Chapter 5
Programming Statements

Chapter Table of Contents

Chapter 5
Programming Statements

Overview

As a programming language, the Interactive Matrix Language has many features that enable you to control the path of execution through the statements. The control statements in IML function in a way that is similar to the corresponding statements in the SAS DATA step. This chapter presents the following control features:

- IF-THEN/ELSE statements
- DO groups
- iterative execution
- jumping (nonconsecutive execution)
- module definition and execution
- termination of execution.

IF-THEN/ELSE Statements

To perform an operation conditionally, use an IF statement to test an expression. Alternative actions appear in a THEN clause and, optionally, an ELSE statement. The general form of the IF-THEN/ELSE statement is

IF *expression* **THEN** *statement1* **;**
ELSE *statement2* **;**

The IF expression is evaluated first. If the expression is true, execution flows through the THEN alternative. If the expression is false, the ELSE statement, if present, is executed. Otherwise, the next statement is executed.

The expression to be evaluated is often a comparison, for example,

```
if max(a)<20 then p=0;
else p=1;
```

The IF statement results in the evaluation of the condition (MAX(A)<20). If the largest value found in matrix **A** is less than 20, P is set to 0. Otherwise, P is set to 1.

You can nest IF statements within the clauses of other IF or ELSE statements. Any number of nesting levels is allowed. The following is an example of nested IF statements:

```
if x=y then
   if abs(y)=z then w=-1;
   else w=0;
else w=1;
```

When the condition to be evaluated is a matrix expression, the result of the evaluation is a temporary matrix of 0s, 1s, and possibly missing values. If all values of the result matrix are nonzero and nonmissing, the condition is true; if any element in the result matrix is 0, the condition is false. This evaluation is equivalent to using the ALL function.

For example, the statement

if x<y then *statement*;

produces the same result as the statement

if all(x<y) then *statement*;

The expressions

if a^=b then *statement*;

and

if ^(a=b) then *statement*;

are valid, but the THEN clause in each case is executed only when all corresponding elements of **A** and **B** are unequal.

If you require that only one element in **A** not be equal to its corresponding element in **B**, use the ANY function. For example, evaluation of the expression

if any(a^=b) then *statement*;

requires only one element of **A** and **B** to be unequal for the expression to be true.

DO Groups

A set of statements can be treated as a unit by putting them into a DO group, which starts with a DO statement and ends with an END statement. In this way, you can submit the entire group of statements for execution as a single unit. For some programming applications, you must use either a DO group or a module. For example, LINK and GOTO statements must be programmed inside of a DO group or a module.

The two principal uses of DO groups are

- to group a set of statements so that they are executed as a unit
- to group a set of statements for a conditional (IF-THEN/ELSE) clause

DO groups have the following general form:

```
DO;
  additional statements

END;
```

You can nest DO groups to any level, just like you nest IF-THEN/ELSE statements. The following is an example of nested DO groups:

```
do;
    statements;
    do;
        statements;
        do;
            statements;
        end;
    end;
end;
```

It is good practice to indent the statements in DO groups, as shown abovein the preceding statements so that their position indicates the levels of nesting.

For IF-THEN/ELSE conditionals, DO groups can be used as units for either THEN or ELSE clauses so that you can perform many statements as part of the conditional action. An example follows:

```
if x<y then
   do;
      z1=abs(x+y);
      z2=abs(x-y);
   end;
else
   do;
      z1=abs(x-y);
      z2=abs(x+y);
   end;
```

Iterative Execution

The DO statement also serves the feature of iteration. With a DO statement, you can repeatedly execute a set of statements until some condition stops the execution. A DO statement is iterative if you specify it with any of the following iteration clauses. The type of clause determines when to stop the repetition.

Clause	DO Statement
DATA	DO DATA statement
variable = *start* TO *stop* < BY *increment* >	iterative DO statement
WHILE(*expression*)	DO WHILE statement
UNTIL(*expression*)	DO UNTIL statement

A DO statement can have any combination of these four iteration clauses, but a given DO statement must be specified in the order listed in the preceding table.

DO DATA Statement

The general form of the DO DATA statement is

DO DATA;

The DATA keyword specifies that iteration is to stop when an end-of-file condition occurs. The group is exited immediately upon encountering the end-of-file condition. Other DO specifications exit after tests are performed at the top or bottom of the loop. See Chapter 6, "Working with SAS Data Sets," and Chapter 7, "File Access," for more information about processing data.

You can use the DO DATA statement to read data from an external file or to process observations from a SAS data set. In the DATA step in base SAS software, the iteration is usually implied. The DO DATA statement simulates this iteration until the end of file is reached.

The following example reads data from an external file named MYDATA and inputs the data values into a vector. The data values are read one at a time into the dummy variable XX and collected into the vector **X** using the vertical concatenation operator (//) after each value is read.

```
infile 'mydata';                    /* infile statement       */
do data;                            /* begin read loop        */
   input xx;                        /* read a data value      */
   x=x//xx;                         /* concatenate values     */
end;                                /* end loop               */
```

Iterative DO Statement

The general form of the iterative DO statement is

DO *variable=start* **TO** *stop* < **BY** *increment* > **;**

The *variable* sequence specification assigns the *start* value to the given variable. This value is then incremented by the *increment* value (or by 1 if *increment* is not specified) until it is greater than or equal to the *stop* value. (If *increment* is negative, then the iterations stop when the value is less than or equal to *stop*.)

For example, the following statement specifies a DO loop that executes by multiples of 10 until I is greater than 100:

```
do i=10 to 100 by 10;
```

DO WHILE Statement

The general form of the DO WHILE statement is

DO WHILE *expression*;

With a WHILE clause, the expression is evaluated at the beginning of each loop, with repetition continuing until the expression is false (that is, until the value contains a 0 or missing value). Note that if the expression is false the first time it is evaluated, the loop is not executed.

For example, if the variable COUNT has an initial value of 1, the statements

```
do while(count<5);
   print count;
   count=count+1;
end;
```

print COUNT four times.

DO UNTIL Statement

The general form of the DO UNTIL statement is

DO UNTIL *expression*;

The UNTIL clause is like the WHILE clause except that the expression is evaluated at the bottom of the loop. This means that the loop always executes at least once.

For example, if the variable COUNT has an initial value of 1, the statements

```
do until(count>5);
   print count;
   count=count+1;
end;
```

print COUNT five times.

Jumping

During normal execution, statements are executed one after another. The GOTO and LINK statements instruct IML to jump from one part of a program to another. The place to which execution jumps is identified by a *label*, which is a name followed by a colon placed before an executable statement. You can program a jump by using either the GOTO statement or the LINK statement:

GOTO *label*;
LINK *label*;

Both the GOTO and the LINK statements instruct IML to jump immediately to the labeled statement. The LINK statement, however, reminds IML where it jumped from so that execution can be returned there if a RETURN statement is encountered. The GOTO statement does not have this feature. Thus, the LINK statement provides a way of calling sections of code as if they were subroutines. The LINK statement calls the routine. The routine begins with the label and ends with a RETURN statement. LINK statements can be nested within other LINK statements to any level.

CAUTION: The GOTO and LINK statements are limited to being inside a module or DO group. These statements must be able to resolve the referenced label within the current unit of statements. Although matrix symbols can be shared across modules, statement labels cannot. Therefore, all GOTO statement labels and LINK statement labels must be local to the DO group or module.

The GOTO and LINK statements are not often used because you can usually write more understandable programs by using other features, such as DO groups for conditionals, iterative DO groups for looping, and module invocations for subroutine calls.

Here are two DO groups that illustrate how the GOTO and LINK statements work:

```
do;
   if x<0 then goto negative;
   y=sqrt(x);
   print y;
   stop;
negative:
   print "Sorry, X is negative";
end;
```

```
do;
   if x<0 then link negative;
   y=sqrt(x);
   print y;
   stop;
negative:
   print "Using Abs. value of negative X";
   x=abs(x);
   return;
end;
```

The following is a comparable way to write the program on the left without using GOTO or LINK statements:

```
if x<0 then print "Sorry, X is negative";
else
   do;
      y=sqrt(x);
      print y;
   end;
```

Modules

Modules are used for

- creating groups of statements that can be invoked as a unit from anywhere in the program, that is, making a subroutine or function
- creating a separate (symbol-table) environment, that is, defining variables that are local to the module rather than global

A module always begins with the START statement and ends with the FINISH statement. Modules can be thought of as being either functions or subroutines. When a module returns a single parameter, it is called a function and is executed as if it were a built-in IML function; a function is invoked by its name in an assignment statement rather than in a CALL or RUN statement. Otherwise, a module is called a subroutine, and you execute the module in either the RUN statement or the CALL statement.

Defining and Executing a Module

Modules begin with a START statement, which has the general form

START < *name* > < **(** *arguments* **)** > < **GLOBAL(** *arguments* **)** > **;**

Modules end with a FINISH statement, which has the general form

FINISH < *name* > **;**

If no name appears in the START statement, the name of the module defaults to MAIN.

There are two ways you can execute a module. You can use either a RUN statement or a CALL statement. The only difference is the order of resolution.

The general forms of these statements are

RUN *name* < **(** *arguments* **)** > **;**
CALL *name* < **(** *arguments* **)** > **;**

The RUN and CALL statements must have arguments to correspond to the ones defined for the modules they invoke. A module can call other modules provided that it never recursively calls itself.

The RUN and CALL statements have orders of resolution that need to be considered only when you have given a module the same name as a built-in IML subroutine. In such cases, use the CALL statement to execute the built-in subroutine and the RUN statement to execute the user-defined module.

The RUN statement is resolved in the following order:

1. user-defined module
2. IML built-in function or subroutine

The CALL statement is resolved in the following order:

1. IML built-in subroutine
2. user-defined module

Nesting Modules

You can nest one module within another. You must make sure that each nested module is completely contained inside of the parent module. Each module is collected independently of the others. When you nest modules, it is a good idea to indent the statements relative to the level of nesting, as shown in the following example:

```
start a;
   reset print;
   start b;
      a=a+1;
   finish b;
   run b;
finish a;
run a;
```

In this example, IML starts collecting statements for a module called A. In the middle of this module, it recognizes the start of a new module called B. It saves its current work on A and collects B until encountering the first FINISH statement. It then finishes collecting A. Thus, it behaves the same as if B were collected before A, as shown below:

```
start b;
   a=a+1;
finish;
start a;
   reset print;
   run b;
finish;
run a;
```

Understanding Symbol Tables

Whenever a variable is defined outside of the module environment, its name is stored in the *global symbol table*. Whenever you are programming in immediate mode outside of a module, you are working with symbols (variables) from the global symbol table. For each module you define with arguments given in a START statement, a separate symbol table called a *local symbol table* is created for that module. All symbols (variables) used inside the module are stored in its local symbol table. There can be many local symbol tables, one for each module with arguments. A symbol can exist in the global table, one or more local tables, or in both the global table and one or more local tables. Also, depending on how a module is defined, there can be a one-to-one correspondence between variables across symbol tables (although there need not be any connection between a variable, say X, in the global table and a variable X in a local table). Values of symbols in a local table are temporary; that is, they exist only while the module is executing and are lost when the module has finished execution. Whether or not these temporary values are transferred to corresponding global variables depends on how the module is defined.

Modules with No Arguments

When you define a module with no arguments, a local symbol table is not created. All symbols (variables) are global, that is, equally accessible inside and outside the module. The symbols referenced inside the module are the same as the symbols outside the module environment. This means that variables created inside the module are also global, and any operations done on variables inside a module affect the global variables as well.

The following example shows a module with no arguments:

```
>     /* module without arguments, all symbols are global.  */
>  proc iml;
>  a=10;                                  /* A is global      */
>  b=20;                                  /* B is global      */
>  c=30;                                  /* C is global      */
>  start mod1;                            /* begin module     */
>     p=a+b;                              /* P is global      */
>     q=b-a;                              /* Q is global      */
>     c=40;                               /* C already global */
>  finish;                                /* end module       */

   NOTE: Module MOD1 defined.

>  run mod1;
>  print a b c p q;

                    A         B         C         P         Q
                   10        20        40        30        10
```

Note that after executing the module,

- A is still 10
- B is still 20
- C has been changed to 40
- P and Q are created, added to the global symbol table, and set to 30 and 10, respectively

Modules with Arguments

In general, the following statements are true about modules with arguments:

- You can specify arguments as variable names.
- If you specify several arguments, use commas to separate them.
- If you have both output variables and input variables, it is good practice to list the output variables first.

- When a module is invoked with either a RUN or a CALL statement, the arguments can be any name, expression, or literal. However, when using arguments for output results, use variable names rather than expressions or literals.

When a module is executed with either a RUN or a CALL statement, the value for each argument is transferred from the global symbol table to the local symbol table. For example, consider the module MOD2 defined in the following statements. The first four statements are submitted in the global environment, and they define variables (A,B,C, and D): the values of these variables are stored in the global symbol table. The START statement begins definition of MOD2 and lists two variables (X and Y) as arguments. This creates a local symbol table for MOD2. All symbols used inside the module (X, Y, P, Q, and C) are in the local symbol table. There is also a one-to-one correspondence between the arguments in the RUN statement (A and B) and the arguments in the START statement (X and Y). Also note that A, B, and D exist only in the global symbol table, whereas X, Y, P, and Q exist only in the local symbol table. The symbol C exists independently in both the local and global tables. When MOD2 is executed with the statement RUN MOD2(A,B), the value of A is transferred from the global symbol table to X in the local table. Similarly, the value of B in the global table is transferred to Y in the local table. Because C is not an argument, there is no correspondence between the value of C in the global table and the value of C in the local table. When the module finishes execution, the final values of X and Y in the local table are transferred back to A and B in the global table.

```
>  proc iml;
>  a=10;
>  b=20;
>  c=30;
>  d=90;
>  start mod2(x,y);                    /* begin module */
>     p=x+y;
>     q=y-x;
>     y=100;
>     c=25;
>  finish mod2;                        /* end module   */

   NOTE: Module MOD2 defined.

>  run mod2(a,b);
>  print a b c d;

                          A         B         C         D
                         10       100        30        90
```

The PRINT statement prints the values of variables in the global symbol table. Notice that

- A is still 10
- B is changed to 100 since the corresponding argument Y was changed to 100 inside the module

- C is still 30. Inside the module, the local symbol C was set equal to 25, but there is no correspondence between the global symbol C and the local symbol C.
- D is still 90

Also note that, inside the module, the symbols A, B, and D do not exist. Outside the module, the symbols P, Q, X, and Y do not exist.

Defining Function Modules

Functions are special modules that return a single value. They are a special type of module because modules can, in general, return any number of values through their argument list. To write a function module, include a RETURN statement that assigns the returned value to a variable. The RETURN statement is necessary for a module to be a function. You invoke a function module in an assignment statement, as you would a standard function.

The symbol-table logic described in the preceding section also applies to function modules. The following is an example of a function module. In this module, the value of C in the local symbol table is transferred to the global symbol Z.

```
>  proc iml;
>  a=10;
>  b=20;
>  c=30;
>  d=90;
>  start mod3(x,y);
>     p=x+y;
>     q=y-x;
>     y=100;
>     c=40;
>     return (c);            /* return function value */
>  finish mod3;

   NOTE: Module MOD3 defined.

>  z = mod3(a,b);            /* call function         */
>  print a b c d z;


                         A         B         C         D         Z
                        10       100        30        90        40
```

Note the following about this example:

- A is still 10.
- B is changed to 100 because Y is set to 100 inside the module, and there is a one-to-one correspondence between B and Y.
- C is still 30. The symbol C in the global table has no connection with the symbol C in the local table.

- Z is set to 40, which is the value of C in the local table.

Again note that, inside the module, the symbols A, B, D, and Z do not exist. Outside the module, symbols P, Q, X, and Y do not exist.

In the next example, you define your own function ADD for adding two arguments:

```
>  proc iml;
>  reset print;
>  start add(x,y);
>     sum=x+y;
>     return(sum);
>  finish;

NOTE: Module ADD defined.

>   a={9 2,5 7};

                                A
                                9         2
                                5         7


>   b={1 6,8 10};

                                B
                                1         6
                                8        10


>   c=add(a,b);

                                C
                               10         8
                               13        17
```

Function modules can also be called inside each other. For example, in the following statements, the ADD function is called twice from within the first ADD function:

```
>  d=add(add(6,3),add(5,5));
>  print d;
                                      D
                                     19
```

Functions are resolved in this order:

1. IML built-in function
2. user-defined function module
3. SAS DATA step function

This means that you should not use a name for a function that is already the name of an IML built-in function.

Using the GLOBAL Clause

For modules with arguments, the variables used inside the module are local and have no connection with any variables of the same name existing outside the module in the global table. However, it is possible to specify that certain variables not be placed in the local symbol table but rather be accessed from the global table. Use the GLOBAL clause to specify variables you want shared between local and global symbol tables. The following is an example of a module using a GLOBAL clause to define the symbol C as global. This defines a one-to-one correspondence between the value of C in the global table and the value of C in the local table.

```
>  proc iml;
>  a=10;
>  b=20;
>  c=30;
>  d=90;
>  start mod4(x,y) global (c);
>     p=x+y;
>     q=y-x;
>     y=100;
>     c=40;
>     d=500;
>  finish mod4;

   NOTE: Module MOD4 defined.

>  run mod4(a,b);
>  print a b c d;

                         A         B         C         D
                        10       100        40        90
```

Note the following about this example:

- A is still 10.
- B is changed to 100.
- C is changed to 40 because it was declared global. The C inside the module and outside the module are the “same.”
- D is still 90 and not 500, since D independently exists in the global and local symbol tables.

Also note that every module with arguments has its own local table; thus it is possible to have a global and many local tables. A variable can independently exist in one or more of these tables. However, a variable can be commonly shared between the global and any number of local tables when the GLOBAL clause is used.

Nesting Modules with Arguments

For nested module calls, the concept of global and local symbol tables is somewhat different. Consider the following example:

```
>  start mod1 (a,b);
>     c=a+b;
>     d=a-b;
>     run mod2 (c,d);
>     print c d;
>  finish mod1;

   NOTE: Module MOD1 defined.

>  start mod2 (x,y);
>     x=y-x;
>     y=x+1;
>     run mod3(x,y);
>  finish mod2;

   NOTE: Module MOD2 defined.

>  start mod3(w,v);
>     w=w#v;
>  finish mod3;

   NOTE: Module MOD3 defined.
```

The local symbol table of MOD1 in effect becomes the global table for MOD2. The local symbol table of MOD2 is the global table for MOD3. The distinction between the global and local environments is necessary only for modules with arguments. If a module (say, A) calls another module (say, B) which has no arguments, B shares all the symbols existing in A's local symbol table.

For example, consider the following statements:

```
>  x=457;
>  start a;
>     print 'from a' x;
>     finish;
>  start b(p);
>     print 'from b' p;
>     run a;
>     finish;
>  run b(x);

                                        P
                          from b      457

ERROR: Matrix X has not been set to a value.
Error occured in module A
called   from    module B
stmt: PRINT

Paused in module A.
```

In this example, module A is called from module B. Therefore, the local symbol table of module B becomes the global symbol table for module A. Module A has access to all symbols available in module B. No X exists in the local environment of module B; thus no X is available in module A as well. This causes the error that X is unvalued.

More about Argument Passing

You can pass expressions and subscripted matrices as arguments to a module, but you must be careful and understand the way IML evaluates the expressions and passes results to the module. Expressions are evaluated, and the evaluated values are stored in temporary variables. Similarly, submatrices are created from subscripted variables and stored in temporary variables. The temporary variables are passed to the module while the original matrix remains intact. Notice that, in the example that follows, the matrix X remains intact. You might expect X to contain the squared values of Y.

```
>  proc iml;
>  reset printall;
>  start square(a,b);
>     a=b##2;
>  finish;
>     /*  create two data matrices                          */
>  x={5 9 };

          X             1 row       2 cols    (numeric)

                           5         9

>  y={10 4};

          Y             1 row       2 cols    (numeric)

                          10         4
>     /*  pass matrices to module element-by-element      */
>  do i=1 to 2;
>     run square(x[i],y[i]);
>  end;
>     /*  RESET PRINTALL prints all intermediate results  */

           I             1 row       1 col     (numeric)

                                     1


           #TEM1002      1 row       1 col     (numeric)

                                    10


           #TEM1001      1 row       1 col     (numeric)

                                     5
```

```
A             1 row       1 col     (numeric)

                         100

#TEM1002      1 row       1 col     (numeric)

                           4

#TEM1001      1 row       1 col     (numeric)

                           9

A             1 row       1 col     (numeric)

                          16

>     /*  show X and Y are unchanged                    */
>  print x y;

                      X                   Y
                      5         9         10         4
```

The symbol X remains unchanged because the temporary variables that you generally do not see are changed. Note that IML will properly warn you of any such instances in which your results may be lost to the temporary variables.

Module Storage

You can store and reload modules using the forms of the STORE and LOAD statements as they pertain to modules:

STORE MODULE=*name*;
LOAD MODULE=*name*;

You can view the names of the modules in storage with the SHOW statement:

```
show storage;
```

See Chapter 14, “Storage Features,” for details on using the library storage facilities.

Stopping Execution

You can stop execution with a PAUSE, STOP, or ABORT statement. The QUIT statement is also a stopping statement, but it immediately removes you from the IML environment; the other stopping statements can be performed in the context of a program. Following are descriptions of the STOP, ABORT, and PAUSE statements.

PAUSE Statement

The general form of the PAUSE statement is

PAUSE < *message* > < ***** > **;**

The PAUSE statement

- stops execution of the module
- remembers where it stopped executing
- prints a pause *message* that you can specify
- puts you in immediate mode within the module environment using the module's local symbol table. At this point you can enter more statements.

A RESUME statement enables you to continue execution at the place where the most recent PAUSE statement was executed.

You can use a STOP statement as an alternative to the RESUME statement to remove the paused states and return to the immediate environment outside of the module.

You can specify a message in the PAUSE statement to display a message as the pause prompt. If no message is specified, IML displays the following default message:

```
paused in module \ob XXX\obe
```

where XXX is the name of the module containing the pause. To suppress the display of any messages, use the * option:

```
pause *;
```

The followingw are some examples of PAUSE statements with operands:

```
pause "Please enter an assignment for X, then enter RESUME;";

msg ="Please enter an assignment for X, then enter RESUME;";
pause msg;
```

When you use the PAUSE, RESUME, STOP, or ABORT statement, you should be aware of the following details:

- The PAUSE statement can be issued only from within a module.
- IML diagnoses an error if you execute a RESUME statement without any pauses outstanding.
- You can define and execute modules while paused from other modules.

- A PAUSE statement is automatically issued if an error occurs while executing statements inside a module. This gives you an opportunity to correct the error and resume execution of the module with a RESUME statement. Alternately, you can submit a STOP or ABORT statement to exit from the module environment.
- You cannot reenter or redefine an active (paused) module; you will get an error for recursive module execution.
- In paused mode, you can run another module that, in turn, pauses; the paused environments are stacked.
- You can put a RESUME statement inside a module. For example, suppose you are paused in module A and then run module B, which executes a RESUME statement. Execution is resumed in module A and does not return to module B.
- IML supports stopping execution while in a paused state in both subroutine and function modules.
- If you pause in a subroutine module that has its own symbol table, then the immediate mode during the pause uses this symbol table rather than the global one. You must use a RESUME or a STOP statement to return to the global symbol table environment.
- You can use the PAUSE and RESUME statements, in conjunction with the PUSH, QUEUE, and EXECUTE subroutines described in Chapter 15, "Using SAS/IML Software to Generate IML Statements," to execute IML statements that you generate within a module.

STOP Statement

The general form of the STOP statement is

STOP;

The STOP statement stops execution and returns you to immediate mode, where new statements that you enter are executed. If execution is interrupted by a PAUSE statement, the STOP statement clears all pauses and returns to immediate mode of execution.

ABORT Statement

The general form of the ABORT statement is

ABORT;

The ABORT statement stops execution and exits from IML much like a QUIT statement, except that the ABORT statement is executable and programmable. For example, you may want to exit IML if a certain error occurs. You can check for the error

in a module and program an ABORT statement to execute if the error occurs. The ABORT statement does not execute until the module is executed, while the QUIT statement executes immediately and ends the IML session.

Summary

In this chapter you learned the basics of programming with SAS/IML software. You learned about conditional execution (IF-THEN/ELSE statements), grouping statements as a unit (DO groups), iterative execution, nonconsecutive execution, defining subroutines and functions (modules), and stopping execution. With these programming capabilities, you are able to write your own sophisticated programs and store the code as a module. You can then execute the program later with a RUN or CALL statement.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/IML User's Guide, Version 8*, Cary, NC: SAS Institute Inc., 1999. 846 pp.

SAS/IML User's Guide, Version 8

ISBN 1–58025–553–1

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

1st printing, October 1999