in a package.

Getting Started

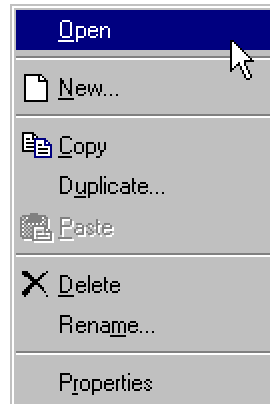
This example explains how to view a Function/Subroutine Package Declaration.

To view the routines in a function/subroutine package declaration, open the SAS Explorer to the SAS Catalog that contains the package you want to view. Then, select the package (catalog entry type FUNDCL) you want to open.

In this example, SASHELP.SLKWXL.FINANCE package is selected as shown in the figure below.

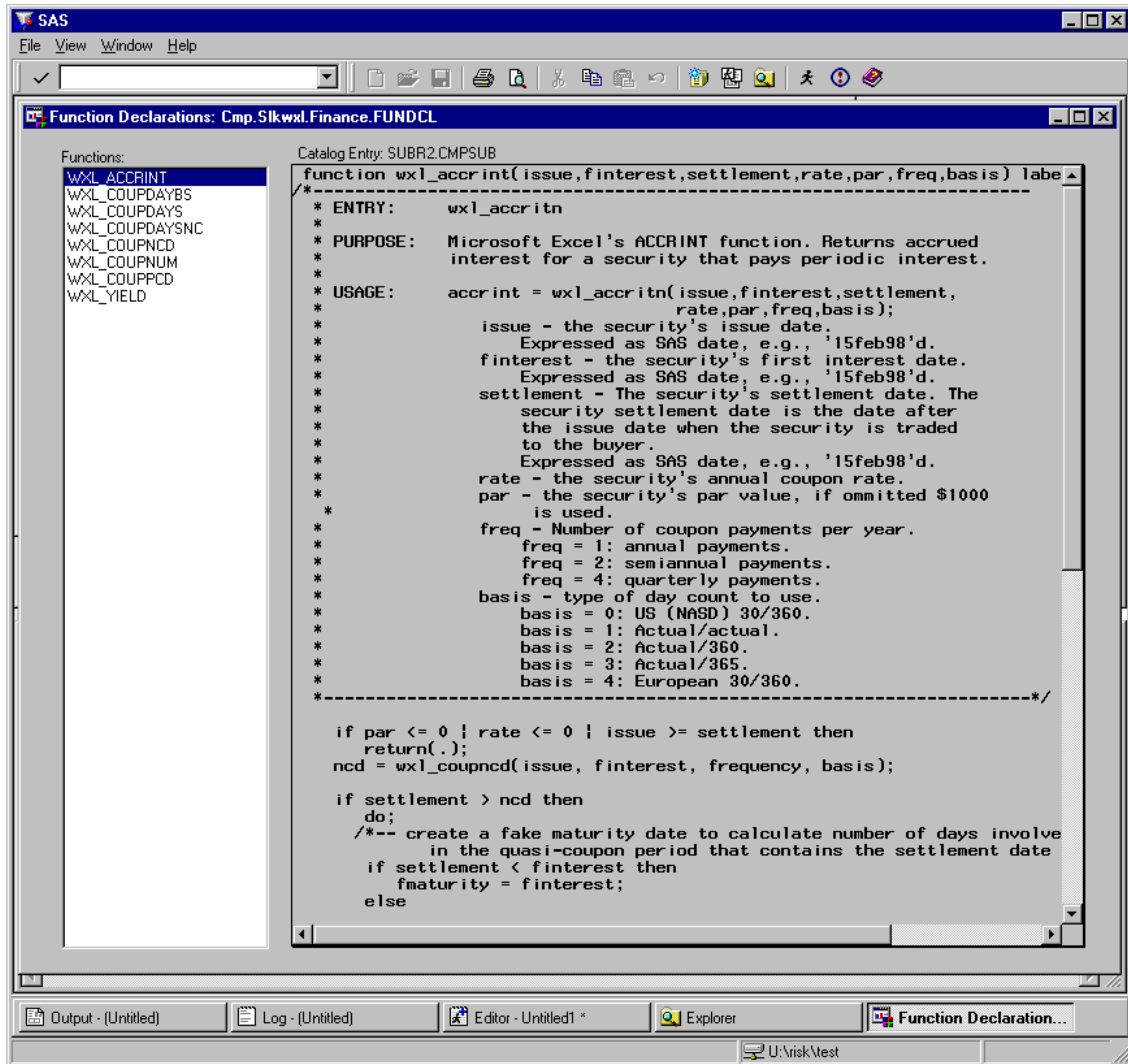


Double-click the package declarartion you want to view or right-click the package and select *Open* as shown in the figure below.



The Package Declaration Viewer opens. The viewer displays a list of routines contained in the package on the left. On the right, the selected (first) routine is displayed. Other routines can be displayed by selecting the routine name from the routines listed on the left.

In this example, the routine ACCRINTM_WXL is selected as shown in the figure below.



Function/Subroutine Package Viewer Details

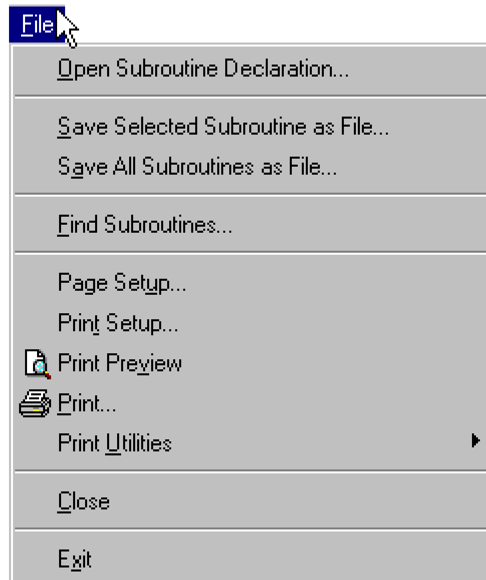
SAS Language functions and subroutines (routines) are stored in packages (SAS catalog entry type FUNDCL). The Function/Subroutine Package Viewer allows you to browse this package.

The following sections discuss the selections of the pull-down menus for the Windows NT version of SAS. If you are using a different operating environment, then the pull-down menus and their selections might vary.

The Function/Subroutine Package Viewer has the following pull-down menus: *File Menu*, *View Menu*, *Windows Menu*, *Help Menu*.

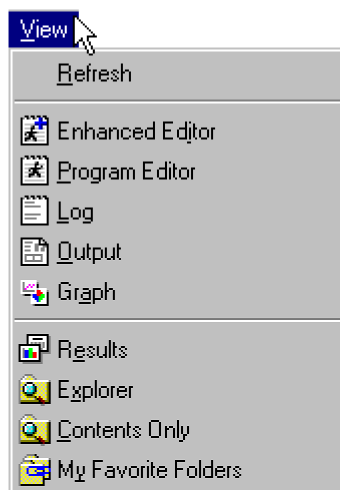
File Menu

The *File* pull-down menu contains the following selections that enable you to manipulate the contents of the viewer.



- Open Package Declaration - This selection enables you to replace the contents of the viewer with an existing function declaration.
- Save Selected Subroutines as File - This selection enables you to save the contents of the selected routine to a text file.
- Save All Subroutines as File - This selection enables you to save the contents of all routines to a text file.
- Find Subroutine - This selection enables you to search for existing routines using the Subroutine Finder.
- Print Preview - This selection is used to preview the contents of the selected routine.
- Print - This selection is used to print the contents of the selected routine.
- Page Setup - This selection is used to specify the page setup for printing the contents of the viewer. For example, to print the pages in portrait or landscape orientation.
- Print Setup - This selection is used to specify the print setup. For example, to select a printer.
- Print Utilities - This selection is used to set the printing options.
- Close - This selection closes the viewer.
- Exit - This selection closes the SAS session.

View Menu



This menu contains selections for accessing and viewing windows of the SAS System.

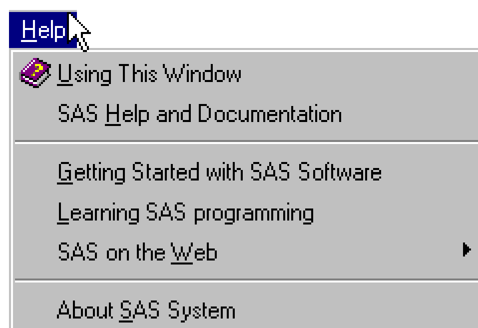
- Refresh - This selection is used to refreshes the contents of the viewer. The viewer should be refreshed whenever the Function Declaration is updated in order to view the most recent version.

Window Menu

This menu contains selections that enable you to control the display of windows and to access windows. To control the display of windows, you can minimize, cascade, and re-size windows, as well as tile the windows vertically or horizontally.

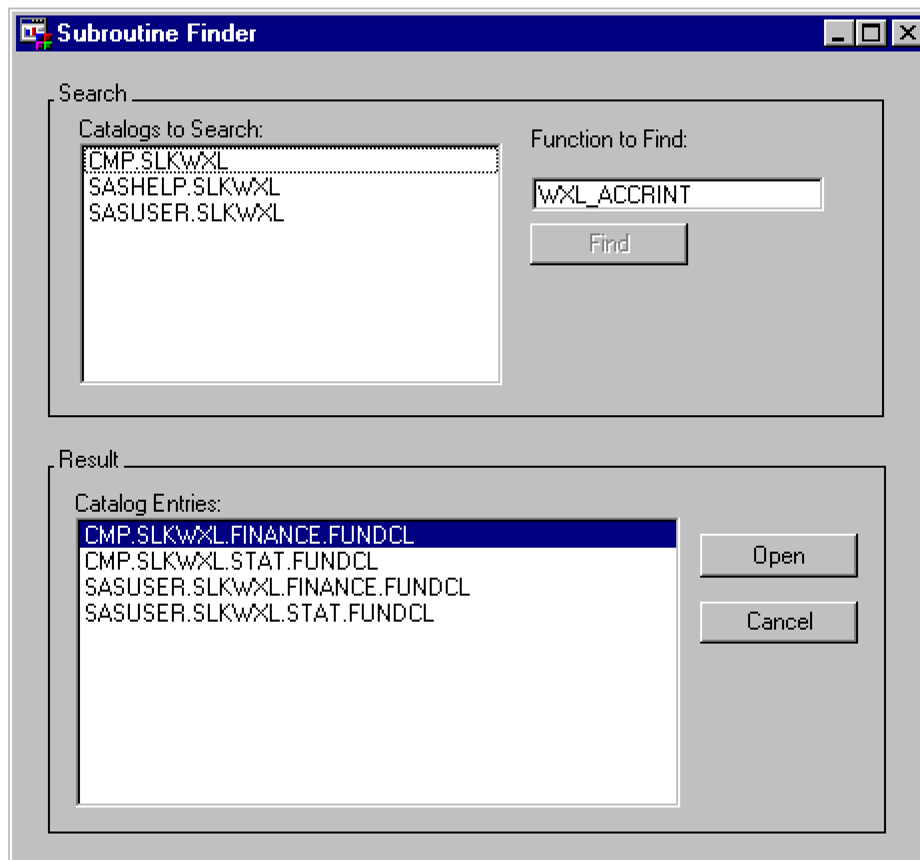
Help Menu

This menu enables you to access online help files.



Subroutine Finder

The Subroutine Finder Window searches for Package Declarations and SAS subroutines and functions. When the window is opened, all current SAS Libraries are searched for SAS Catalogs that contain Package Declarations (FUNDCL). The SAS Catalogs found to contain FUNDCL entries are listed in the *Catalogs to Search* list box. The FUNDCL entries are listed in the *Package Entries* list box. The *Function to Find* text box is initialized to the last viewed routine and the *Package Entries* list box is selected to the FUNDCL entry that declares this routine.



To subset the search to a particular SAS Catalog(s), select the catalog(s) in the *Catalogs to Search* list box. The *Package Entries* list box reflects this selection.

To further subset the search to a particular routine, type the routines name in the *Function to Find* text entry and hit the return key or click the *Find* button. The *Package Entries* list box reflects this selection.

Select the desired *Package Entries* list box entry the click the *Open* button to view the selected package entry or double click the package entry that you want to open in the viewer. BUG: when there is more than on FUNDCL in the catalog the FUNDCL is not selected.

Select the *Cancel* button to cancel the search.

Function/Subroutine Reader

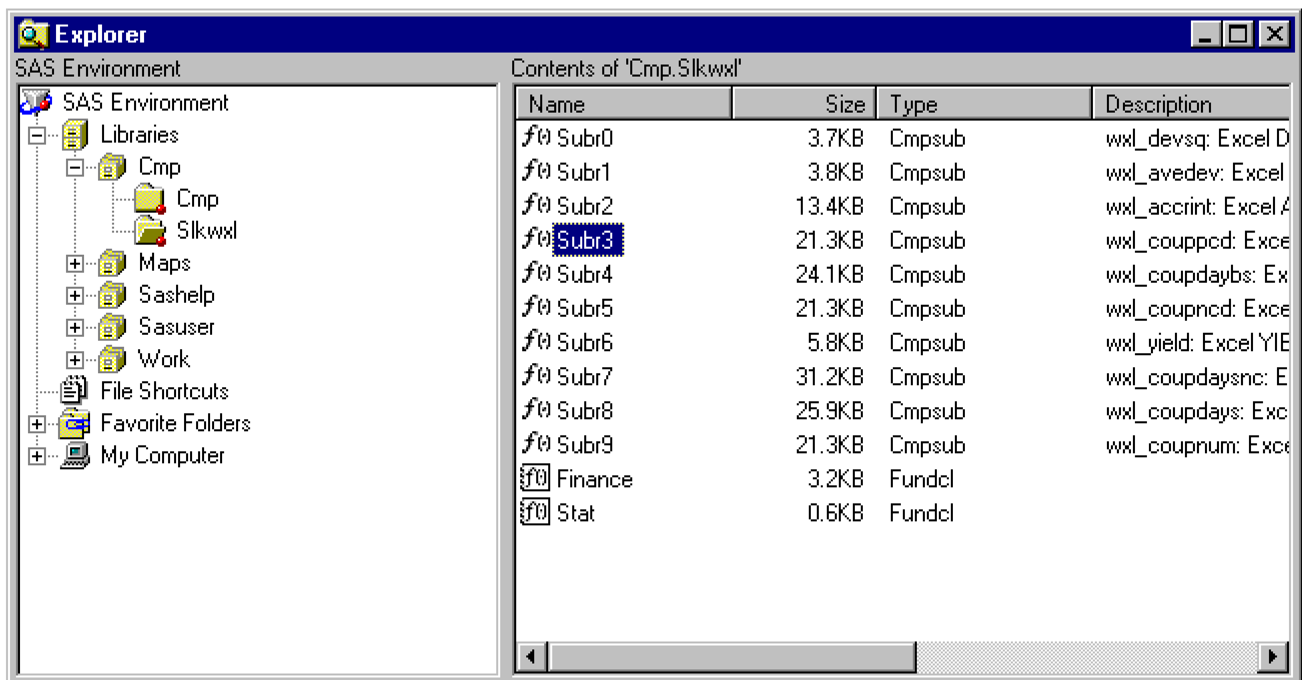
Indiviual SAS Language functions and/or subroutines (routines) are stored in SAS catalog entry type CMPSUB. The Subroutine Reader (reader) displays an individual CMPSUB catalog entry.

Getting Started

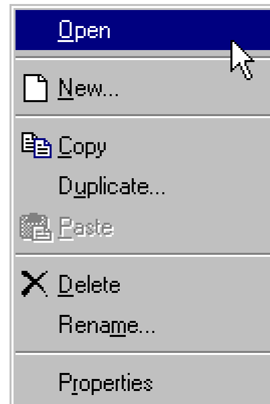
This example explains how to view CMPSUB catalog entries.

To view the routine in a CMPSUB catalog entry, open the SAS Explorer to the SAS Catalog that contains the entry you want to view. Then select the CMPSUB catalog entry you want to open.

In this example, SASHELP.SLKWXL.FINANCE.SUBR3 is selected as shown in the figure below.



Double-click the CMPSUB entry you want to view or Right-click the subroutine entry and select Open as shown in the figure below.



The Function/Subroutine Reader opens. The reader displays the selected routine.

In this example, the routine FACTDOUBLE_SLK is selected as shown in the figure below.

```
function wxl_couppcd( settlement, maturity, frequency, basis) label= "Excel COUPPCD";
/*--------------------------------------------------------------------------
 * ENTRY:     wxl_couppcd
 *
 * PURPOSE:   Microsoft Excel's COUPPCD function. Returns the previous
 *            coupon date before the settlement date.
 *
 * USAGE:     coupncd = wxl_coupncd(settlement,maturity,freq,basis);
 *               settlement - The security's settlement date. The
 *                   security settlement date is the date after
 *                   the issue date when the security is traded
 *                   to the buyer.
 *                   Expressed as SAS date, e.g., '15feb98'd
 *               maturity - The security's maturity date. The
 *                   is the maturity te when the security expires.
 *                   Expressed as SAS date, e.g., '10jun98'd.
 *               freq - Number of coupon payments per year.
 *                   freq = 1: annual payments.
 *                   freq = 2: semiannual payments.
 *                   freq = 4: quarterly payments.
 *               basis - type of day count to use.
 *                   basis = 0: US (NASD) 30/360.
 *                   basis = 1: Actual/actual.
 *                   basis = 2: Actual/360.
 *                   basis = 3: Actual/365.
 *                   basis = 4: European 30/360.
 *-----------------------------------------------------------------------*/

   n = wxl_coupnum(settlement,maturity,frequency,basis);
   matday = day(maturity);
   matmonth = month(maturity);
   matyear = year(maturity);

   select(frequency);
      when(1)
         do;
         cpstartyear = matyear - n;
         cpstartmon = matmonth;
      end;
      when(2)
         do;
         if int(n/2) = n/2 then
            do;
            cpstartyear = matyear - n/2;
            cpstartmon = matmonth;
```

Examples

Example 1 - Creating a Subroutine and a Function

This example demonstrates how to use PROC FCMP to create and store subroutines and functions.

First, a generic subroutine called **calc_years** is declared to calculate the number of years to maturity, given that date variables are stored in days.

A second, more complicated function **garkhprc** is also declared, which calculates Garman-Kohlhagen pricing for FX options. Notice that it makes use of the SAS functions **garkhclprc** and **garkhptprc**.

```
proc fcmp outlib= sasuser.exsubs.pkt1;

   /*-- subroutine to calculate ----------*/
   /*-- years to maturity ----------------*/
   subroutine calc_years( maturity, current_date, years );
      outargs years;
      years = (maturity - current_date) / 365.25;
   endsub;

   /*-- function for Garman-Kohlhagen ----*/
   /*-- pricing for FX options -----------*/
   function garkhprc( type$, buysell$, amount,
                      E, t, S, rd, rf, sig );

      if buysell = "Buy"  then  sign = 1.;
      else do;
         if  buysell = "Sell"  then  sign = -1.;
         else  sign = . ;
      end;

      if type = "Call" then
         garkhprc = sign * amount
                    * garkhclprc( E, t, S, rd, rf, sig );
      else do;
         if type = "Put" then
            garkhprc = sign * amount
                       * garkhptprc( E, t, S, rd, rf, sig );
         else garkhprc = . ;
      end;

      return(garkhprc);
   endsub;
run;
```