SAS/OR® 15.1 User's Guide
Mathematical Programming
The OPTMODEL Procedure

This document is an individual chapter from *SAS/OR® 15.1 User's Guide: Mathematical Programming*.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2018. *SAS/OR® 15.1 User's Guide: Mathematical Programming*. Cary, NC: SAS Institute Inc.

SAS/OR® 15.1 User's Guide: Mathematical Programming

Copyright © 2018, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

For a hard-copy book: 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.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

November 2018

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.

SAS software may be provided with certain third-party software, including but not limited to open-source software, which is licensed under its applicable third-party software license agreement. For license information about third-party software distributed with SAS software, refer to http://support.sas.com/thirdpartylicenses.

Chapter 5
The OPTMODEL Procedure

Contents

Overview: OPTMODEL Procedure

The OPTMODEL procedure includes the powerful OPTMODEL modeling language and state-of-the-art solvers for several classes of mathematical programming problems. The problems and their solvers are listed in Table 5.1.

Table 5.1 Solvers in PROC OPTMODEL

Problem	Solver
Constraint programming	**CLP**
Linear programming	**LP**
Local search optimization	**LSO**
Mixed integer linear programming	**MILP**
Network algorithms	**Network**
General nonlinear programming	**NLP**
Quadratic programming	**QP**

The OPTMODEL modeling language provides a modeling environment tailored to building, solving, and maintaining optimization models. This makes the process of translating the symbolic formulation of an optimization model into OPTMODEL virtually transparent since the modeling language mimics the symbolic algebra of the formulation as closely as possible. The OPTMODEL language also streamlines and simplifies

the critical process of populating optimization models with data from SAS data sets. All of this transparency produces models that are more easily inspected for completeness and correctness, more easily corrected, and more easily modified, whether through structural changes or through the substitution of new data for old.

In addition to invoking optimization solvers directly with PROC OPTMODEL as already mentioned, you can use the OPTMODEL language purely as a modeling facility. You can save optimization models built with the OPTMODEL language in SAS data sets that can be submitted to other SAS/OR optimization procedures. In general, the OPTMODEL language serves as a common point of access for many of the SAS/OR optimization capabilities, whether providing both modeling and solver access or acting as a modeling interface for other optimization procedures.

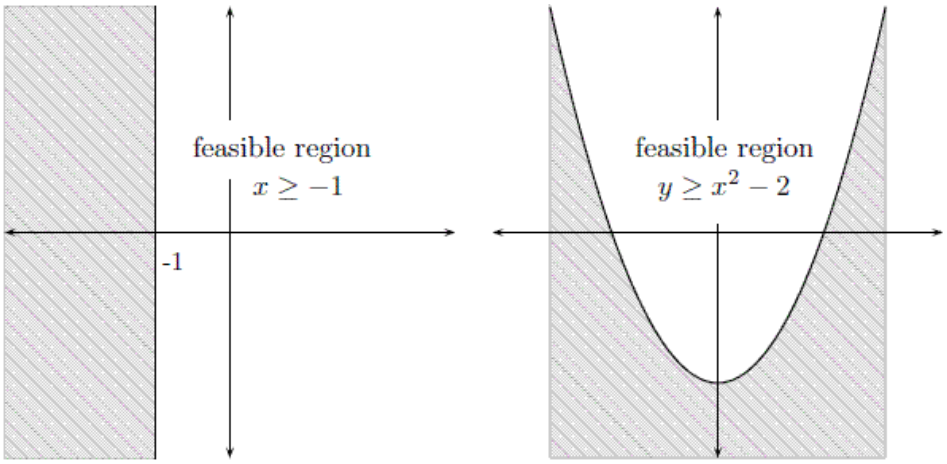
For details and examples of the problems addressed and corresponding solvers, please see the dedicated chapters in this book. This chapter aims to give you a comprehensive understanding of the OPTMODEL procedure by discussing the framework provided by the OPTMODEL modeling language. For additional examples that demonstrate the features of the OPTMODEL procedure, see *SAS/OR User's Guide: Mathematical Programming Examples*.

The OPTMODEL modeling language features automatic differentiation, advanced flow control, optimization-oriented syntax (parameters, variables, arrays, constraints, objective functions), dynamic model generation, model-data separation, and transparent access to SAS data sets.

Getting Started: OPTMODEL Procedure

Optimization or mathematical programming is a search for a maximum or minimum of an *objective function* (also called a *cost function*), where search variables are restricted to particular constraints. Constraints are said to define a *feasible region* (see Figure 5.1).

Figure 5.1 Examples of Feasible Regions



A more rigorous general formulation of such problems is as follows.

Let

$$f : S \rightarrow \mathbb{R}$$

be a real-valued function. Find x^* such that

- $x^* \in S$
- $f(x^*) \leq f(x), \quad \forall x \in S$

Note that the formulation is for the minimum of f and that the maximum of f is simply the negation of the minimum of $-f$.

Here, function f is the *objective function*, and the variable in the objective function is called the optimization variable (or decision variable). S is the *feasible region*. Typically S is a subset of the Euclidean space $\mathbb{R}^n$ specified by the set of *constraints*, which are often a set of equalities ($=$) or inequalities ($\leq, \geq$) that every element in S is required to satisfy simultaneously. For the special case where $S = \mathbb{R}^n$, the problem is an *unconstrained optimization*. An element x of S is called a *feasible solution* to the optimization problem, and the value $f(x)$ is called the *objective value*. A feasible solution x^* that minimizes the objective function is called an *optimal solution* to the optimization problem, and the corresponding objective value is called the *optimal value*.

In mathematics, special notation is used to denote an optimization problem. Generally, you can write an optimization problem as follows:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & x \in S \end{array}$$

Normally, an empty body of constraint (the part after "subject to") implies that the optimization is unconstrained (that is, the feasible region is the whole space $\mathbb{R}^n$). The optimal solution (x^*) is denoted as

$$x^* = \underset{x \in S}{\operatorname{argmin}}\, f(x)$$

The optimal value ($f(x^*)$) is denoted as

$$f(x^*) = \min_{x \in S} f(x)$$

Optimization problems can be classified by the forms (linear, quadratic, nonlinear, and so on) of the functions in the objective and constraints. For example, a problem is said to be *linearly constrained* if the functions in the constraints are linear. A *linear programming* problem is a linearly constrained problem with a linear objective function. A nonlinear programming problem occurs where some function in the objective or constraints is nonlinear, and so on.

An Unconstrained Optimization Example

An unconstrained optimization problem formulation is simply

$$\text{minimize} \quad f(x)$$

For example, suppose you wanted to find the minimum value of this polynomial:

$$z(x, y) = x^2 - x - 2y - xy + y^2$$

You can compactly specify and solve the optimization problem by using the OPTMODEL modeling language. Here is the program:

```
/* invoke procedure */
proc optmodel;
   var x, y;  /* declare variables */

   /* objective function */
   min z=x**2 - x - 2*y - x*y + y**2;

   /* now run the solver */
   solve;

   print x y;
quit;
```

This program produces the output in Figure 5.2.

Figure 5.2 Optimizing a Simple Polynomial

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	z
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	0
Constraint Coefficients	0
Hessian Diagonal Elements	2
Hessian Elements Below Diagonal	1

Figure 5.2 *continued*

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	z
Solution Status	Optimal
Objective Value	-2.333333333
Primal Infeasibility	0
Dual Infeasibility	6.861556E-17
Bound Infeasibility	0
Duality Gap	0
Complementarity	0
Iterations	0
Presolve Time	0.00
Solution Time	0.07

x	y
1.3333	1.6667

In PROC OPTMODEL you specify the mathematical formulas that describe the behavior of the optimization problem that you want to solve. In the preceding example there were two independent variables in the polynomial, *x* and *y*. These are the *optimization variables* of the problem. In PROC OPTMODEL you declare optimization variables with the VAR statement. The formula that defines the quantity that you are seeking to optimize is called the *objective function*, or *objective*. The solver varies the values of the optimization variables when searching for an optimal value for the objective.

In the preceding example the objective function is named *z*, declared with the MIN statement. The keyword MIN is an abbreviation for MINIMIZE. The expression that follows the equal sign (=) in the MIN statement defines the function to be minimized in terms of the optimization variables.

The VAR and MIN statements are just two of the many available PROC OPTMODEL declaration and programming statements. PROC OPTMODEL processes all such statements interactively, meaning that each statement is processed as soon as it is complete.

After PROC OPTMODEL has completed processing of declaration and programming statements, it processes the SOLVE statement, which submits the problem to a solver and prints a summary of the results. The PRINT statement displays the optimal values of the optimization variables x and y found by the solver.

It is worth noting that PROC OPTMODEL does not use a RUN statement but instead operates on an interactive basis throughout. You can continue to interact with PROC OPTMODEL even after invoking a solver. For example, you could modify the problem and issue another SOLVE statement (see the section "Model Update" on page 147).

The Rosenbrock Problem

You can use parameters to produce a clear formulation of a problem. Consider the Rosenbrock problem,

$$\text{minimize } f(x_1, x_2) = \alpha\,(x_2 - x_1^2)^2 + (1 - x_1)^2$$

where $\alpha = 100$ is a parameter (constant), x_1 and x_2 are optimization variables (whose values are to be determined), and $f(x_1, x_2)$ is an objective function.

Here is a PROC OPTMODEL program that solves the Rosenbrock problem:

```
proc optmodel;
   number alpha = 100; /* declare parameter */
   var x {1..2};       /* declare variables */
   /* objective function */
   min f = alpha*(x[2] - x[1]**2)**2 +
           (1 - x[1])**2;
   /* now run the solver */
   solve;

   print x;
quit;
```

The PROC OPTMODEL output is shown in Figure 5.3.

Figure 5.3 Rosenbrock Function Results

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Nonlinear
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	0

Figure 5.3 *continued*

Solution Summary	
Solver	NLP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	8.204873E-23
Optimality Error	9.704881E-11
Infeasibility	0
Iterations	14
Presolve Time	0.00
Solution Time	0.01

[1]	x
1	1
2	1

A Transportation Problem

You can easily translate the symbolic formulation of a problem into the OPTMODEL procedure. Consider the transportation problem, which is mathematically modeled as the following linear programming problem:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i \in O, j \in D} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{j \in D} x_{ij} = a_i, \quad \forall i \in O \qquad \text{(SUPPLY)} \\
& \sum_{i \in O} x_{ij} = b_j, \quad \forall j \in D \qquad \text{(DEMAND)} \\
& x_{ij} \geq 0, \quad \forall (i,j) \in O \times D
\end{aligned}$$

where O is the set of origins, D is the set of destinations, c_{ij} is the cost to transport one unit from i to j, a_i is the supply of origin i, b_j is the demand of destination j, and x_{ij} is the decision variable for the amount of shipment from i to j.

Here is a very simple example. The cities in the set O of origins are Detroit and Pittsburgh. The cities in the set D of destinations are Boston and New York. The cost matrix, supply, and demand are shown in Table 5.2.

Table 5.2 A Transportation Problem

	Boston	New York	Supply
Detroit	30	20	200
Pittsburgh	40	10	100
Demand	150	150	

The problem is compactly and clearly formulated and solved by using the OPTMODEL procedure with the following statements:

```
proc optmodel;
   /* specify parameters */
   set O={'Detroit','Pittsburgh'};
   set D={'Boston','New York'};
   number c{O,D}=[30 20
                  40 10];
   number a{O}=[200 100];
   number b{D}=[150 150];
   /* model description */
   var x{O,D} >= 0;
   min total_cost = sum{i in O, j in D}c[i,j]*x[i,j];
   constraint supply{i in O}: sum{j in D}x[i,j]=a[i];
   constraint demand{j in D}: sum{i in O}x[i,j]=b[j];
   /* solve and output */
   solve;
   print x;
```

The output is shown in Figure 5.4.

Figure 5.4 Solution to the Transportation Problem

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	total_cost
Objective Type	Linear
Number of Variables	4
Bounded Above	0
Bounded Below	4
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	4
Linear LE (<=)	0
Linear EQ (=)	4
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	8

Figure 5.4 *continued*

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	total_cost
Solution Status	Optimal
Objective Value	6500
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	0
Presolve Time	0.00
Solution Time	0.00

x		
	Boston	**New York**
Detroit	150	50
Pittsburgh	0	100

Syntax: OPTMODEL Procedure

PROC OPTMODEL statements are divided into three categories: the PROC statement, the declaration statements, and the programming statements. The PROC statement invokes the procedure and sets initial option values. The declaration statements declare optimization model components. The programming statements read and write data, invoke the solver, and print results. In the following text, the statements are listed in the order in which they are grouped, with declaration statements first.

NOTE: Solver specific options are described in the individual chapters that correspond to the solvers.

PROC OPTMODEL *options* **;**

Declaration Statements:
CONSTRAINT *constraints* **;**
IMPVAR *optimization expression declarations* **;**
MAX *objective* **;**
MIN *objective* **;**
NUMBER *parameter declarations* **;**
PROBLEM *problem declaration* **;**
SET [**<** *types* **>**] *parameter declarations* **;**
STRING *parameter declarations* **;**
VAR *variable declarations* **;**

Programming Statements:
Assignment parameter = *expression* **;**
CALL *name* [**(** *expressions* **)**] **;**
CLOSEFILE *files* **;**
COFOR { *index-set* **}** *statement* **;**
CONTINUE ;
CREATE DATA *SAS-data-set* **FROM** *columns* **;**
DO ; *statements* **; END ;**
DO *variable* **=** *specifications* **;** *statements* **; END ;**
DO UNTIL (*logic* **) ;** *statements* **; END ;**
DO WHILE (*logic* **) ;** *statements* **; END ;**
DROP *constraint* **;**
EXPAND *name* [*/ options*] **;**
FILE *file* **;**
FIX *variable* [**=** *expression*] **;**
FOR { *index-set* **}** *statement* **;**
IF *logic* **THEN** *statement* **;** [**ELSE** *statement*] **;**
LEAVE ;
Null ;
PRINT *print items* **;**
PROFILE [*mode*] *options* **;**
PUT *put items* **;**
QUIT ;
READ DATA *SAS-data-set* **INTO** *columns* **;**
RESET OPTIONS *options* **;**
RESTORE *constraint* **;**
SAVE MPS *SAS-data-set* [**OBJECTIVE** *name*] [**NOOBJECTIVE**] **;**
SAVE QPS *SAS-data-set* [**OBJECTIVE** *name*] [**NOOBJECTIVE**] **;**
SOLVE [**WITH** *solver*] [**OBJECTIVE** *name*] [**NOOBJECTIVE**] [**RELAXINT**] [*/ options*] **;**
STOP ;
SUBMIT *arguments* [*/ options*] **;**
UNFIX *variable* [**=** *expression*] **;**
USE PROBLEM *problem* **;**

Functional Summary

The statements and options available with PROC OPTMODEL are summarized by purpose in Table 5.3.

Table 5.3 Functional Summary

Description	Statement	Option
Declaration Statements:		
Declares a constraint	CONSTRAINT	
Declares optimization expressions	IMPVAR	
Declares a maximization objective	MAX	
Declares a minimization objective	MIN	
Declares a number type parameter	NUMBER	
Declares a problem	PROBLEM	
Declares a set type parameter	SET	
Declares a string type parameter	STRING	
Declares optimization variables	VAR	
Programming Statements:		
Assigns a value to a variable or parameter	=	
Invokes a library subroutine	CALL	
Closes the opened file	CLOSEFILE	
Executes the statement repeatedly with support for concurrent solver invocations	COFOR	
Terminates one iteration of a loop statement	CONTINUE	
Creates a new SAS data set and copies data into it from PROC OPTMODEL parameters and variables	CREATE DATA	
Groups a sequence of statements together as a single statement	DO	
Executes statements repeatedly	DO (iterative)	
Executes statements repeatedly until some condition is satisfied	DO UNTIL	
Executes statements repeatedly as long as some condition is satisfied	DO WHILE	
Ignores the specified constraint	DROP	
Prints the specified constraint, variable, or objective declaration expressions after expanding aggregation operators, and so on	EXPAND	
Selects a file for the PUT statement	FILE	
Treats a variable as fixed in value	FIX	
Executes the statement repeatedly	FOR	
Executes the statement conditionally	IF	
Terminates the execution of the entire loop body	LEAVE	
Null statement	;	
Outputs string and numeric data	PRINT	

Description	Statement	Option
Provides timing and execution count information for statements and declarations	PROFILE	
Writes text data to the current output file	PUT	
Terminates the PROC OPTMODEL session	QUIT	
Reads data from a SAS data set into PROC OPTMODEL parameters and variables	READ DATA	
Sets PROC OPTMODEL option values or restores them to their defaults	RESET OPTIONS	
Adds a constraint that was previously dropped back into the model	RESTORE	
Saves the structure and coefficients for a linear programming model into a SAS data set	SAVE MPS	
Saves the structure and coefficients for a quadratic programming model into a SAS data set	SAVE QPS	
Invokes a PROC OPTMODEL solver	SOLVE	
Halts the execution of all statements that contain it	STOP	
Submits SAS code for execution	SUBMIT	
Reverses the effect of FIX statement	UNFIX	
Selects the current problem	USE PROBLEM	
PROC OPTMODEL Options:		
Specifies the accuracy for nonlinear constraints	PROC OPTMODEL	CDIGITS=
Specifies the maximum number of error messages displayed	PROC OPTMODEL	ERRORLIMIT=
Specifies the method used to approximate numeric derivatives	PROC OPTMODEL	FD=
Specifies the accuracy for the objective function	PROC OPTMODEL	FDIGITS=
Forces finite differences to be used for nonlinear equations	PROC OPTMODEL	FORCEFD=
Enables the OPTMODEL presolver for the CLP, LP, MILP, and QP solvers	PROC OPTMODEL	FORCEPRESOLVE=
Passes initial values for variables to the solver	PROC OPTMODEL	INITVAR=
Specifies the tolerance for rounding the bounds on integer and binary variables	PROC OPTMODEL	INTFUZZ=
Specifies the maximum length for MPS row and column labels	PROC OPTMODEL	MAXLABLEN=
Checks missing values	PROC OPTMODEL	MISSCHECK=
Specifies the maximum number of non-error messages displayed	PROC OPTMODEL	MSGLIMIT=
Specifies the number for threads to use for threaded processing	PROC OPTMODEL	NTHREADS=
Specifies the number of digits to display	PROC OPTMODEL	PDIGITS=
Adjusts how two-dimensional array is displayed	PROC OPTMODEL	PMATRIX=
Specifies the type of presolve performed by the PROC OPTMODEL presolver	PROC OPTMODEL	PRESOLVER=

Description	Statement	Option
Specifies the tolerance, enabling the PROC OPTMODEL presolver to remove slightly infeasible constraints	PROC OPTMODEL	PRESTOL=
Enables or disables printing summary	PROC OPTMODEL	PRINTLEVEL=
Specifies the width to display numeric columns	PROC OPTMODEL	PWIDTH=
Specifies the smallest difference that is permitted by the PROC OPTMODEL presolver between the upper and lower bounds of an unfixed variable	PROC OPTMODEL	VARFUZZ=

PROC OPTMODEL Statement

PROC OPTMODEL [*options*] **;**

The PROC OPTMODEL statement invokes the OPTMODEL procedure. You can specify options to control how the optimization model is processed and how results are displayed. You can specify the following *options* (these options can also be specified in the RESET OPTIONS statement).

CDIGITS=*number*

NLCDIGITS=*number*

specifies the expected number of decimal digits of accuracy for nonlinear constraints, where *number* is any positive number, including fractions. PROC OPTMODEL uses this option to choose a step length when numeric derivative approximations are required to evaluate the Jacobian of nonlinear constraints. The default value depends on your operating environment. It is assumed that constraint values are accurate to the limits of machine precision.

For more information about numeric derivative approximations, see the section "Automatic Differentiation" on page 155.

ERRORLIMIT=*number* | **NONE**

specifies the maximum number of error messages that can be displayed during SOLVE statement processing. Specifying a value of *number* in the range 1 to $2^{31} - 1$ sets a specific limit. Specifying ERRORLIMIT=NONE removes any existing limit. The default value is 10.

NOTE: Some errors abort processing immediately.

FD=FORWARD | CENTRAL

FINITEDIFF=FORWARD | CENTRAL

specifies the method to use to approximate numeric derivatives when analytic derivatives are unavailable. Most solvers require the derivatives of the objective and constraints. You can specify the following values:

FORWARD uses forward differences.

CENTRAL uses central differences.

By default, FD=FORWARD. For more information about numeric derivative approximations, see the section "Automatic Differentiation" on page 155.

FDIGITS=*number*

OBJDIGITS=*number*

specifies the expected number of decimal digits of accuracy for the objective function, where *number* is any positive number, including fractions. PROC OPTMODEL uses this option to choose a step length when numeric derivatives are required. The default value depends on your operating environment. It is assumed that objective function values are accurate to the limits of machine precision.

For more information about numeric derivative approximations, see the section "Automatic Differentiation" on page 155.

FORCEFD=NONE | OBJ | CON | ALL

FORCEFINITEDIFF=NONE | OBJ | CON | ALL

forces PROC OPTMODEL to use finite differences instead of analytic derivatives for the specified set of nonlinear expressions. This option can be useful with FCMP functions to provide more control over derivative computation. You can specify the following values:

ALL restricts all derivative computations to use finite differences.

CON restricts derivative computations for the nonlinear constraint expressions and any IMPVAR expressions they reference to use finite differences.

NONE requests analytic derivatives where they are available.

OBJ restricts derivative computations for the objective and any IMPVAR expressions it references to use finite differences.

By default, FORCEFD=NONE.

FORCEPRESOLVE=*number* | *string*

specifies whether PROC OPTMODEL can use the OPTMODEL presolver with the CLP, LP, MILP, and QP solvers. By default, the OPTMODEL presolver is disabled when PROC OPTMODEL solves linear problems or problems with predicates, or when the CLP, LP, MILP, or QP solver is specified in the SOLVE statement. Table 5.4 shows the valid values for this option.

Table 5.4 Values for the FORCEPRESOLVE= Option

number	*string*	**Description**
0	FALSE	Restores the default behavior.
1	TRUE	Enables PROC OPTMODEL to use the OPTMODEL presolver when the CLP, LP, MILP, or QP solver is specified in the SOLVE statement.

By default, FORCEPRESOLVE=0.

INITVAR

is equivalent to INITVAR=TRUE.

INITVAR=TRUE | FALSE
specifies whether to pass initial values for variables to the solver when the SOLVE statement is executed. The CLP, LP, and QP solvers always ignore initial values. The NLP solvers attempt to use specified initial values. The LSO and MILP solvers use initial values only if the PRIMALIN option is specified.

You can specify the following values:

TRUE	passes the current variable values.
FALSE	invokes the solver without any specific initial values for variables.

By default, INITVAR=TRUE.

INTFUZZ=*number*
specifies the tolerance for rounding the bounds on integer and binary variables to integer values. Bounds that differ from an integer by at most *number* are rounded to that integer. Otherwise, lower bounds are rounded up to the next greater integer and upper bounds are rounded down to the next lesser integer. The value of *number* can range between 0 and 0.5. The default value is 0.00001.

MAXLABLEN=*number*
MAXLABELLEN=*number*
specifies the maximum length for MPS row and column labels, where *number* is an integer from 8 to 256, inclusive. This option can also be used to control the length of row and column names that are displayed by solvers, such as those found in the LP solver iteration log. See also the description of the .label suffix in the section "Suffixes" on page 135. By default, MAXLABLEN=32.

MISSCHECK
is equivalent to MISSCHECK=TRUE.

MISSCHECK=TRUE | FALSE
specifies whether to perform detailed checking of missing values in expressions. You can specify the following values:

TRUE	produces a message each time PROC OPTMODEL evaluates an arithmetic operation or function that has missing value operands (except when the operation or function specifically supports missing values). This value can increase processing time.
FALSE	turns off the detailed reporting.

By default, MISSCHECK=FALSE.

MSGLIMIT=*number* **| NONE**
MESSAGELIMIT=*number* **| NONE**
specifies the maximum number of messages about certain issues that can be displayed during processing of a single top-level statement, such as a SOLVE or FOR statement. Specifying a value of *number* in the range 0 to $2^{31} - 1$ sets a specific limit. Specifying MSGLIMIT=NONE removes any existing limit. The default value is 25.

The limit is applied to notes and warnings for the following issues:

- arithmetic evaluation issues, such as division by zero
- function evaluation issues, such as invalid arguments
- problem generation and PROC OPTMODEL presolver issues
- duplicate members in set constructor and literal expressions
- string concatenation results truncated to the maximum string length
- duplicate READ DATA keys
- truncated data set column labels for the CREATE DATA statement
- misspelled keywords for an option value specified using a string expression
- unrecognized file specifications in the CLOSEFILE statement

NOTE: Because of complications of concurrent execution, PROC OPTMODEL might display more or fewer messages than the limit when you use a COFOR statement.

NLCDIGITS=*number*
is an alias for the CDIGITS= option.

NOINITVAR
is equivalent to INITVAR=FALSE.

NOMISSCHECK
is equivalent to MISSCHECK=FALSE.

NTHREADS=*number*

NUMTHREADS=*number*
specifies the maximum number of threads to use for threaded processing, where *number* must be an integer between 1 and 256, inclusive. Problem generation and many of the solvers can take advantage of threaded processing. The default depends on the execution environment. When the distributed computing environment is used, the default is the maximum number of threads that are allowed on each computing node. Otherwise the default depends on the SAS system options CPUCOUNT= and THREADS. For more information, see the section "Threaded and Distributed Processing" on page 161.

OBJDIGITS=*number*
is an alias for the FDIGITS= option.

PDIGITS=*number*

PRINTDIGITS=*number*
requests that the PRINT statement display *number* significant digits for numeric columns for which no format is specified. The value can range from 1 to 9. By default, PDIGITS=5.

PMATRIX=*number*

PRINTMATRIX=*number*
adjusts the density evaluation of a two-dimensional array to affect how it is displayed, where *number* can be any nonnegative value. The value *number* scales the total number of nonempty array elements and is used by the PRINT statement to evaluate whether a two-dimensional array is "sparse" or "dense." Tables that contain a single two-dimensional array are printed in list form if they are sparse and in matrix form if they are dense. If $number < 1$, the list form is used in more cases. If $number > 1$, the matrix form to be used in more cases. If $number = 0$, the list form is always used. For more information, see the section "PRINT Statement" on page 75. By default, PMATRIX=1.

PRESOLVER=*number* | *string*

specifies the type of presolve that the OPTMODEL presolver performs. Table 5.5 shows the valid values of this option.

Table 5.5 Values for the PRESOLVER= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies presolver using default setting.
0	NONE	Disables presolver.
1	BASIC	Performs minimal processing, only substituting fixed variables and removing empty feasible constraints.
2	MODERATE	Applies a higher level of presolve processing.
3	AGGRESSIVE	Applies the highest level of presolve processing.

The OPTMODEL presolver tightens variable bounds and eliminates redundant constraints. In general, this tightening improves the performance of any solver. Higher levels of presolve processing allow more tightening and substitution passes, but might take more time to execute. The AUTOMATIC option is intermediate between the MODERATE and AGGRESSIVE levels.

NOTE: The OPTMODEL presolver is normally bypassed when PROC OPTMODEL uses the CLP, LP, QP, MILP, or network solver and when the SAVE MPS and SAVE QPS statements execute. The FORCEPRESOLVE= option enables the OPTMODEL presolver to be used with the CLP, LP, QP, and MILP solvers. PROC OPTMODEL always bypasses the OPTMODEL presolver when you specify certain solver options. For more information, see the chapter for the relevant solver in this book.

PRESTOL=*number*

PRESOLVETOL=*number*

specifies a tolerance (where *number* can range between 0 and 0.1) so that slightly infeasible constraints can be eliminated by the PROC OPTMODEL presolver. If the magnitude of the infeasibility is no greater than num($|X| + 1$), where X is the value of the original bound, then the empty constraint is removed from the presolved problem. PROC OPTMODEL's presolver does not print messages about infeasible constraints and variable bounds when the infeasibility is within the tolerance that is specified by *number*. The default value is 1E–12.

PRINTLEVEL=*number*

controls the level of listing output during a SOLVE or COFOR command. The Output Delivery System (ODS) tables printed at each level are listed in Table 5.6. Some solvers can produce additional tables; see the individual solver chapters for more information.

Table 5.6 Values for the PRINTLEVEL= Option

number	**Description**
0	Disables all tables
1	Prints Problem Summary and Solution Summary
2	Prints Problem Summary, Solution Summary, Methods of Derivative Computation (for NLP solvers), Solver Options, Optimization Statistics, Timing, and solver-specific ODS tables

For more information about the ODS tables produced by PROC OPTMODEL, see the section "ODS Table and Variable Names" on page 125.

PWIDTH=*number*

PRINTWIDTH=*number*

sets the width that is used by the PRINT statement to display numeric columns when no format is specified. The smallest value *number* can take is the value of the PDIGITS= option plus 7; the largest value *number* can take is 16. The default value is equal to the value of the PDIGITS= option plus 7.

VARFUZZ=*number*

specifies the smallest difference that is permitted by the OPTMODEL presolver between the upper and lower bounds of an unfixed variable. If the difference is smaller than *number*, then the variable is fixed to the average of the upper and lower bounds before it is presented to the solver. Any nonnegative value can be assigned to *number*; the default value is 0.

Declaration Statements

The declaration statements define the parameters, variables, constraints, and objectives that describe a PROC OPTMODEL optimization model. Declarations in the PROC OPTMODEL input are saved for later use. Unlike programming statements, declarations cannot be nested in other statements. Declaration statements are terminated by a semicolon.

Many declaration attributes, such as variable bounds, are defined using expressions. Expressions in declarations are handled symbolically and are resolved as needed. In particular, expressions are generally reevaluated when one of the parameter values they use has been changed.

CONSTRAINT Declaration

CONSTRAINT *constraint* [**,** *. . . constraint*] **;**

CON *constraint* [**,** *. . . constraint*] **;**

The constraint declaration defines one or more constraints on expressions in terms of the optimization variables. You can specify multiple constraint declaration statements.

Constraints can have an upper bound, a lower bound, or both bounds. The allowed forms are as follows:

[*name* [**{** *index-set* **}**] **:**] *expression* **=** *expression* [**SUFFIXES=(** *values* **)**]

declares an equality constraint or, when an *index-set* is specified, a family of equality constraints. The solver attempts to assign values to the optimization variables to make the two expressions equal.

[*name* [**{** *index-set* **}**] **:**] *expression* ~**=** *expression* [**SUFFIXES=(** *values* **)**]

declares a disequality constraint or, when an *index-set* is specified, a family of disequality constraints. The solver attempts to assign values to the optimization variables to make the two expressions unequal. The CLP solver must be used with this type of constraint.

[*name* [**{** *index-set* **}**] **:**] *expression relation expression* [**SUFFIXES=(** *values* **)**]

declares an inequality constraint that has a single upper or lower bound. *index-set* declares a family of inequality constraints. *relation* is the <=, <, >=, or > operator. When *relation*

is the <= operator, the solver tries to assign optimization variable values so that the value of the left *expression* is less than or equal to the value of the right *expression*. When *relation* is the < operator, the solver tries to assign optimization variable values so that the value of the left *expression* is less than the value of the right *expression*. When *relation* is the >= operator, the solver tries to assign optimization variable values so that the value of the left *expression* is greater than or equal to the value of the right *expression*. When *relation* is the > operator, the solver tries to assign optimization variable values so that the value of the left *expression* is greater than the value of the right *expression*. The CLP solver must be used when the < or > operator is specified.

[*name* [**{** *index-set* **}**] **:**] *bound relation body relation bound* [**SUFFIXES=(** *values* **)**]
declares an inequality constraint that is bounded on both sides, called a range constraint. *index-set* declares a family of range constraints. *relation* is the <=, <, >=, or > operator. Both *relation* operators must match in direction. If the <= or < operator is used in the first position, then a <= or < operator must be used in the second position. If the >= or > operator is used in the first position, then a >= or > operator must be used in the second position. The first *bound* expression defines the lower bound (if the <= or < operator is used) or the upper bound (if the >= or > operator is used). The second *bound* defines the upper bound (if the <= or < operator is used) or the lower bound (if the >= or > operator is used). The solver tries to assign optimization variables so that the value of the *body* expression is in the range between the upper and lower bounds. The CLP solver must be used when the < or > operator is specified.

[*name* [**{** *index-set* **}**] **:**] *predicate* [**SUFFIXES=(** *values* **)**]
declares a predicate constraint for the CLP solver. See the section "Predicates" on page 203 in Chapter 6, "The Constraint Programming Solver," for a description of the syntax and meaning of *predicates*.

NOTE: You can use the alternate forms from Table 5.10 for the relational operators.

name defines the name for the constraint. Use the name to reference constraint attributes, such as the bounds, elsewhere in the PROC OPTMODEL model. If no name is provided, then a default name is created of the form _ACON_[*n*], where *n* is an integer. See the section "Constraints" on page 131 for more information.

The SUFFIXES=() clause enables you to specify certain auxiliary values in the constraint declaration. (For more information, see the section "Suffixes" on page 135.) Dummy parameters that you declare in an *index-set* specification after the constraint *name* can be used in *value* expressions.

Here is a simple example that defines a constraint with a lower bound:

```
proc optmodel;
   var x, y;
   number low;
   con a: x+y >= low;
```

The following example adds an upper bound:

```
   var x, y;
   number low;
   con a: low <= x+y <= low+10;
```

Indexed families of constraints can be defined by specifying an *index-set* after the name. Any dummy parameters that are declared in the *index-set* can be referenced in the expressions that define the constraint.

A particular member of an indexed family can be specified by using an *identifier-expression* with a bracketed index list, in the same fashion as array parameters and variables. For example, the following statements create an indexed family of constraints named incr:

```
proc optmodel;
   number n;
   var x{1..n}
   /* require nondecreasing x values */
   con incr{i in 1..n-1}: x[i+1] >= x[i];
```

The CON statement in the example creates constraints incr[1] through incr[n–1].

Constraint expressions cannot be defined using functions that return different values each time they are called. See the section "Indexing" on page 95 for details.

IMPVAR Declaration

IMPVAR *impvar-decl* [**,** *. . . impvar-decl*] **;**

The IMPVAR declaration specifies one or more names that refer to optimization expressions in the model. The declared name is called an implicit variable. An implicit variable is useful for structuring models so that complex expressions do not need to be repeated each time they are used. The value of an implicit variable needs to be computed only once instead of at each place where the original expression is used, which helps reduce computational overhead. Implicit variables are evaluated without intervention from the solver.

Multiple IMPVAR declarations are allowed. The names of implicit variables must be distinct from other model declarations, such as variables and constraints. Implicit variables can be used in model expressions in the same places where ordinary variables are allowed. When the defining expression does not depend on optimization variables, you can also use implicit variables in constant expressions.

This is the syntax for an *impvar-decl*:

name [**{** *index-set* **}**] **=** *expression*

Each *impvar-decl* declares a name for an implicit variable. The name can be followed by an *index-set* specification to declare a family of implicit variables. The *expression* that the name refers to follows. Dummy parameters that are declared in the *index-set* specification can be used in the expression. The *expression* can refer to other model components, including variables, the current implicit variable, and other implicit variables.

As an example, in the following model statements the implicit variable total_weight is used in multiple constraints to set a limit on various product quantities, represented by locations in array x:

```
impvar total_weight = sum{p in PRODUCTS} Weight[p]*x[p];

con prod1_limit: Weight['Prod1'] * x['Prod1'] <= 0.3 * total_weight;
con prod2_limit: Weight['Prod2'] * x['Prod2'] <= 0.25 * total_weight;
```

MAX and MIN Objective Declarations

MAX *name* [**{** *index-set* **}**] **=** *expression* [**SUFFIXES=(** *values* **)**] **;**

MIN *name* [**{** *index-set* **}**] **=** *expression* [**SUFFIXES=(** *values* **)**] **;**

The MAX or MIN declaration specifies an objective for the solver. The *name* names the objective function for later reference. When an objective declaration is read, the declaration is added to the current problem.

The solver maximizes an objective that is specified with the MAX keyword and minimizes an objective that is specified with the MIN keyword. An objective is not allowed to have the same name as a parameter or variable. Multiple objectives are permitted, but only the LSO solver can process more than one objective at a time. Other solvers consider only the last objective that is added after array objectives are expanded.

The *expression* specifies the numeric function to maximize or minimize in terms of the optimization variables. Specify an *index-set* to declare a family of objectives. Dummy parameters that you declare in the *index-set* specification can be used in the *expression* and *values* that follow it. The SUFFIXES=() clause enables you to specify certain auxiliary values in the objective declaration. For more information, see the section "Suffixes" on page 135.

Objectives can also be used as implicit variables. When used in an expression, an objective name refers to the current value of the named objective function. The value of an unsuffixed objective name can depend on the value of optimization variables, so objective names cannot be used in constant expressions such as variable bounds. You can reference objective names in objective or constraint expressions. For example, the following statements declare two objective names, q and l, which are immediately referred to in the objective declaration of z and the declarations of the constraints:

```
proc optmodel;
   var x, y;
   min q=(x+y)**2;
   max l=x+2*y;
   min z=q+l;
   con c1: q<=4;
   con c2: l>=2;
```

Objectives cannot be defined using functions that return different values each time they are called. See the section "Indexing" on page 95 for details.

NUMBER, STRING, and SET Parameter Declarations

NUMBER *parameter-decl* [**,** *. . . parameter-decl*] **;**

STRING *parameter-decl* [**,** *. . . parameter-decl*] **;**

SET [**<** *scalar-type*, *. . . scalar-type* **>**] *parameter-decl* [**,** *. . . parameter-decl*] **;**

Parameters provide names for constants. Parameters are declared by specifying the parameter type followed by a list of parameter names. Declarations of parameters that have NUMBER or STRING types start with a *scalar-type* specification:

NUMBER | NUM ;

STRING | STR ;

The NUM and STR keywords are abbreviations for the NUMBER and STRING keywords, respectively.

The declaration of a parameter that has the set type begins with a *set-type* specification:

SET [**<** *scalar-type*, *. . . scalar-type* **>**] **;**

In a *set-type* declaration, the SET keyword is followed by a list of *scalar-type* items that specify the member type. A set with scalar members is specified with a single *scalar-type* item. A set with tuple members has a *scalar-type* item for each tuple element. The *scalar-type* items specify the types of the elements at each tuple position.

If the SET keyword is not followed by a list of *scalar-type* items, then the set type is determined from the type of the initialization expression. The declared type defaults to SET<NUMBER> if no initialization expression is given or if the expression type cannot be determined.

For any parameter type, the type declaration is followed by a list of *parameter-decl* items that specify the names of the parameters to declare. In a *parameter-decl* item the parameter name can be followed by an optional index specification and any necessary options, as follows:

name [**{** *index-set* **}**] [*parameter-options*]

The parameter *name* and *index-set* can be followed by a list of *parameter-options*. Dummy parameters declared in the *index-set* can be used in the *parameter-options*. The parameter options can be specified with the following forms:

= *expression*
: provides an explicit value for each parameter location. In this case the parameter acts like an alias for the *expression* value.

INIT *expression*
: specifies a default value that is used when a parameter value is required but no other value has been supplied. For example:

```
number n init 1;
set s init {'a', 'b', 'c'};
```

PROC OPTMODEL evaluates the expression for each parameter location the first time the parameter needs to be resolved. The expression is not used when the parameter already has a value.

= [*initializers* **]**
: provides a compact means to define the values for an array, in which each array location value can be individually specified by the *initializers*.

INIT [*initializers* **]**
: provides a compact means to define multiple default values for an array. Each array location value can be individually specified by the *initializers*. With this option the array values can still be updated outside the declaration.

The =*expression* parameter option defines a parameter value by using a formula. The formula can refer to other parameters. The parameter value is updated when the referenced parameters change. The following example shows the effects of the update:

```
proc optmodel;
   number n;
   set<number> s = 1..n;
   number a{s};
   n = 3;
   a[1] = 2;    /* OK */
   a[7] = 19;   /* error, 7 is not in s */
   n = 10;
   a[7] = 19;   /* OK now */
```

In the preceding example the value of set s is resolved for each use of array a that has an index. For the first use of a[7], the value 7 is not a member of the set s. However, the value 7 is a member of s at the second use of a[7].

The INIT *expression* parameter option specifies a default value for a parameter. The following example shows the usage of this option:

```
proc optmodel;
   num a{i in 1..2} init i**2;
   a[1] = 2;
   put a[*]=;
```

When the value of a parameter is needed but no other value has been supplied, the default value specified by INIT *expression* is used, as shown in Figure 5.5.

Figure 5.5 INIT Option: Output

```
a[1]=2 a[2]=4
```

NOTE: Parameter values can also be read from files or specified with assignment statements. However, the value of a parameter that is assigned with the =*expression* or =[*initializers*] forms can be changed only by modifying the parameters used in the defining expressions. Parameter values specified by the INIT option can be reassigned freely.

Initializing Arrays

Arrays can be initialized with the =[*initializers*] or INIT [*initializers*] forms. These forms are convenient when array location values need to be individually specified. The forms behave the same way, except that the INIT [*initializers*] form allows the array values to be modified after the declaration. These forms of initialization are used in the following statements:

```
proc optmodel;
   number a{1..3} = [5 4 7];
   number b{1..3} INIT [5 4 7];
   put a[*]=;
   b[1] = 1;
   put b[*]=;
```

Each array location receives a different value, as shown in Figure 5.6. The displayed values for b are a combination of the default values from the declaration and the assigned value in the statements.

Figure 5.6 Array Initialization

```
a[1]=5 a[2]=4 a[3]=7
b[1]=1 b[2]=4 b[3]=7
```

Each *initializer* takes the following form:

[**[** *index* **]**] *value*

The *value* specifies the value of an array location and can be a numeric or string constant, a set literal, or an expression enclosed in parentheses.

In array initializers, string constants can be specified using quoted strings. When the string text follows the rules for a SAS name, the text can also be specified without quotation marks. String constants that begin with a digit, contain blanks, or contain other special characters must be specified with a quoted string.

As an example, the following statements define an array parameter that could be used to map numeric days of the week to text strings:

```
proc optmodel;
   string dn{1..5} =
          [Monday Tuesday Wednesday Thursday Friday];
```

The optional *index* in square brackets specifies the index of the array location to initialize. The index specifies one or more numeric or string subscripts. The subscripts allow the same syntactic forms as the *value* items. Commas can be used to separate index subscripts. For example, location a[1,'abc'] of an array a could be specified with the index [1 abc]. The following example initializes just the diagonal locations in a square array:

```
proc optmodel;
   number m{1..3,1..3} = [[1 1] 0.1 [2 2] 0.2 [3 3] 0.3];
```

An index does not need to specify all the subscripts of an array location. If the index begins with a comma, then only the rightmost subscripts of the index need to be specified. The preceding subscripts are supplied from the index that was used by the preceding *initializer*. This can simplify the initialization of arrays that are indexed by multiple subscripts. For example, you can add new entries to the matrix of the previous example by using the following statements:

```
proc optmodel;
   number m{1..3,1..3} = [[1 1] 0.1            [,3] 1
                                  [2 2] 0.2    [,3] 2
                                             [3 3] 0.3];
```

The spacing shows the layout of the example array. The previous example was updated by initializing two more values at m[1,3] and m[2,3].

If an index is omitted, then the next location in the order of the array's index set is initialized. If the index set has multiple *index-set-items*, then the rightmost indices are updated before indices to the left are updated. At the beginning of the initializer list, the rightmost index is the first member of the index set. The index set must use a range expression to avoid unpredictable results when an index value is omitted.

The initializers can be followed by commas. The use of commas has no effect on the initialization. The comma can be used to clarify layout. For example, the comma could separate rows in a matrix.

Not every array location needs to be initialized. The locations without an explicit initializer are set to missing for numeric arrays, set to an empty string for string arrays, and set to an empty set for set arrays. You can also specify a different default value in the last initializer by using an asterisk (*) in place of the *index*. For example:

```
proc optmodel;
   /* initialize to [2 7 30 40] */
   number a{i in 1..4} = [2 7 [*] (i*10)];
```

NOTE: An array location must not be initialized more than once during the processing of the initializer list.

PROBLEM Declaration

PROBLEM *name* [**{** *index-set* **}**] [**FROM** *problem-id*] [**INCLUDE** *problem-items*] [**SUFFIXES=(** *values* **)**] **;**

Problems are declared with the PROBLEM declaration. Problem declarations track objectives, a set of included variables and constraints, and some status information that is associated with the variables and constraints. The problem name can optionally be followed by an *index-set* to create a family of problems. When a problem is first used (via the USE PROBLEM statement), the specifications from the optional FROM and INCLUDE clauses create the initial set of included variables, constraints, and the problem objectives. An empty problem is created if neither clause is specified.

The FROM clause specifies an existing problem from which to copy the set of included symbols. The *problem-id* is an *identifier expression*. The dropped and fixed status for these symbols in the specified problem is also copied. Subsequent changes to the FROM problem via the DROP, FIX, RESTORE, and UNFIX statements do not affect the new problem status.

The INCLUDE clause specifies a list of variables, constraints, and objectives to include in the problem. These items are included with default status (unfixed and undropped) which overrides the status from the FROM clause, if it exists. Each item is specified with one of the following forms:

identifier-expression
: includes the specified items in the problem. The *identifier-expression* can be a symbol name or an array symbol with explicit index. If an array symbol is used without an index, then all array elements are included.

{ *index-set* **}** *identifier-expression*
: includes the specified subset of items in the problem. The item specified by the *identifier-expression* is added to the problem for each member of the *index-set*. The dummy parameters from the *index-set* can be used in the indexing of the *identifier-expression*. If the *identifier-expression* is an array symbol without indexing, then the *index-set* provides the indices for the included locations.

The set of elements that are included by an INCLUDE clause is updated when array indexing for the *identifier-expression* changes or when parameters referenced in the INCLUDE clause change. However, status changes from the DROP, FIX, RESTORE, and UNFIX statements override INCLUDE clauses, including clauses that are inherited from a FROM problem.

You can use the FROM and INCLUDE clauses to designate the objectives for a problem. The objectives are copied from the problem that is designated by the FROM clause, if present. Then the INCLUDE clause, if any, is applied. The LSO solver can process multiple objectives. Other solvers use only the last objective that is specified.

The SUFFIXES=() clause enables you to specify certain auxiliary values in the problem declaration. (For more information, see the section "Suffixes" on page 135.) Dummy parameters that you declare in an *index-set* specification for the problem *name* can be used in the *value* expressions.

The following statements declare some problems with a variable x and different objectives to illustrate some of the ways of including model components. Note that the statements use the predeclared problem _START_ to avoid resetting the objective in prob2 when the objective z3 is declared.

```
proc optmodel;
   problem prob1;
   use problem prob1;
   var x >= 0;              /* included in prob1 */
   min z1 = (x-1)**2;       /* included in prob1 */
   expand;                  /* prob1 contains x, z1 */

   problem prob2 from prob1;
   use problem prob2;       /* includes x, z1 */
   min z2 = (x-2)**2;       /* included in prob2 */
   expand;                  /* prob2 contains x, z1, z2 */
                            /* note z1 is not expanded */

   use problem _start_;     /* don't modify prob2 */
   min z3 = (x-3)**2;
   problem prob3 include x z3;
   use problem prob3;
   expand;                  /* prob3 contains x, z3 */
```

See the section "Multiple Subproblems" on page 151 for more details about problem processing.

VAR Declaration

VAR *var-decl* [**,** *. . . var-decl*] **;**

The VAR statement declares one or more optimization variables. Multiple VAR statements are permitted. A variable is not allowed to have the same name as a parameter or constraint.

Each *var-decl* specifies a variable name. The name can be followed by an array *index-set* specification and then variable options. Dummy parameters declared in the index set specification can be used in the following variable options.

Here is the syntax for a *var-decl*:

name [**{** *index-set* **}**] [*var-options*] [**SUFFIXES=(** *values* **)**]

For example, the following statements declare a group of 100 variables, x[1]–x[100]:

```
proc optmodel;
   var x{1..100};
```

Here are the available variable options:

INIT *expression*
: sets an initial value for the variable. The expression is used only the first time the value is required. If no initial value is specified, then 0 is used by default.

>= *expression*
: sets a lower bound for the variable value. The default lower bound is $-\infty$, or 0 if the BINARY option is used.

<= *expression*
: sets an upper bound for the variable value. The default upper bound is ∞, or 1 if the BINARY option is used.

INTEGER

requests that the solver assign the variable an integer value.

BINARY

requests that the solver assign the variable an integer value within default bounds of 0 to 1.

Dummy parameters that you declare in an *index-set* specification for the variable *name* can be used in the variable declaration expressions, including the SUFFIXES=() clause. The SUFFIXES=() clause enables you to specify certain auxiliary values in the declaration. For more information, see the section "Suffixes" on page 135.

For example, the following statements declare a variable that has an initial value of 0.5. The variable is bounded by 0 and 1:

```
proc optmodel;
   var x init 0.5 >= 0 <= 1;
```

The values of the bounds can be determined later by using suffixed references to the variable. For example, the upper bound for variable x can be referred to as x.ub. In addition, the bounds options can be overridden by explicit assignment to the suffixed variable name. Suffixes are described further in the section "Suffixes" on page 135. Note that the bounds for integer and binary variables are rounded to integers according to the value that you specify in the PROC OPTMODEL option INTFUZZ=.

When used in an expression, an unsuffixed variable name refers to the current value of the variable. Unsuffixed variables are not allowed in the expressions for options that define variable bounds or initial values. Such expressions have values that must be fixed during execution of the solver.

Programming Statements

PROC OPTMODEL supports several programming statements. You can perform various actions with these statements, such as reading or writing data sets, setting parameter values, generating text output, or invoking a solver.

Statements are read from the input and are executed immediately when complete. Certain statements can contain one or more substatements. The execution of substatements is held until the statements that contain them are submitted. Parameter values that are used by expressions in programming statements are resolved when the statement is executed; this resolution might cause errors to be detected. For example, the use of undefined parameters is detected during resolution of the symbolic expressions from declarations.

A statement is terminated by a semicolon. The positions at which semicolons are placed are shown explicitly in the following statement syntax descriptions.

The programming statements can be grouped into the categories shown in Table 5.7.

Table 5.7 Types of Programming Statements in PROC OPTMODEL

Control	Looping	General	Input/Output	Model
DO	COFOR	Assignment	CLOSEFILE	DROP
IF	CONTINUE	CALL	CREATE DATA	EXPAND
Null (;)	DO Iterative	PROFILE	FILE	FIX
QUIT	DO UNTIL	RESET OPTIONS	PRINT	RESTORE
STOP	DO WHILE	SUBMIT	PUT	SOLVE
	FOR		READ DATA	UNFIX
	LEAVE		SAVE MPS	USE PROBLEM
			SAVE QPS	

Assignment Statement

identifier-expression = *expression* **;**

The assignment statement assigns a variable or parameter value. The type of the target *identifier-expression* must match the type of the right-hand-side expression.

For example, the following statements set the current value for variable x to 3:

```
proc optmodel;
   var x;
   x = 3;
```

NOTE: Parameters that were declared with the equal sign (=) initialization forms must not be reassigned a value with an assignment statement. If this occurs, PROC OPTMODEL reports an error.

CALL Statement

CALL *name* **(** *argument-1* [**,** *. . . argument-n*] **) ;**

The CALL statement invokes the named library subroutine. The values that are determined for each argument expression are passed to the subroutine when the subroutine is invoked. The subroutine can update the values of PROC OPTMODEL parameters and variables when an argument is an *identifier-expression* (see the section "Identifier Expressions" on page 101). For example, the following statements set the parameter array a to a random permutation of 1 to 4:

```
proc optmodel;
   number a{i in 1..4} init i;
   number seed init -1;
   call ranperm(seed, a[1], a[2], a[3], a[4]);
```

NOTE: The maximum length of the string value returned from an output argument is equal to the character length of the argument before the call. An undefined STRING parameter that is used as an output argument has a character length of 8.

For a list of CALL routines, see *SAS Functions and CALL Routines: Reference*. You can also call subroutines that are compiled by the FCMP procedure. For more information, see the section "FCMP Routines" on page 157.

CLOSEFILE Statement

CLOSEFILE *file-specifications* **;**

The CLOSEFILE statement closes files that were opened by the FILE statement. Each file is specified by a logical name, a physical file name in quotation marks, or an expression enclosed in parentheses that evaluates to a physical file name. See the section "FILE Statement" on page 70 for more information about file specifications.

The following example shows how the CLOSEFILE statement is used with a logical file name:

```
filename greet 'hello.txt';
proc optmodel;
   file greet;
   put 'Hi!';
   closefile greet;
```

Generally you must close a file with a CLOSEFILE statement before external programs can access the file. However, any open files are automatically closed when PROC OPTMODEL terminates.

COFOR Statement

COFOR { *index-set* **}** *statement* **;**

The COFOR statement executes its *statement* for each member of the specified *index-set*, similar to how the FOR statement executes. However, in a COFOR statement, PROC OPTMODEL can execute the SOLVE statement concurrently with other statements. The execution of the COFOR substatement is interleaved between loop iterations so that other iterations can be processed while an iteration waits for a SOLVE statement to complete. Multiple solvers can run concurrently. This interleaving is managed so that in many cases a FOR loop can be replaced by a COFOR loop to achieve concurrency with minimal or no other changes to the code.

The following code shows a simple example:

```
proc optmodel printlevel=0;
   var x {1..6} >= 0;

   minimize z = sum {j in 1..6} x[j];

   con a1: x[1] + x[2] + x[3]                        <= 4;
   con a2:                    x[4] + x[5] + x[6] <= 6;
   con a3: x[1] +             x[4]               >= 5;
   con a4:        x[2] +             x[5]        >= 2;
   con a5:               x[3] +             x[6] >= 3;

   cofor{i in 3..5} do;
      fix x[1]=i;
      solve;
      put i= x[1]= _solution_status_=;
   end;
```

Figure 5.7 shows the PROC OPTMODEL output. The order of the output from different iterations can vary between runs, depending on the order in which the SOLVE statements complete. A FOR statement could have been used instead of COFOR; the FOR statement would produce a consistent output order but only one

solver would execute at a time. Note that because the solver execution in this example is trivial, the benefits from concurrency are limited.

Figure 5.7 A Simple COFOR Loop

```
i=4 x[1]=4 _SOLUTION_STATUS_=OPTIMAL
i=5 x[1]=5 _SOLUTION_STATUS_=INFEASIBLE
i=3 x[1]=3 _SOLUTION_STATUS_=OPTIMAL
```

A COFOR statement can contain other control and looping statements, including nested COFOR loops. The maximum number of threads that can be used is controlled by the PROC OPTMODEL and SAS options that are in effect when the outermost COFOR loop is entered, as described in the section "Threaded and Distributed Processing" on page 161. The outermost COFOR statement allocates threads for execution on the computer that is running PROC OPTMODEL. When you specify options to request distributed computing, the outermost COFOR statement also creates a distributed execution environment. Solvers within the COFOR loop can then run remotely in single-machine mode on the compute nodes (as shown in the solver output).

The COFOR statement supports simultaneous processing of several SOLVE statements. Processing proceeds through the iteration body statements as it would through a FOR loop until a SOLVE statement that uses the CLP, LP, LSO, MILP, network, NLP, or QP solver is executed. After the problem is generated, the solver starts processing in a background thread (or remote computing node in the distributed case) and the COFOR loop switches execution to another iteration of the loop, assuming enough threads and iterations are available. (Note that you need at least two threads on the computer that is running the COFOR loop to enable overlap of statement execution with solver execution.) Execution could switch to an existing iteration where the solver has completed. Alternatively, a new iteration of a COFOR loop could be started. All output from an iteration, except within a SUBMIT block, is displayed together after the iteration has completed. Output from a SUBMIT block is displayed as the block is executed.

You can change the NTHREADS= option value within a COFOR loop by using the RESET OPTIONS statement. The new value affects SOLVE statements that are executed subsequently in the same iteration but not those in other iterations. When a COFOR statement is running in distributed mode, the default value of the NTHREADS= option is the number of threads per compute node, as determined when the outermost COFOR loop starts. Otherwise the default value of the NTHREADS= option is 1. Executing SOLVE statements in the background by using a single thread usually provides the best performance on a single computer.

The order in which the solvers complete is unpredictable. So it is usually not useful for a problem that is solved within an iteration to depend on the results of SOLVE statements that are executed in other iterations of the COFOR loop. It is advisable to limit global parameter updates to operations where order is not important, such as accumulating counts, sums, or unions or writing mutually exclusive subsets of an array. It is possible to execute multiple SOLVE statements within a loop iteration, and subsequent solver invocations within an iteration can use results from prior solvers in the same iteration.

In many cases, a COFOR loop iteration solves a specialized version of a common problem structure. This requires it to modify problem attributes that are also used in other iterations, such as coefficient values or the fixed status of variables. Changes to problem attributes are not made visible to other iterations of a COFOR loop in order to avoid confusing behavior due to interleaved execution. For example, the value printed for x[1] in Figure 5.7 is the local value for the iteration, not the most recent global value. Changes to these attributes create or update a copy of the value that is local to the iteration. These attribute values along with the local dummy parameters provide a local context for the iteration.

The following problem attributes are automatically made local to the modifying iteration when they are changed within a COFOR loop:

- the current problem, selected by USE PROBLEM
- the value of variables and their suffix values
- the fixed status of variables
- the constraint suffix values
- the dropped status of constraints
- the .LABEL suffix
- NUMBER, STRING, and SET parameters that determine values that are used in the bounds or body expressions of problem declarations (CONSTRAINT, IMPVAR, MIN, MAX, or VAR)
- NUMBER, STRING, and SET parameters that determine values that are used in solver arguments within the same outermost COFOR loop
- the predeclared string parameters _SOLVER_OPTIONS_ and _*solver*_OPTIONS_ (for each solver)

To illustrate these rules, consider the following code, which uses the NLP solver to solve a MINLP portfolio optimization problem by selecting random subsets of the assets to optimize:

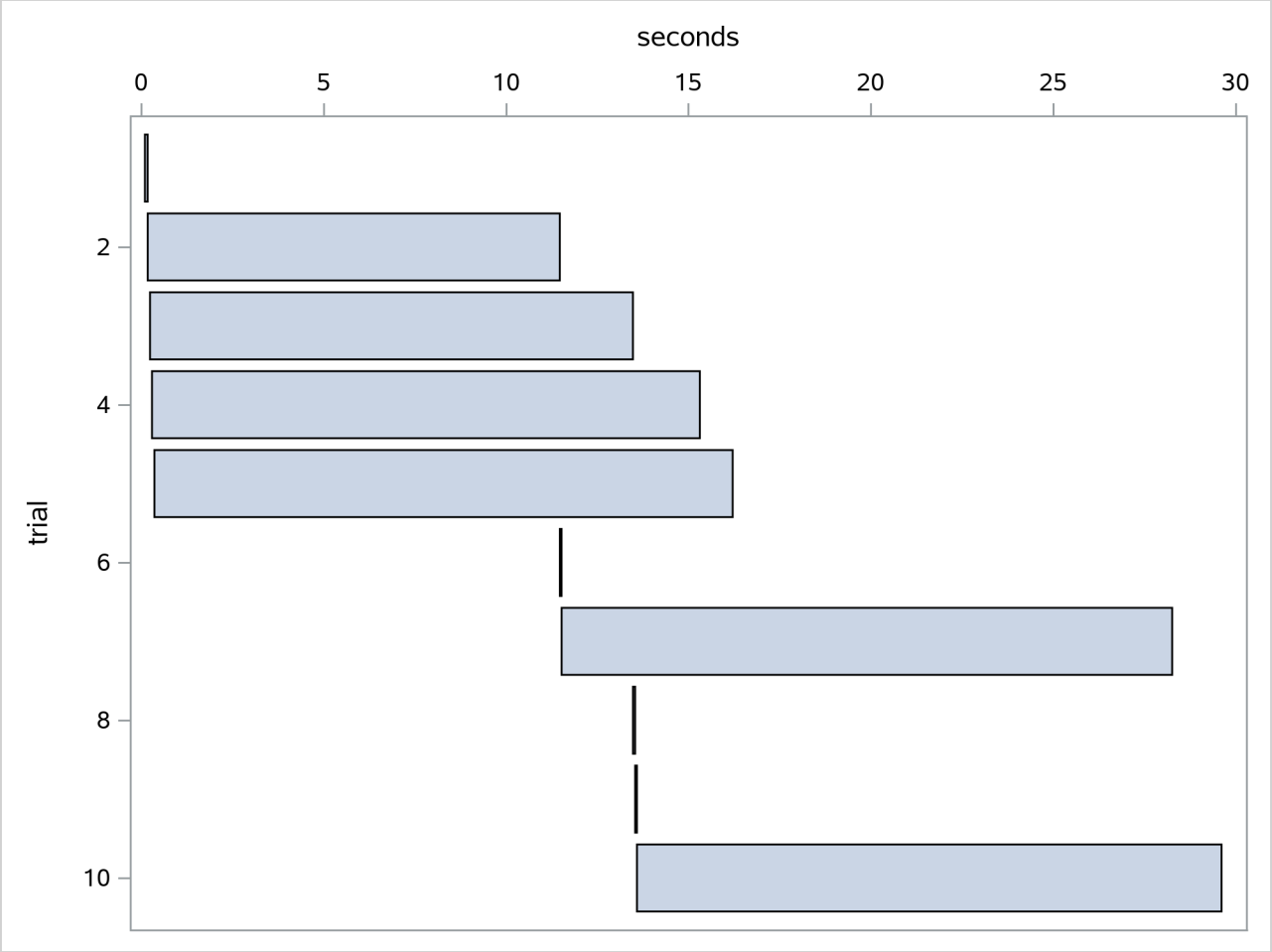
```
proc optmodel printlevel=0 nthreads=4;
   /* assets and related parameters */
   set ASSETS;
   num return {ASSETS};
   num cov {ASSETS, ASSETS} init 0;
   read data means into ASSETS=[_n_] return;
   read data covdata into [asset1 asset2] cov cov[asset2,asset1]=cov;
   num riskLimit init 0.00025;
   num minThreshold init 0.1;
   num numTrials = 10;

   /* number of random trials */
   set TRIALS = 1..numTrials;

   /* declare NLP problem for fixed set of assets */
   set ASSETS_THIS;
   var AssetPropVar {ASSETS} >= minThreshold <= 1;
   max ExpectedReturn = sum {i in ASSETS} return[i] * AssetPropVar[i];
   con RiskBound:
      sum {i in ASSETS_THIS, j in ASSETS_THIS}
          cov[i,j] * AssetPropVar[i] * AssetPropVar[j] <= riskLimit;
   con TotalPortfolio:
      sum {asset in ASSETS} AssetPropVar[asset] = 1;

   /* parameters to track best solution */
   num infinity = constant('BIG');
   num best_objective init -infinity;
   set INCUMBENT;

   /* iterate over trials */
   num start  {TRIALS};
```

```
   num finish {TRIALS};
   num overall_start;
   overall_start = time();
   call streaminit(1);
   cofor {trial in TRIALS} do;
      start[trial] = time() - overall_start;
      put;
      put trial=;
      ASSETS_THIS = {i in ASSETS: rand('UNIFORM') < 0.5};
      put ASSETS_THIS=;
      for {i in ASSETS diff ASSETS_THIS}
         fix AssetPropVar[i] = 0;
      solve with NLP / logfreq=0;
      put _solution_status_=;
      if _solution_status_ ne 'INFEASIBLE' then do;
         if best_objective < ExpectedReturn then do;
            best_objective = ExpectedReturn;
            INCUMBENT = ASSETS_THIS;
         end;
      end;
      finish[trial] = time() - overall_start;
   end;

   put best_objective= INCUMBENT=;
   create data plotdata from [trial] start finish;

proc sgplot data=plotdata;
   highlow y=trial low=start high=finish / type=bar x2axis;
   yaxis reverse;
   x2axis label="seconds";
run;
```

All the COFOR loop iterations use the same problem, _START_. However, the changes to the problem are local to the iteration that makes them. For example, the FIX statement does not affect variables in other iterations. The value of the ASSETS_THIS parameter is used by the RiskBound constraint, so the change to it is local. Because AssetPropVar is a VAR, the changes to its value are also local.

On the other hand, the values of the best_objective and INCUMBENT parameters do not affect any problem declarations. Therefore, their global values are used, enabling the code in the COFOR loop to select and save the best result. Similarly, the start and finish parameters are not used in the problem and allow the overlapping of iterations to be illustrated. Figure 5.8 from the SGPLOT procedure shows how the iterations have overlapped execution times.

Figure 5.8 Overlapped COFOR Iterations



Changes to problem attributes from completed iterations are made visible after the loop is finished. They appear in the context that contained the COFOR statement. If multiple iterations modify the same problem attribute value, then the value from the iteration that completed last is the one made visible.

The LEAVE statement can be used to terminate execution of a COFOR loop. This completes the current iteration of the COFOR loop. The currently active solvers for the COFOR loop are terminated, and the output of the incomplete iterations is discarded. The CONTINUE statement within a COFOR loop can also be used to complete the current iteration, but it has no effect on other iterations.

Using the LEAVE statement to terminate is useful, for example, when a sufficiently good solution is found for a problem. The preceding code has been modified as follows to keep generating solutions until a time limit is reached. The code sets a time limit and then executes the LEAVE statement to stop processing when the limit is exceeded. The COFOR loop uses a very large iteration range to allow it to run indefinitely.

```
proc optmodel printlevel=0;
   set ASSETS;
   num return {ASSETS};
   num cov {ASSETS, ASSETS} init 0;
   read data means into ASSETS=[_n_] return;
   read data covdata into [asset1 asset2] cov cov[asset2,asset1]=cov;
```

```
   num riskLimit init 0.00025;
   num minThreshold init 0.1;

   /* declare NLP problem for fixed set of assets */
   set ASSETS_THIS;
   var AssetPropVar {ASSETS} >= minThreshold <= 1;
   max ExpectedReturn = sum {i in ASSETS} return[i] * AssetPropVar[i];
   con RiskBound:
      sum {i in ASSETS_THIS, j in ASSETS_THIS}
          cov[i,j] * AssetPropVar[i] * AssetPropVar[j] <= riskLimit;
   con TotalPortfolio:
      sum {asset in ASSETS} AssetPropVar[asset] = 1;

   num infinity = constant('BIG');
   num best_objective init -infinity;
   set INCUMBENT;

   /* run for 30 seconds */
   num last_time;
   last_time = time() + 30;
   num n_trials init 0;
   call streaminit(1);
   cofor {trial in 1..1e9} do;
      put;
      put trial=;
      ASSETS_THIS = {i in ASSETS: rand('UNIFORM') < 0.5};
      put ASSETS_THIS=;
      for {i in ASSETS diff ASSETS_THIS} fix AssetPropVar[i] = 0;
      solve with NLP / logfreq=0;
      put _solution_status_=;
      if _solution_status_ ne 'INFEASIBLE' then do;
         if best_objective < ExpectedReturn then do;
            best_objective = ExpectedReturn;
            INCUMBENT = ASSETS_THIS;
         end;
      end;
      n_trials = n_trials + 1;
      if time() >= last_time then leave;
   end;

   put n_trials=;
   put best_objective= INCUMBENT=;
quit;
```

CONTINUE Statement

CONTINUE ;

The CONTINUE statement terminates the current iteration of the loop statement (iterative DO, DO UNTIL, DO WHILE, FOR, or COFOR) that immediately contains the CONTINUE statement. Execution resumes at the start of the loop after checking WHILE or UNTIL tests. The FOR, COFOR, or iterative DO loops apply new iteration values.

CREATE DATA Statement

CREATE DATA *SAS-data-set* **FROM** [**[** *key-columns* **]** [**=** *key-set*]] *columns* **;**

The CREATE DATA statement creates a new SAS data set and copies data into it from PROC OPTMODEL parameters and variables. The CREATE DATA statement can create a data set with a single observation or a data set with observations for every location in one or more arrays. The data set is closed after the execution of the CREATE DATA statement.

The arguments to the CREATE DATA statement are as follows:

SAS-data-set
: specifies the output data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses.

key-columns
: declares index values and their corresponding data set variables. The values are used to index array locations in *columns*.

key-set
: specifies a set of index values for the *key-columns*.

columns
: specifies data set variables as well as the PROC OPTMODEL source data for the variables.

Each *column* or *key-column* defines output data set variables and a data source for a column. For example, the following statement generates the output SAS data set resdata from the PROC OPTMODEL array opt, which is indexed by the set indset:

```
create data resdata from [solns]=indset opt;
```

The output data set variable solns contains the index elements in indset.

Columns

Columns can have the following forms:

identifier-expression [/ *options*]
: transfers data from the PROC OPTMODEL parameter or variable specified by the *identifier-expression*. The output data set variable has the same name as the *name* part of the *identifier-expression* (see the section "Identifier Expressions" on page 101). If the *identifier-expression* refers to an array, then the index can be omitted when it matches the *key-columns*. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 62 for more information. The following example creates a data set with the variables m and n:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from m n;
```

name **=** *expression* [/ *options*]
: transfers the value of a PROC OPTMODEL expression to the output data set variable *name*. The *expression* is reevaluated for each observation. If the *expression* contains any

operators or function calls, then it must be enclosed in parentheses. If the *expression* is an *identifier-expression* that refers to an array, then the index can be omitted if it matches the *key-columns*. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 62 for more information. The following example creates a data set with the variable ratio:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from ratio=(m/n);
```

COL(*name-expression***) =** *expression* [/ *options*]

transfers the value of a PROC OPTMODEL expression to the output data set variable named by the string expression *name-expression*. The PROC OPTMODEL expression is reevaluated for each observation. If this expression contains any operators or function calls, then it must be enclosed in parentheses. If the PROC OPTMODEL *expression* is an *identifier-expression* that refers to an array, then the index can be omitted if it matches the *key-columns*. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 62 for more information. The following example uses the COL expression to form the variable s5:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from col("s"||n)=(m+n);
```

{ *index-set* **} <** *columns* **>**

performs the transfers by iterating each column specified by **<** *columns* **>** for each member of the *index set*. If there are n columns and m index set members, then $n \times m$ columns are generated. The dummy parameters from the index set can be used in the columns to generate distinct output data set variable names in the iterated columns, using COL expressions. The columns are expanded when the CREATE DATA statement is executed, before any output is performed. This form of *columns* cannot be nested. In other words, the following form of *columns* is NOT allowed:

{ *index-set* **} < {** *index-set* **} <** *columns* **> >**

The following example demonstrates the use of the iterated *columns* form:

```
proc optmodel;
   set<string> alph = {'a', 'b', 'c'};
   var x{1..3, alph} init 2;
   create data example from [i]=(1..3)
      {j in alph}<col("x"||j)=x[i,j]>;
```

The data set created by these statements is shown in Figure 5.9.

Figure 5.9 CREATE DATA with COL Expression

Obs	i	xa	xb	xc
1	1	2	2	2
2	2	2	2	2
3	3	2	2	2

NOTE: When no *key-columns* are specified, the output data set has a single observation.

The following statements incorporate several of the preceding examples to create and print a data set by using PROC OPTMODEL parameters:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from m n ratio=(m/n) col("s"||n)=(m+n);

proc print;
run;
```

The output from the PRINT procedure is shown in Figure 5.10.

Figure 5.10 CREATE DATA for Single Observation

Obs	m	n	ratio	s5
1	7	5	1.4	12

Column Options

Each *column* or *key-column* that defines a data set variable can be followed by zero or more of the following modifiers:

FORMAT=*format.*
: associates a format with the current column.

INFORMAT=*informat.*
: associates an informat with the current column.

LABEL=*'label'*
: associates a label with the current column. The label can be specified by a quoted string or an expression in parentheses.

LENGTH=*length*
: specifies a length for the current column. The length can be specified by a numeric constant or a parenthesized expression. For character columns, the length specifies a character length. The character length is used to determine a byte length for the column so that any character string of the specified length can be represented in the output encoding. For example, specifying LENGTH=5 with UTF-8 encoding produces a byte length of 20, whereas a single-byte encoding produces a byte length of 5. For character columns, the minimum length is 1, and the maximum length depends on the data set engine. Output to fixed CHAR variables is limited to 32,767 bytes, but VARCHAR output can use up to $2^{31} - 1$ bytes. For numeric columns, the length specifies a byte length for the output variable. The range for numeric variables depends on the operating environment and has a minimum of 2 or 3.

TRANSCODE=YES | **NO**
: specifies whether character variables can be transcoded. The default value is YES. See the TRANSCODE=option of the ATTRIB statement in *SAS DATA Step Statements: Reference* for more information.

The following statements demonstrate the use of column options, including the use of multiple options for a single column:

```
proc optmodel;
   num sq{i in 1..10} = i*i;
   create data squares from [i/format=hex2./length=3] sq/format=6.2;

proc print;
run;
```

The output from the PRINT procedure is shown in Figure 5.11.

Figure 5.11 CREATE DATA with Column Options

Obs	i	sq
1	01	1.00
2	02	4.00
3	03	9.00
4	04	16.00
5	05	25.00
6	06	36.00
7	07	49.00
8	08	64.00
9	09	81.00
10	0A	100.00

Key Columns

Key-columns declare index values that enable multiple observations to be written from array *columns*. An observation is created for each unique index value combination. The index values supply the index for array *columns* that do not have an explicit index.

Key-columns define the data set variables where the index value elements are written. They can also declare local dummy parameters for use in expressions in the *columns*. *Key-columns* are syntactically similar to *columns*, but are more restricted in form. The following forms of *key-columns* are allowed:

name [/ *options*]
: transfers an index element value to the data set variable *name*. A local dummy parameter, *name*, is declared to hold the index element value. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 62 for more information.

COL(*name-expression***)** [**=** *index-name*] [/ *options*]
: transfers an index element value to the data set variable named by the string-valued *name-expression*. The argument *index-name* optionally declares a local dummy parameter to hold the index element value. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 62 for more information.

A *key-set* in the CREATE DATA statement explicitly specifies the set of index values. You can specify *key-set* as an identifier expression, an index set expression, a set constructor expression, or an arbitrary set expression enclosed in parentheses. When you specify *key-set* as an index set expression, the *index-set* dummy parameters remain visible in *columns* and override dummy parameters that are declared in the *key-columns* items with the same name.

The following statements create a data set from the PROC OPTMODEL parameter m, a matrix whose only nonzero entries are located at (1, 1) and (4, 1):

```
proc optmodel;
   number m{1..5, 1..3} = [[1 1] 1 [4 1] 1 [*] 0];
   set ISET = setof{i in 1..2} i**2;
   create data example
      from [i j] = {i in ISET, {1, 2}} m;

proc print data=example noobs;
run;
```

The dummy parameter i in the *key-set* expression takes precedence over the dummy parameter i declared in the *key-columns* item. The output from these statements is shown in Figure 5.12.

Figure 5.12 CREATE: *key-set* with SETOF Aggregation Expression

i	j	m
1	1	1
1	2	0
4	1	1
4	2	0

If no *key-set* is specified, then the set of index values is formed from the union of the index sets of the implicitly indexed *columns*. The number of index elements for each implicitly indexed array must match the number of *key-columns*. The type of each index element (string versus numeric) must match the element of the same position in other implicit indices.

The arrays for implicitly indexed columns in a CREATE DATA statement do not need to have identical index sets. A missing value is supplied for the value of an implicitly indexed array location when the implied index value is not in the array's index set.

In the following statements, the *key-set* is unspecified. The set of index values is $\{1, 2, 3\}$, which is the union of the index sets of x and y. These index sets are not identical, so missing values are supplied when necessary. The results of these statements are shown in Figure 5.13.

```
proc optmodel;
   number x{1..2} init 2;
   var y{2..3} init 3;
   create data exdata from [keycol] x y;

proc print;
run;
```

Figure 5.13 CREATE: Unspecified *key-set*

Obs	keycol	x	y
1	1	2	.
2	2	2	3
3	3	.	3

The types of the output data set variables match the types of the source values. The output variable type for a *key-columns* value matches the corresponding element type in the index value tuple. A numeric element matches a numeric data set variable, and a string element matches a character variable. For regular *columns*, the source expression type determines the output data set variable type. A numeric expression produces a numeric variable, and a string expression produces a character variable. Character variables are stored in VARCHAR form when the data set engine supports them. Otherwise, character variables are stored in fixed CHAR form.

Lengths of character variables in the output data set are determined automatically when the LENGTH= column option is not specified. The length is set to accommodate the longest string value output in that column.

You can use the iterated *columns* form to output selected rows of multiple arrays, assigning a different data set variable to each column. For example, the following statements output the last two rows of the two-dimensional array, a, along with corresponding elements of the one-dimensional array, b:

```
proc optmodel;
   num m = 3;  /* number of rows/observations */
   num n = 4;  /* number of columns in a */
   num a{i in 1..m, j in 1..n} = i*j;  /* compute a */
   num b{i in 1..m} = i**2;  /* compute b */
   set<num> subset = 2..m;  /* used to omit first row */
   create data out
          from [i]=subset {j in 1..n}<col("a"||j)=a[i,j]> b;
```

The preceding statements create a data set out, which has $m - 1$ observations and $n + 2$ variables. The variables are named i, a1 through an, and b, as shown in Figure 5.14.

Figure 5.14 CREATE DATA Set: The Iterated Column Form

Obs	i	a1	a2	a3	a4	b
1	2	2	4	6	8	4
2	3	3	6	9	12	9

See the section "Data Set Input/Output" on page 119 for more examples of using the CREATE DATA statement.

DO Statement

DO ; *statements* **; END ;**

The DO statement groups a sequence of statements together as a single statement. Each statement within the list is executed sequentially. The DO statement can be used for grouping with the IF, FOR, and COFOR statements.

DO Statement, Iterative

DO *name* **=** *specification-1* [**,** *. . . specification-n*] **;** *statements* **; END ;**

The iterative DO statement assigns the values from the sequence of *specification* items to a previously declared parameter or variable, *name*. The specified statement sequence is executed after each assignment. This statement corresponds to the iterative DO statement of the DATA step.

Each *specification* provides either a single number or a single string value, or a sequence of such values. Each *specification* takes the following form:

expression [**WHILE(** *logic-expression* **)** | **UNTIL(** *logic-expression* **)**]

The *expression* in the *specification* provides a single value or set of values to assign to the target *name*. Multiple values can be provided for the loop by giving multiple *specification* items that are separated by commas. For example, the following statements output the values 1, 3, and 5:

```
proc optmodel;
   number i;
   do i=1,3,5;
      put i;
   end;
```

In this case, the same effect can be achieved with a single range expression in place of the explicit list of values, as in the following statements:

```
proc optmodel;
   number i;
   do i=1 to 5 by 2;
      put 'value of i assigned by the DO loop = ' i;
      i=i**2;
      put 'value of i assigned in the body of the loop = ' i;
   end;
```

The output of these statements is shown in Figure 5.15.

Figure 5.15 DO Loop: Name Parameter Unaffected

```
value of i assigned by the DO loop = 1
value of i assigned in the body of the loop = 1
value of i assigned by the DO loop = 3
value of i assigned in the body of the loop = 9
value of i assigned by the DO loop = 5
value of i assigned in the body of the loop = 25
```

Unlike the DATA step, a range expression requires the limit to be specified. Additionally the BY part, if any, must follow the limit expression. Moreover, although the *name* parameter can be reassigned in the body of the loop, the sequence of values that is assigned by the DO loop is unaffected.

The argument *expression* can also be an expression that returns a set of numbers or strings. For example, the following statements produce the same sequence of values for i as the previous statements but use a set parameter value:

```
proc optmodel;
   set s = {1,3,5};
   number i;
   do i = s;
      put i;
   end;
```

Each *specification* can include a WHILE or UNTIL clause. A WHILE or UNTIL clause applies to the *expression* that immediately precedes the clause. The sequence that is specified by an *expression* can be terminated early by a WHILE or UNTIL clause. A WHILE *logic-expression* is evaluated for each sequence value before the nested *statements*. If the *logic-expression* returns a false (zero or missing) value, then the current sequence is terminated immediately. An UNTIL *logic-expression* is evaluated for each sequence value after the nested *statements*. The sequence from the current *specification* is terminated if the *logic-expression* returns a true value (nonzero and nonmissing). After early termination of a sequence due to a WHILE or UNTIL expression, the DO loop execution continues with the next *specification*, if any.

To demonstrate use of the WHILE clause, the following statements output the values 1, 2, and 3. In this case the sequence of values from the set s is stopped when the value of i reaches 4.

```
proc optmodel;
   set s = {1,2,3,4,5};
   number i;
   do i = s while(i NE 4);
      put i;
   end;
```

DO UNTIL Statement

DO UNTIL (*logic-expression* **) ;** *statements* **; END ;**

The DO UNTIL loop executes the specified sequence of statements repeatedly until the *logic-expression*, evaluated after the *statements*, returns true (a nonmissing nonzero value).

For example, the following statements output the values 1 and 2:

```
proc optmodel;
   number i;
   i = 1;
   do until (i=3);
      put i;
      i=i+1;
   end;
```

Multiple criteria can be introduced using expression operators, as in the following example:

```
do until (i=3 and j=7);
```

For a list of expression operators, see Table 5.10.

DO WHILE Statement

DO WHILE (*logic-expression* **) ;** *statements* **; END ;**

The DO WHILE loop executes the specified sequence of statements repeatedly as long as the *logic-expression*, evaluated before the *statements*, returns true (a nonmissing nonzero value).

For example, the following statements output the values 1 and 2:

```
proc optmodel;
   number i;
   i = 1;
   do while (i<3);
      put i;
      i=i+1;
   end;
```

Multiple criteria can be introduced using expression operators, as in the following example:

```
do while (i<3 and j<7);
```

For a list of expression operators, see Table 5.10.

DROP Statement

DROP *constraint-list* **;**

The DROP statement causes the solver to ignore a list of constraints, constraint arrays, or constraint array locations. The space-delimited *constraint-list* specifies the names of the dropped constraints. Each constraint, constraint array, or constraint array location is named by an *identifier-expression*. An entire constraint array is dropped if an *identifier-expression* omits the index for an array name. Each *identifier-expression* can be prefixed by an indexing set to specify the constraint indices.

The following example statements use the DROP statement:

```
proc optmodel;
   var x{1..10};
   con c1: x[1] + x[2] <= 3;
   con disp{i in 1..9}: x[i+1] >= x[i] + 0.1;

   drop c1;         /* drops the c1 constraint */
   drop disp[5];    /* drops just disp[5] */
   drop {i in 1..3} disp;  /* drop disp[1] disp[2] disp[3] */
   drop disp;       /* drops all disp constraints */
```

The following line drops both the c1 and disp[5] constraints:

```
   drop c1 disp[5];
```

Constraints can be added back to the model by using the RESTORE statement.

EXPAND Statement

EXPAND [*identifier-expression*] [/ *options*] **;**

The EXPAND statement prints the specified constraint, variable, implicit variable, or objective declaration expressions in the current problem after expanding aggregation operators, substituting the current value for parameters and indices, and resolving constant subexpressions. *identifier-expression* is the name of a variable, objective, or constraint. If the name is omitted and no *options* are specified, then all variables, the last objective, used implicit variables, and undropped constraints in the current problem are printed. The following statements show an example EXPAND statement:

```
proc optmodel;
   number n=2;
   var x{1..n};
   min z1=sum{i in 1..n}(x[i]-i)**2;
   max z2=sum{i in 1..n}(i-x[i])**3;
   con c{i in 1..n}: x[i]>=0;
   fix x[2]=3;
   expand;
```

These statements produce the output in Figure 5.16.

Figure 5.16 EXPAND Statement Output

```
Var x[1]
Fix x[2] = 3
Maximize z2=(-x[1] + 1)**3 + (-x[2] + 2)**3
Constraint c[1]: x[1] >= 0
Constraint c[2]: x[2] >= 0
```

Specifying an *identifier-expression* restricts output to the specified declaration. A non-array name prints only the specified item. If an array name is used with a specific index, then information for the specified array location is output. Using an array name without an index restricts output to all locations in the array.

You can use the following *options* to further control the EXPAND statement output:

SOLVE

causes the EXPAND statement to print the variables, objectives, and constraints in the same form that would be seen by the solver if a SOLVE statement were executed. This includes any transformations by the PROC OPTMODEL presolver (see the section "Presolver" on page 146). In this form any fixed variables are replaced by their values. Unless an *identifier-expression* specifies a particular non-array item or array location, the EXPAND output is restricted to only the variables, the constraints, and the current problem objective.

The following options restrict the types of declarations output when no specific non-array item or array location is requested. By default, variables, implicit variables, the current objective, and constraints are output. Only the requested declaration types are output when one or more of the following options are used.

ALL

requests the output of all declaration types. This includes the complete list of problem objectives, not just the current objective.

ALLOBJ

requests the output of all objectives that are used in the current problem. This includes the complete list of problem objectives and any objectives that are referenced as implicit variables.

CONSTRAINT | CON

requests the output of undropped constraints.

FIX

requests the output of fixed variables. These variables might have been fixed by the FIX

statement (or by the presolver if the SOLVE option is specified). The FIX option can also be used in combination with the name of a variable array to display just the fixed elements of the array.

IIS

restricts the display to items found in the irreducible infeasible set (IIS) after the most recent SOLVE performed by the LP solver with the IIS=TRUE option. The IIS option for the EXPAND statement can also be used in combination with the name of a variable or constraint array to display only the elements of the array in the IIS. For more information about IIS, see the section "Irreducible Infeasible Set" on page 275.

IMPVAR

requests the output of nonconstant implicit variables referenced in the current problem.

OBJECTIVE | OBJ

requests the output of objectives used in the current problem. This includes the current problem objective and any objectives referenced as implicit variables.

OMITTED

requests the output of variables that are referenced by problem equations but were not included in the current USE PROBLEM instance. The OPTMODEL procedure omits these variables from the generated problem.

VAR

requests the output of unfixed variables. The VAR option can also be used in combination with the name of a variable array to display just the unfixed elements of the array.

For example, you can see the effect of a FIX statement on the problem that is presented to the solver by using the SOLVE option. You can modify the previous example as follows:

```
proc optmodel;
   number n=2;
   var x{1..n};
   min z1=sum{i in 1..n}(x[i]-i)**2;
   max z2=sum{i in 1..n}(i-x[i])**3;
   con c{i in 1..n}: x[i]>=0;
   fix x[2]=3;
   expand / solve;
```

These statements produce the output in Figure 5.17.

Figure 5.17 Expansion with Fixed Variable

```
Var x[1] >= 0
Fix x[2] = 3
Maximize z2=(-x[1] + 1)**3 - 1
```

Compare the results in Figure 5.17 to those in Figure 5.16. The constraint c[1] has been converted to a variable bound. The subexpression that uses the fixed variable has been resolved to a constant.

FILE Statement

FILE *file-specification* [**LRECL=***value*] **;**

The FILE statement selects the current output file for the PUT statement. By default PUT output is sent to the

SAS log. Use the FILE statement to manage a group of output files. The specified file is opened for output if it is not already open. The output file remains open until it is closed with the CLOSEFILE statement.

file-specification names the output file. It can use any of the following forms:

'*external-file*'
: specifies the physical name of an external file in quotation marks. The interpretation of the file name depends on the operating environment.

file-name
: specifies the logical name associated with a file by the FILENAME statement or by the operating environment. The names PRINT and LOG are reserved to refer to the SAS listing and log files, respectively.

  NOTE: Details about the FILENAME statement can be found in *SAS Global Statements: Reference*.

(*expression* **)**
: specifies an expression that evaluates to a string that contains the physical name of an external file.

The LRECL= option sets the line length of the output file. The LRECL= option is ignored if the file is already open or if the PRINT or LOG file is specified.

The LRECL= *value* can be specified in these forms:

integer
: specifies the desired line length.

identifier-expression
: specifies the name of a numeric parameter that contains the length.

(*expression* **)**
: specifies a numeric expression in parentheses that returns the line length.

The LRECL= *value* cannot exceed the largest four-byte signed integer, which is $2^{31} - 1$.

The following example shows how to use the FILE statement to handle multiple files:

```
proc optmodel;
   file 'file.txt' lrecl=80;   /* opens file.txt     */
   put 'This is line 1 of file.txt.';
   file print;                  /* selects the listing */
   put 'This goes to the listing.';
   file 'file.txt';            /* reselects file.txt */
   put 'This is line 2 of file.txt.';
   closefile 'file.txt';       /* closes file.txt    */
   file log;                    /* selects the SAS log */
   put 'This goes to the log.';

   /* using expression to open and write a collection of files */
   str ofile;
   num i;
   num l = 40;
   do i = 1 to 3;
```

```
      ofile = ('file' || i || '.txt');
      file (ofile) lrecl=(1*i);
      put ('This goes to ' || ofile);
      closefile (ofile);
   end;
```

The following statements illustrate the usefulness of using a logical name associated with a file by FILENAME statement:

```
proc optmodel;
   /* assigns a logical name to file.txt */
   /* see FILENAME statement in        */
   /* SAS Global Statements: Reference */
   filename myfile 'file.txt' mod;

   file myfile;
   put 'This is line 3 of file.txt.';
   closefile myfile;
   file myfile;
   put 'This is line 4 of file.txt.';
   closefile myfile;
```

Notice that the FILENAME statement opens the file referenced for append. Therefore, new data are appended to the end every time the logical name, myfile, is used in the FILE statement.

FIX Statement

FIX *variable-list* [**=** *expression*] **;**

The FIX statement causes the solver to treat a list of variables, variable arrays, or variable array locations as fixed in value. The *variable-list* consists of one or more variable names separated by spaces. Each member of the *variable-list* is fixed to the same *expression*. For example, the following statements fix the variables x and y to 3:

```
proc optmodel;
   var x, y;
   num a = 2;
   fix x y=a+1;
```

A variable is specified with an *identifier-expression* (see the section "Identifier Expressions" on page 101). An entire variable array is fixed if the *identifier-expression* names an array without providing an index. Each *identifier-expression* can be prefixed by an indexing set to specify the variable indices.

You can specify a new value for the variables by using the *expression*. If a single *identifier-expression* is specified, prefixed by an indexing set, then the *expression* is evaluated for each element in the indexing set. Otherwise the *expression* is evaluated exactly once. For example, the following statements fix all locations in array x to 0 except x[10], which is fixed to 1:

```
proc optmodel;
   var x{1..10};
   fix x = 0;
   fix x[10] = 1;
```

The same effect can be achieved using a single statement:

```
fix {i in 1..10} x[i] = if i eq 10 then 1 else 0;
```

If *expression* is omitted, the variable is fixed at its current value. For example, you can fix some variables to be their optimal values after the SOLVE statement is invoked. **NOTE:** The fixed value is equal to the current value for a fixed variable. The fixed value is updated if a new value is assigned to a fixed variable.

The effect of FIX can be reversed by using the UNFIX statement.

FOR Statement

FOR { *index-set* **}** *statement* **;**

The FOR statement executes its substatement for each member of the specified *index-set*. The index set can declare local dummy parameters. You can reference the value of these parameters in the substatement. For example, consider the following statements:

```
proc optmodel;
   for {i in 1..2, j in {'a', 'b'}} put i= j=;
```

These statements produce the output in Figure 5.18.

Figure 5.18 FOR Statement Output

```
i=1 j=a
i=1 j=b
i=2 j=a
i=2 j=b
```

As another example, the following statements set the current values for variable x to random values between 0 and 1:

```
proc optmodel;
   var x{1..10};
   for {i in 1..10}
      x[i] = ranuni(-1);
```

Multiple statements can be controlled by specifying a DO statement group for the substatement.

CAUTION: Avoid modifying the parameters that are used by the FOR or COFOR statement index set from within the substatement. The set value that is used for the left-most index set item is not affected by such changes. However, the effect of parameter changes on later index set items cannot be predicted.

IF Statement

IF *logic-expression* **THEN** *statement* [**ELSE** *statement*] **;**

The IF statement evaluates the logical expression and then conditionally executes the THEN or ELSE substatements. The substatement that follows the THEN keyword is executed when the logical expression result is nonmissing and nonzero. The ELSE substatement, if any, is executed when the logical expression result is a missing value or zero. The ELSE part is optional and must immediately follow the THEN substatement. When IF statements are nested, an ELSE is always matched to the nearest incomplete unmatched IF-THEN. Multiple statements can be controlled by using DO statements with the THEN or ELSE substatements.

NOTE: When an IF-THEN statement is used **without** an ELSE substatement, substatements of the IF statement are executed when possible as they are entered. Under certain circumstances, such as when an IF statement is nested in a FOR loop, the statement is not executed during interactive input until the next statement is seen. By following the IF-THEN statement with an extra semicolon, you can cause it to be executed upon submission, since the extra semicolon is handled as a null statement.

LEAVE Statement

LEAVE ;

The LEAVE statement terminates the execution of the entire loop body (iterative DO, DO UNTIL, DO WHILE, FOR, or COFOR) that immediately contains the LEAVE statement. Execution resumes at the statement that follows the loop. The following example demonstrates a simple use of the LEAVE statement:

```
proc optmodel;
   number i, j;
   do i = 1..5;
      do j = 1..4;
         if i >= 3 and j = 2 then leave;
      end;
      print i j;
   end;
```

The results from these statements are displayed in Figure 5.19.

Figure 5.19 LEAVE Statement Output

i	j
1	4

i	j
2	4

i	j
3	2

i	j
4	2

i	j
5	2

For values of i equal to 1 or 2, the inner loop continues uninterrupted, leaving j with a value of 4. For values of i equal to 3, 4, or 5, the inner loop terminates early, leaving j with a value of 2.

Null Statement

;

The null statement is treated as a statement in the PROC OPTMODEL syntax, but its execution has no effect. It can be used as a placeholder statement.

PRINT Statement

PRINT *print-items* **;**

The PRINT statement outputs string and numeric data in tabular form. The statement specifies a list of arrays or other data items to print. Multiple items can be output together as data columns in the same table.

If no format is specified, the PRINT statement handles the details of formatting automatically (see the section "Formatted Output" on page 123 for details). The default format for a numerical column is the fixed-point format (*w.d* format), which is chosen based on the values of the PDIGITS= and PWIDTH= options (see the section "PROC OPTMODEL Statement" on page 38) and on the values in the column. The PRINT statement uses scientific notation (the E*w.* format) when a value is too large or too small to display in fixed format. The default format for a character column is the $*w.* format, where the width is set to be the length of the longest string (ignoring trailing blanks) in the column.

print-item can be specified in the following forms:

identifier-expression [*format*]
: specifies a data item to output. *identifier-expression* can name an array. In that case all defined array locations are output. *format* specifies a SAS format that overrides the default format.

(*expression* **)** [*format*]
: specifies a data value to output. *format* specifies a SAS format that overrides the default format.

{ *index-set* **}** *identifier-expression* [*format*]
: specifies a data item to output under the control of an *index set*. The item is printed as if it were an array with the specified set of indices. This form can be used to print a subset of the locations in an array, such as a single column. If the *identifier-expression* names an array, then the indices of the array must match the indices of the *index-set*. The *format* argument specifies a SAS format that overrides the default format.

{ *index-set* **}** **(** *expression* **)** [*format*]
: specifies a data item to output under the control of an *index set*. The item is printed as if it were an array with the specified set of indices. In this form the *expression* is evaluated for each member of the *index-set* to create the array values for output. *format* specifies a SAS format that overrides the default format.

string
: specifies a string value to print.

PAGE
: specifies a page break.

The following example demonstrates the use of several *print-item* forms:

```
proc optmodel;
   num x = 4.3;
   var y{j in 1..4} init j*3.68;
   print y; /* identifier-expression */
   print (x * .265) dollar6.2; /* (expression) [format] */
   print {i in 2..4} y; /* {index-set} identifier-expression */
   print {i in 1..3}(i + i*.2345692) best7.;
                              /* {index-set} (expression) [format] */
   print "Line 1"; /* string */
```

The output is displayed in Figure 5.20.

Figure 5.20 *Print-item* Forms

[1]	y
1	3.68
2	7.36
3	11.04
4	14.72

$1.14

[1]	y
2	7.36
3	11.04
4	14.72

[1]	
1	1.23457
2	2.46914
3	3.70371

Line 1

Adjacent print items that have similar indexing are grouped together and output in the same table. Items have similar indexing if they specify arrays that have the same number of indices and have matching index types (numeric versus string). Nonarray items are considered to have the same indexing as other nonarray items. The resulting table has a column for each array index followed by a column for each print item value. This format is called *list form*. For example, the following statements produce a list form table:

```
proc optmodel;
   num a{i in 1..3} = i*i;
   num b{i in 3..5} = 4*i;
   print a b;
```

These statements produce the listing output in Figure 5.21.

Figure 5.21 List Form PRINT Table

[1]	a	b
1	1	
2	4	
3	9	12
4		16
5		20

The array index columns show the set of valid index values for the print items in the table. The array index column for the ith index is labeled [i]. There is a row for each combination of index values that was used. The index values are displayed in sorted ascending order.

The data columns show the array values that correspond to the index values in each row. If a particular array index is invalid or the array location is undefined, then the corresponding table entry is displayed as blank for numeric arrays and as an empty string for string arrays. If the print items are scalar, then the table has a single row and no array index columns.

If a table contains a single array print item, the array is two-dimensional (has two indices), and the array is dense enough, then the array is shown in *matrix form*. In this format there is a single index column that contains the row index values. The label of this column is blank. This column is followed by a column for every unique column index value for the array. The latter columns are labeled by the column value. These columns contain the array values for that particular array column. Table entries that correspond to array locations that have invalid or undefined combinations of row and column indices are blank or (for strings) printed as an empty string.

The following statements generate a simple example of matrix output:

```
proc optmodel;
   print {i in 1..6, j in i..6} (i*10+j);
```

The PRINT statement produces the output in Figure 5.22.

Figure 5.22 Matrix Form PRINT Table

	1	2	3	4	5	6
1	11	12	13	14	15	16
2		22	23	24	25	26
3			33	34	35	36
4				44	45	46
5					55	56
6						66

The PRINT statement prints single two-dimensional arrays in the form that uses fewer table cells (headings are ignored). Sparse arrays are normally printed in list form, and dense arrays are normally printed in matrix form. In a PROC OPTMODEL statement, the PMATRIX= option enables you to tune how the PRINT statement displays a two-dimensional array. The value of this option scales the total number of nonempty array elements, which is used to compute the tables cells needed for list form display. Specifying values for the PMATRIX= option less than 1 causes the list form to be used in more cases, while specifying values greater than 1 causes the matrix form to be used in more cases. If the value is 0, then the list form is always used. The default value of the PMATRIX= option is 1. Changing the default can be done with the RESET OPTIONS statement.

The following statements illustrate how the PMATRIX= option affects the display of the PRINT statement:

```
proc optmodel;
   num a{i in 1..6, i..i} = i;
   num b{i in 1..3, j in 1..3} = i*j;
   print a;
   print b;
   reset options pmatrix=3;
   print a;
   reset options pmatrix=0.5;
   print b;
```

The output is shown in Figure 5.23.

Figure 5.23 PRINT Statement: Effects of PMATRIX= Option

[1]	[2]	a
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6

	b		
	1	2	3
1	1	2	3
2	2	4	6
3	3	6	9

	a					
	1	2	3	4	5	6
1	1					
2		2				
3			3			
4				4		
5					5	
6						6

[1]	[2]	b
1	1	1
1	2	2
1	3	3
2	1	2
2	2	4
2	3	6
3	1	3
3	2	6
3	3	9

From Figure 5.23, you can see that, by default, the PRINT statement tries to make the display compact. However, you can change the default by using the PMATRIX= option.

PROFILE Statement

PROFILE [*mode*] *options* **;**

The PROFILE statement controls the PROC OPTMODEL profiler, which enables you to collect and display timing and execution count information for PROC OPTMODEL processing. The profiler can be very useful for finding bottlenecks during execution, such as constraints that require large amounts of time during problem generation. When the profiler is enabled, PROC OPTMODEL records the time for processing a declaration, the time for executing statements, and the number of times that statements are executed. See "Example 5.7: Sparse Modeling" on page 184 for an example use of the PROFILE statement.

The *mode* argument specifies the action that the PROFILE statement performs. You can use the following values for *mode*:

ON

enables the profiler. Data are collected until the profiler is disabled or PROC OPTMODEL terminates. The profiler is also enabled when no *mode* is specified in a PROFILE statement.

OFF

disables the profiler. Note that the profiler is disabled when PROC OPTMODEL begins execution.

PRINT

prints the current accumulated profiler data. Items for declarations and statements are displayed in a table in descending order of their net time. Accumulated data are printed automatically when PROC OPTMODEL terminates.

The *options* control how PROC OPTMODEL collects and displays profiler information. You can specify the following options:

MEMORY | NOMEMORY

specifies whether to add the memory requirements for declarations to the output for the PROFILE PRINT action. The MemSize column of the ProfileInfo ODS table contains the approximate memory requirements in bytes when the table is displayed. The memory that is used by shared values, such as strings and sets, is divided among the declarations that reference the values. NOMEMORY is the default.

PERCENT=*number*

restricts the output for PROFILE PRINT to items whose net times account for at least the specified percentage of total profiled time, $\text{total time} \cdot \text{number}/100$. Items that have smaller times are aggregated into a single item at the end of the table. You can set this option before the display of profile data, and it does not affect the data collection. The value of *number* can range from 0 to 100. The default value is 1.

RESET

discards accumulated profiler data when the PROFILE statement completes execution. Accumulated data are retained until they are explicitly reset.

STMTDEPTH=*number* **| ALL**

allows collection of profiler data for nested statements. With the default option, STMTDEPTH=1, profiler data are collected only for top-level statements. The elapsed time for nested statement timing is included in the top-level statement timing. For example, time for a top-level FOR statement would include the execution of its substatement. Use the STMTDEPTH= option to profile the nested statements individually. The value *number* specifies the maximum nesting depth at which to profile statements individually. The nesting depth of a top-level statement is 1. Otherwise the nesting depth of a statement is one more than the nesting depth of the statement that encloses it, such as a DO, IF, or FOR statement.

For a PROFILE statement within a DO block or DO loop, the statement depth value is interpreted relative to the enclosing DO statement. For example, specifying STMTDEPTH=1 within a DO block

causes the top-level statements of the DO block to be profiled. The STMTDEPTH= option is reset to its previous value when the enclosing DO statement completes execution.

The value of *number* can be an integer between 1 and 32,767. Using the ALL keyword is equivalent to specifying 32,767. Note that profiler timing can add significant overhead. Use a small *number* to minimize overhead.

The elapsed time that is required to process an item includes the processing of other profiled items that it depends on. For example, the processing of a constraint during problem generation might require the evaluation of parameter values. The PROFILE statement reports net time so that the total of profiled times represents actual processing time.

The equation that is used to compute net time is

$$\text{net time} = \text{elapsed time} - \text{nested time} - \text{wait time}$$

Elapsed time is the elapsed clock time this is required for processing an item. Nested time is the total of the elapsed times that are spent within the same thread to process other profiled items, such as substatements or declaration values. Wait time represents the time that a single thread is allocated but idle because it is waiting for other threads to perform the required processing. The total of net times can exceed the elapsed wall clock time when multiple threads are used.

PUT Statement

PUT [*put-items*] [@ | @@] **;**

The PUT statement writes text data to the current output file. The syntax of the PUT statement in PROC OPTMODEL is similar to the syntax of the PROC IML and DATA step PUT statements. The PUT statement contains a list of items that specify data for output and provide instructions for formatting the data.

The current output file is initially the SAS log. This can be overridden with the FILE statement. An output file can be closed with the CLOSEFILE statement.

Normally the PUT statement outputs the current line after processing all items. Final @ or @@ operators suppress this automatic line output and cause the current column position to be retained for use in the next PUT statement.

put-item can take any of the following forms.

identifier-expression [**=**] [*format*]
: outputs the value of the parameter or variable that is specified by the *identifier-expression*. The equal sign (=) causes a name for the location to be printed before each location value. If the *identifier-expression* uses a suffix that can be followed by a solution index, you can use an asterisk (*) instead of the index *expression* to display all saved solution values.

  Normally each item value is printed in a default format. Any leading and trailing blanks in the formatted value are removed, and the value is followed by a blank space. When an explicit format is specified, the value is printed within the width determined by the format.

*name***[*]** [**.***suffix* [**[** *expression* | ***** **]**]] [**=**] [*format*]
: outputs each defined location value for an array parameter. The array name is specified as in the *identifier-expression* form except that the index list is replaced by an asterisk (*). The equal sign (=) causes a name for the location to be printed before each location value

along with the actual index values to be substituted for the asterisk. If the *suffix* can be followed by a solution index, you can use an asterisk (*) instead of the index *expression* to display all saved solution values.

Each item value normally prints in a default format. Any leading and trailing blanks in the formatted value are removed, and the value is followed by a blank space. When an explicit format is specified, the value is printed within the width determined by the format.

(*expression* **)** [**=**] [*format*]
outputs the value of the expression enclosed in parentheses. This produces similar results to the *identifier-expression* form except that the equal sign (=) uses the expression to form the name.

'*quoted-string*'
copies the string to the output file.

@*integer* | *identifier-expression* | **(** *expression* **)** sets the absolute column position within the current line. The literal or expression value determines the new column position.

+*integer* |*identifier-expression*|**(** *expression* **)** sets the relative column position within the current line. The literal or expression value determines the amount to update the column position.

/
outputs the current line and moves to the first column of the next line.

PAGE
outputs any pending line data and moves to the top of the next page.

QUIT Statement

QUIT ;

The QUIT statement terminates the OPTMODEL execution. The statement is executed immediately, so it cannot be a nested statement. A QUIT statement is implied when a DATA or PROC statement is read.

READ DATA Statement

READ DATA *SAS-data-set* [**NOMISS**] **INTO** [[*set-name* **=**] **[** *read-key-columns* **]**] [*read-columns*] **;**

The READ DATA statement reads data from a SAS data set into PROC OPTMODEL parameter and variable locations. The arguments to the READ DATA statement are as follows:

SAS-data-set
specifies the input data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses.

set-name
specifies a set parameter in which to save the set of observation key values read from the input data set.

read-key-columns
provide the index values for array destinations.

read-columns
specify the data values to read and the destination locations.

The following example uses the READ DATA statement to copy data set variables j and k from the SAS data set indata into parameters of the same name. The READ= data set option specifies a password.

```
proc optmodel;
   number j, k;
   read data indata(read=secret) into j k;
```

Key Columns

If any *read-key-columns* are specified, then the READ DATA statement reads all observations from the input data set. If no *read-key-columns* are specified, then only the first observation of the data set is read. The data set is closed after reading the requested information.

Each *read-key-column* specifies a data set variable that supplies the column value. The values of the specified data set variables from each observation are combined into a key tuple. This combination is known as the *observation key*. The observation key is used to index array locations specified by the *read-columns* items. The observation key is expected to be unique for each observation read from the data set.

The syntax for a *read-key-column* is as follows:

[*name* **=**] *source-name* [/ *trim-option*]

Each *read-key-column* specifies an element of the observation key tuple. The *source-name* specifies the data set variable name, either as a *name* or by using a COL expression (as described later for *read-columns*). Use the special data set variable name _N_ to refer to the observation's iteration number, starting at 1.

A local dummy parameter can be created for each *read-key-column*. You can use the dummy parameter to reference the key tuple element value in subsequent *read-columns* items. A dummy parameter is created when the *read-key-column* specifies a *name* preceding the equal sign (=). The dummy parameter has the specified name. Also, a dummy parameter is created when the *source-name* is a nonliteral *name* and no alternate name has been specified using the equal sign. In this case, the dummy parameter has the same name as the data set variable.

You can specify a *set-name* to save the set of observation keys into a set parameter. If the observation key consists of a single scalar value, then the set member type must match the scalar type. Otherwise the set member type must be a tuple with element types that match the corresponding observation key element types.

The READ DATA statement initially assigns an empty set to the target *set-name* parameter. As observations are read, a tuple for each observation key is added to the set. A set used to index an array destination in the *read-columns* can be read at the same time as the array values. Consider a data set, invdata, created by the following statements:

```
data invdata;
   input item $ invcount;
   datalines;
table 100
sofa  250
chair 80
;
```

The following statements read the data set invdata, which has two variables, item and invcount. The READ DATA statement constructs a set of inventory items, Items. At the same time, the parameter location invcount[item] is assigned the value of the data set variable invcount in the corresponding observation.

```
proc optmodel;
   set<string> Items;
   number invcount{Items};
   read data invdata into Items=[item] invcount;
   print invcount;
```

The output of these statements is shown in Figure 5.24.

Figure 5.24 READ DATA Statement: Key Column

[1]	invcount
chair	80
sofa	250
table	100

When observations are read, the values of data set variables are copied to parameter locations. Numeric values are copied unchanged. For character values, *trim-option* controls how leading and trailing blanks are processed. *trim-option* is ignored when the value type is numeric. Specify any of the following keywords for *trim-option*:

TRIM | **TR**
: removes leading and trailing blanks from the data set value. This is the default behavior.

LTRIM | **LT**
: removes only leading blanks from the data set value.

RTRIM | **RT**
: removes only trailing blanks from the data set value.

NOTRIM | **NT**
: copies the data set value with no changes.

Columns

read-columns specify data set variables to read and PROC OPTMODEL parameter locations to which to assign the values. The types of the input data set variables must match the types of the parameters. Array parameters can be implicitly or explicitly indexed by the observation key values.

Normally, missing values from the data set are assigned to the parameters that are specified in the *read-columns*. The NOMISS keyword suppresses the assignment of missing values, leaving the corresponding parameter locations unchanged. Note that the parameter location does not need to have a valid index in this case. This permits a single statement to read data into multiple arrays that have different index sets.

read-columns have the following forms:

identifier-expression [**=** *name* | **COL(** *name-expression* **)**] [/ *trim-option*]
: transfers an input data set variable to a target parameter or variable. *identifier-expression* specifies the target. If the *identifier-expression* specifies an array without an explicit index, then the observation key provides an implicit index. The name of the input data set variable can be specified with a *name* or a COL expression. Otherwise the data set variable name is given by the *name* part of the *identifier-expression*. For COL expressions, the string-valued *name-expression* is evaluated to determine the data set variable name.

trim-option controls removal of leading and trailing blanks in the incoming data. For example, the following statements read the data set variables column1 and column2 from the data set exdata into the PROC OPTMODEL parameters p and q, respectively. The observation numbers in exdata are read into the set indx, which indexes p and q.

```
data exdata;
   input column1 column2;
   datalines;
1 2
3 4
;


proc optmodel;
   number n init 2;
   set<num> indx;
   number p{indx}, q{indx};
   read data exdata into
      indx=[_N_] p=column1 q=col("column"||n);
   print p q;
```

The output is shown in Figure 5.25.

Figure 5.25 READ DATA Statement: Identifier Expressions

[1]	p	q
1	1	2
2	3	4

{ *index-set* **}** **<** *read-columns* **>**

performs the transfers by iterating each column specified by <*read-columns*> for each member of the *index-set*. If there are n columns and m index set members, then $n \times m$ columns are generated. The dummy parameters from the index set can be used in the columns to generate distinct input data set variable names in the iterated columns, using COL expressions. The columns are expanded when the READ DATA statement is executed, before any observations are read. This form of *read-columns* cannot be nested. In other words, the following form of *read-columns* is NOT allowed:

{ *index-set* **}** **<** **{** *index-set* **}** **<** *read-columns* **>** **>**

An example that demonstrates the use of the iterated column *read-option* follows.

You can use an iterated column *read-option* to read multiple data set variables into the same array. For example, a data set might store an entire row of array data in a group of data set variables. The following statements demonstrate how to read a data set that contains demand data divided by day:

```
data dmnd;
   input loc $ day1 day2 day3 day4 day5;
   datalines;
East 1.1 2.3 1.3 3.6 4.7
```

```
West 7.0 2.1 6.1 5.8 3.2
;

proc optmodel;
   set DOW = 1..5;  /* days of week, 1=Monday, 5=Friday */
   set<string> LOCS; /* locations */
   number demand{LOCS, DOW};
   read data dmnd
        into LOCS=[loc]
        {d in DOW} < demand[loc, d]=col("day"||d) >;
   print demand;
```

These statements read a set of demand variables named DAY1–DAY5 from each observation, filling in the two-dimensional array demand. The output is shown in Figure 5.26.

Figure 5.26 Demand Data

	demand				
	1	2	3	4	5
East	1.1	2.3	1.3	3.6	4.7
West	7.0	2.1	6.1	5.8	3.2

RESET OPTIONS Statement

RESET OPTIONS *options* **;**

RESET OPTION *options* **;**

The RESET OPTIONS statement sets PROC OPTMODEL option values or restores them to their defaults. Options can be specified by using the same syntax as in the PROC OPTMODEL statement. The RESET OPTIONS statement provides two extensions to the option syntax. If an option normally requires a value (specified with an equal sign (=) operator), then specifying the option name alone resets it to its default value. You can also specify an expression enclosed in parentheses in place of a literal value. See the section "OPTMODEL Options" on page 154 for an example.

The RESET OPTIONS statement can be placed inside loops or conditional statements. The statement is applied each time it is executed.

RESTORE Statement

RESTORE *constraint-list* **;**

The RESTORE statement adds a list of constraints, constraint arrays, or constraint array locations that were dropped by the DROP statement back into the solver model, or includes constraints in a problem where they were not previously present. The space-delimited *constraint-list* specifies the names of the constraints. Each constraint, constraint array, or constraint array location is named by an *identifier-expression*. An entire constraint array is restored if an *identifier-expression* omits the index from an array name. Each *identifier-expression* can be prefixed by an indexing set to specify the constraint indices.

For example, the following statements declare a constraint array and then drop it:

```
con c{i in 1..4}: x[i] + y[i] <=1;
drop c;
```

The following statement restores the first constraint:

```
restore c[1];
```

The following statement restores the second and third constraints:

```
restore {i in 2..3} c[i]; /* like "restore c[2] c[3];" */
```

If you want to restore all of the constraints, you can submit the following statement:

```
restore c;
```

SAVE MPS Statement

SAVE MPS *SAS-data-set* [**(OBJECTIVE** | **OBJ)** *name*] [**(NOOBJECTIVE** | **NOOBJ)]** **;**

The SAVE MPS statement saves the structure and coefficients for a linear programming model into a SAS data set. This data set can be used as input data for the OPTLP or OPTMILP procedure.

NOTE: The OPTMODEL presolver (see the section "Presolver" on page 146) is automatically bypassed so that the statement saves the original model *without* eliminating fixed variables, tightening bounds, and so on.

The *SAS-data-set* argument specifies the output data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses. The output data set uses the MPS format described in Chapter 18, "The MPS-Format SAS Data Set." An _ID_ column is automatically added to the data set when it is created using a CAS engine libref. The generated data set contains observations that define different parts of the linear program.

Variables, constraints, and objectives are referenced in the data set by using label text from the corresponding .label suffix value. The default text is based on the name in the model. See the section "Suffixes" on page 135 for more details. Labels are limited by default to 32 characters and are abbreviated to fit. You can change the maximum length for labels by using the MAXLABLEN= option. When needed, a programmatically generated number is added to labels to avoid duplication.

If the OBJECTIVE keyword is used, the objective *name* becomes the current problem objective. If the NOOBJECTIVE keyword is used or the current problem does not have an objective, then the data set includes a default constant zero objective. Otherwise, the current problem objective is included in the data set.

When an integer variable has been assigned a nondefault branching priority or direction, the MPS data set includes a BRANCH section. See Chapter 18, "The MPS-Format SAS Data Set," for more details.

The following statements show an example of the SAVE MPS statement. The model is specified using the OPTMODEL procedure. Then it is saved as the MPS data set MPSData, as shown in Figure 5.27. Next, PROC OPTLP is used to solve the resulting linear program.

```
proc optmodel;
   var x >= 0, y >= 0;
   con c: x >= y;
   con bx: x <= 2;
   con by: y <= 1;
   min obj=0.5*x-y;
   save mps MPSData;
quit;
```

```
proc optlp data=MPSData pout=PrimalOut dout=DualOut;
run;
```

Figure 5.27 The MPS Data Set Generated by SAVE MPS Statement

Obs	FIELD1	FIELD2	FIELD3	FIELD4	FIELD5	FIELD6
1	NAME		MPSData	.		.
2	ROWS			.		.
3	N	obj		.		.
4	G	c		.		.
5	L	bx		.		.
6	L	by		.		.
7	COLUMNS			.		.
8		x	obj	0.5	c	1
9		x	bx	1.0		.
10		y	obj	-1.0	c	-1
11		y	by	1.0		.
12	RHS			.		.
13		.RHS.	bx	2.0		.
14		.RHS.	by	1.0		.
15	ENDATA			.		.

SAVE QPS Statement

SAVE QPS *SAS-data-set* [**(OBJECTIVE** | **OBJ)** *name*] [**(NOOBJECTIVE** | **NOOBJ)**] **;**

The SAVE QPS statement saves the structure and coefficients for a quadratic programming model into a SAS data set. This data set can be used as input data for the OPTQP procedure.

NOTE: The OPTMODEL presolver (see the section "Presolver" on page 146) is automatically bypassed so that the statement saves the original model *without* eliminating fixed variables, tightening bounds, and so on.

The *SAS-data-set* argument specifies the output data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses. The output data set uses the QPS format described in Chapter 18. An _ID_ column is automatically added to the data set when it is created using a CAS engine libref. The generated data set contains observations that define different parts of the quadratic program.

Variables, constraints, and objectives are referenced in the data set by using label text from the corresponding .label suffix value. The default text is based on the name in the model. See the section "Suffixes" on page 135 for more details. Labels are limited by default to 32 characters and are abbreviated to fit. You can change the maximum length for labels by using the MAXLABLEN= option. When needed, a programmatically generated number is added to labels to avoid duplication.

If the OBJECTIVE keyword is used, the objective *name* becomes the current problem objective. If the NOOBJECTIVE keyword is used or the current problem does not have an objective, then the data set includes a default constant zero objective. Otherwise, the current problem objective is included in the data set. The quadratic coefficients of the objective function appear in the QSECTION section of the output data set.

The following statements show an example of the SAVE QPS statement. The model is specified using the OPTMODEL procedure. Then it is saved as the QPS data set QPSData, as shown in Figure 5.28. Next, PROC OPTQP is used to solve the resulting quadratic program.

```
proc optmodel;
   var x{1..2} >= 0;
   min z = 2*x[1] + 3 * x[2] + x[1]**2 + 10*x[2]**2
                  + 2.5*x[1]*x[2];
   con c1: x[1] - x[2] <= 1;
   con c2: x[1] + 2*x[2] >= 100;
   save qps QPSData;
quit;

proc optqp data=QPSData pout=PrimalOut dout=DualOut;
run;
```

Figure 5.28 QPS Data Set Generated by the SAVE QPS Statement

Obs	FIELD1	FIELD2	FIELD3	FIELD4	FIELD5	FIELD6
1	NAME		QPSData	.		.
2	ROWS			.		.
3	N	z		.		.
4	L	c1		.		.
5	G	c2		.		.
6	COLUMNS			.		.
7		x[1]	z	2.0	c1	1
8		x[1]	c2	1.0		.
9		x[2]	z	3.0	c1	-1
10		x[2]	c2	2.0		.
11	RHS			.		.
12		.RHS.	c1	1.0		.
13		.RHS.	c2	100.0		.
14	QSECTION			.		.
15		x[1]	x[1]	2.0		.
16		x[1]	x[2]	2.5		.
17		x[2]	x[2]	20.0		.
18	ENDATA			.		.

SOLVE Statement

SOLVE [**WITH** *solver*] [**(OBJECTIVE** | **OBJ)** *name*] [**(NOOBJECTIVE** | **NOOBJ)**] [**RELAXINT**] [/ *options*] **;**

The SOLVE statement invokes a PROC OPTMODEL solver. The current model is first resolved to the numeric form that is required by the solver. The resolved model and possibly the current values of any optimization variables are passed to the solver. After the solver finishes executing, the SOLVE statement prints a short table that shows a summary of results from the solver (see the section "ODS Table and Variable Names" on page 125) and updates the _OROPTMODEL_ macro variable.

Here are the arguments to the SOLVE statement:

solver

selects the named solver: CLP, LP, LSO, MILP, NETWORK, NLP, or QP (see corresponding chapters in this book for details). If you do not specify a WITH clause, PROC

OPTMODEL chooses a solver that depends on the problem type. Table 5.8 lists the default solver for each problem type.[1]

Table 5.8 Default Solvers and Algorithms in PROC OPTMODEL

Problem	Solver	Algorithm
Constraint programming	**CLP**	Constraint propagation and backtracking search
Linear programming	**LP**	Dual simplex
Mixed integer linear programming	**MILP**	Branch-and-cut
General nonlinear programming	**NLP**	Interior point NLP
Quadratic programming	**QP**	Interior point QP

name

specifies the objective to use. This resets the objective list for the problem. You can abbreviate the OBJECTIVE keyword as OBJ. If you do not specify this argument, then the problem objectives are unchanged. You can also specify multiple objectives for the LSO solver by using a list in parentheses of *names*, separated by spaces, in place of *name*. Within this list, you can prefix each *name* by using an indexing set to specify a group of objectives.

NOOBJECTIVE requests that the solver ignore the objectives for the problem and use a constant zero objective instead. This keyword enables the solver to process the current model as a feasibility problem. You can abbreviate the NOOBJECTIVE keyword as NOOBJ.

RELAXINT requests that any integral variables be relaxed to be continuous. RELAXINT can be used with linear and nonlinear problems in addition to any solver.

options specifies solver options. You can specify solver options directly only when you use the WITH clause. A list of the options available with the solver is provided in the individual chapters that describe each solver. Table 5.9 lists the available option types. You can use an expression in parentheses in place of a literal option value for numeric and keyword options. A string expression is matched to a keyword. OPTMODEL parameters that are changed by the solver must be specified by a parameter or array option.

Table 5.9 Solver Option Types

Type	Syntax	Example
Boolean	*option* | NO*option*	**solve with nlp / NOMULTISTART;**
Keyword	*option=name*	**solve with lp / ALGORITHM=PS;**
Numeric	*option=number*	**solve with nlp / OPTTOL=1E–4;**
Parameter	*option=identifier-expression*	**solve with network / links=(INCLUDE=LINKS) concomp;**
Array	*option=array-name* [*.suffix*]	**solve with network / links=(WEIGHT=WEIGHT) tsp;**

[1] The OPTMODEL procedure never uses the LSO or network solver as a default. If the QP solver detects nonconvexity (nonconcavity) for a minimization (maximization) problem, then PROC OPTMODEL calls the NLP solver instead.

The SOLVE statement uses the value of the predeclared _SOLVER_OPTIONS_ and *_solver_*_OPTIONS_ string parameters to provide default solver options. Any options that are specified by these parameters are added before options that are specified in the SOLVE statement, with options from _SOLVER_OPTIONS_ appearing first. These options are included even when the SOLVE statement does not contain a WITH clause to specify a solver; in this case, *solver* is the name of the default solver as shown in Table 5.8.

Initially the predeclared string parameters _SOLVER_OPTIONS_ and *_solver_*_OPTIONS_ (for each solver) are empty strings, but you can assign them. You must use keywords or literal values to specify option values in these strings. Redundant white space is allowed. For example, the following statements set up some simple defaults:

```
_SOLVER_OPTIONS_ = "MAXTIME = 600"; /* options for all solvers */
_LP_OPTIONS_ = "PRESOLVER=AGGRESSIVE"; /* options for LP solver */
```

Optimization techniques that use initial values obtain them from the current values of the optimization variables unless the NOINITVAR option is specified. When the solver finishes executing, the current value of each optimization variable is replaced by the optimal value found by the solver. These values can then be used as the initial values for subsequent solver invocations.

NOTE: If a solver fails, any currently pending statement is stopped and processing continues with the next complete statement read from the input. For example, if a SOLVE statement that is enclosed in a DO group (see the section "DO Statement" on page 65) fails, then the subsequent statements in the group are not executed and processing resumes at the point immediately following the DO group. Neither an infeasible result, an unbounded result, nor reaching an iteration limit is considered to be a solver failure.

NOTE: The information that appears in the macro variable _OROPTMODEL_ (see the section "Macro Variable _OROPTMODEL_" on page 162) varies by solver.

NOTE: The RELAXINT keyword is applied immediately before the problem is passed to the solver, after any processing by the PROC OPTMODEL presolver. So the problem presented to the solver might not be equivalent to the one produced by setting the .RELAX suffix of all variables to a nonzero value. In particular, the bounds of integer variables are still adjusted to be integral, and PROC OPTMODEL's presolver might use integrality to tighten bounds further.

STOP Statement

STOP ;

The STOP statement halts the execution of all statements that contain it, including DO statements and other control or looping statements. Execution continues with the next top-level source statement. The following statements demonstrate a simple use of the STOP statement:

```
proc optmodel;
   number i, j;
   do i = 1..5;
      do j = 1..4;
         if i = 3 and j = 2 then stop;
      end;
   end;
   print i j;
```

The output is shown in Figure 5.29.

Figure 5.29 STOP Statement: Output

i	j
3	2

When the counters i and j reach 3 and 2, respectively, the STOP statement terminates both loops. Execution continues with the PRINT statement.

SUBMIT Statement

SUBMIT *arguments* [/ *options*] **;**

SAS statements **;**

ENDSUBMIT ;

The SUBMIT statement allows SAS code to be executed before PROC OPTMODEL processing continues. For example, you can use the SUBMIT statement to invoke other SAS procedures to perform analysis or to display results. The following statements use PROC SORT to order a list of nodes by decreasing priority; the nodes can be used for further processing:

```
proc optmodel;
   set<str> NODES;
   num priority{NODES};

   /* set up priority data... */

   /* sort nodes by descending priority */
   create data temppri from [id] priority;
   submit;
      proc sort;
         by descending priority;
      run;
   endsubmit;

   /* load nodes by priority */
   str nodesByPri{i in 1..card(NODES)};
   read data temppri into [_n_] nodesByPri=id;

   /* use the sorted list... */
```

The SUBMIT statement must appear as the last or only statement on a line. It is followed by lines of SAS statements, terminated by the ENDSUBMIT statement on a line of its own. The SAS statements between the SUBMIT and ENDSUBMIT statements are referred to as a *SUBMIT block*. The SUBMIT block is sent to the SAS language processor each time the SUBMIT statement is executed.

The SUBMIT block can include SAS global statements and procedure and invocations. Macros are not expanded until the SUBMIT block is executed. So you can change macro variables to modify the behavior of the SUBMIT block each time it is processed.

The *arguments* list specifies macro variables to initialize before the SUBMIT block is executed. List items are separated by spaces. Each of the *arguments* takes one of the following forms:

name
: copies the value of the PROC OPTMODEL parameter *name* to the macro variable that has the same name.

name **=** *identifier-expression*
: copies the value of the PROC OPTMODEL parameter specified by *identifier-expression* to the macro variable *name*.

name **=** *number* | "*string*" | '*string*'
: copies the value of the specified *number* or *string* constant to the macro variable *name*.

name **=** *(expression)*
: copies the result of evaluating *expression* to the macro variable *name*.

The following statements use a SUBMIT argument to modify the output each time the SUBMIT block is invoked:

```
for {i in 1..5} do;
   submit a=i;
      %put Value of a is &a..;
   endsubmit;
end;
```

The *options* in the SUBMIT statement are used to retrieve status information after a SUBMIT block is executed. Each item in the space-delimited *options* list has one of the following forms:

OK = *identifier-expression*
: specifies a PROC OPTMODEL numeric parameter location, *identifier-expression*, that is updated to indicate the success of the SUBMIT block execution. The location is set to 1 if execution is successful or 0 if errors are detected. PROC OPTMODEL continues execution when the SUBMIT block encounters errors only if the OK= option is specified.

OUT [**=**] *output-argument*
: specifies a single *output-argument* for retrieving macro variable values after each execution of the block.

OUT [**=**] **(** *output-argument* **)**
: specifies a list of space-delimited *output-arguments* for retrieving macro variable values after the block is executed.

Each *output-argument* item specifies a macro variable to copy after the block is executed. Each item takes one of the following two forms:

identifier-expression
: copies the macro variable specified by the *name* portion of the *identifier-expression* into the PROC OPTMODEL parameter location specified by *identifier-expression*.

identifier-expression **=** *name*
: copies the macro variable specified by *name* into the PROC OPTMODEL parameter location specified by *identifier-expression*.

The following statements show how to use the *options* in the SUBMIT statement to retrieve the result of a SUBMIT block execution:

```
proc optmodel;
   num success, syscc;
   submit / OK = success out syscc;
      data example;
         set notfound;
         j = i*i;
      run;
   endsubmit;
   print success syscc;
```

The DATA step fails, so the success parameter is set to 0 and syscc is set to the error code in the &SYSCC macro variable. The output is shown in Figure 5.30.

Figure 5.30 SUBMIT Statement Error Handling

success	syscc
0	1012

NOTE: The SUBMIT block runs in the same environment that the OPTMODEL procedure is running in. The SUBMIT block can change the values for options, LIBNAME librefs, FILENAME filerefs, titles, footnotes, and macro variables. The OPTMODEL procedure sees these changes but might not process them until the next top-level source statement is read.

NOTE: A SUBMIT block can reset the ODS environment of the OPTMODEL procedure. For example, the ODS SELECT and EXCLUDE lists could be cleared after the SUBMIT block executes.

NOTE: A SUBMIT statement can appear only in open code. An error message is displayed if the SUBMIT statement is read from a macro. You can avoid this limitation by placing the SUBMIT statement, SUBMIT block, and ENDSUBMIT in a separate file and by using the %INCLUDE statement to include the file in the macro.

UNFIX Statement

UNFIX *variable-list* [**=** *expression*] **;**

The UNFIX statement reverses the effect of FIX statements. The solver can vary the specified variables, variable arrays, or variable array locations specified by *variable-list*. The *variable-list* consists of one or more variable names separated by spaces.

Each variable name in the *variable-list* is an *identifier expression* (see the section "Identifier Expressions" on page 101). The UNFIX statement affects an entire variable array if the identifier expression omits the index from an array name. Each *identifier-expression* can be prefixed by an indexing set to specify the variable indices.

The *expression* specifies a new initial value that is stored in each element of the *variable-list*. If a single *identifier-expression* is specified, prefixed by an indexing set, then the *expression* is evaluated for each element in the indexing set. Otherwise the *expression* is evaluated exactly once.

The following example demonstrates the UNFIX command:

```
proc optmodel;
   var x{1..3};
   fix x;       /* fixes entire array to 0 */
   unfix x[1];  /* x[1] can now be varied again */
   unfix x[2] = 2;  /* x[2] is given an initial value 2 */
                 /* and can be varied now */
   unfix x;  /* all x indices can now be varied */
```

After the following statements are executed, the variables x[1] and x[2] are not fixed. They each hold the value 4. The variable x[3] is fixed at a value of 2.

```
proc optmodel;
   var x{1..3} init 2;
   num a = 1;
   fix x;
   unfix x[1] x[2]=a+3;
   unfix {1..2} x = a+3;  /* equivalent to prior UNFIX statement */
```

USE PROBLEM Statement

USE PROBLEM *identifier-expression* **;**

The USE PROBLEM programming statement makes the problem specified by the *identifier-expression* be the current problem. If the problem has not been previously used, the problem is created using the PROBLEM declaration corresponding to the name. The problem must have been previously declared.

Details: OPTMODEL Procedure

Named Parameters

In the example described in the section "An Unconstrained Optimization Example" on page 28, all the numeric constants that describe the behavior of the objective function were specified directly in the objective expression. This is a valid way to formulate the objective expression. However, in many cases it is inconvenient to specify the numeric constants directly. Direct specification of numeric constants can also hide the structure of the problem that is being solved. The objective expression text would need to be modified when the numeric values in the problem change. This can be very inconvenient with large models.

In PROC OPTMODEL, you can create named numeric values that behave as constants in expressions. These named values are called *parameters*. You can write an expression by using mnemonic parameter names in place of numeric literals. This produces a clearer formulation of the optimization problem. You can easily modify the values of parameters, define them in terms of other parameters, or read them from a SAS data set.

The model from this same example can be reformulated in a more general polynomial form, as follows:

```
data coeff;
   input c_xx c_x c_y c_xy c_yy;
   datalines;
1 -1  -2 -1 1
```

```
;
proc optmodel;
   var x, y;
   number c_xx, c_x, c_y, c_xy, c_yy;
   read data coeff into c_xx c_x c_y c_xy c_yy;
   min z=c_xx*x**2 + c_x*x + c_y*y + c_xy*x*y + c_yy*y**2;
   solve;
```

These statements read the coefficients from a data set, COEFF. The NUMBER statement declares the parameters. The READ DATA statement reads the parameters from the data set. You can apply this model easily to coefficients that you have generated by various means.

Indexing

Many models have large numbers of variables or parameters that can be categorized into families of similar purpose or behavior. Such families of items can be compactly represented in PROC OPTMODEL by using indexing. You can use indexing to assign each item in such families to a separate value location.

PROC OPTMODEL indexing is similar to array indexing in the DATA step, but it is more flexible. Index values can be numbers or strings, and are not required to fit into some rigid sequence. PROC OPTMODEL indexing is based on index sets, described further in the section "Index Sets" on page 103. For example, the following statement declares an indexed parameter:

```
number p{1..3};
```

The construct that follows the parameter name p, "{1..3}," is a simple index set that uses a range expression (see "Range Expression" on page 110). The index set contains the numeric members 1, 2, and 3. The parameter has distinct value locations for each of the index set members. The first such location is referenced as p[1], the second as p[2], and the third as p[3].

The following statements show an example of indexing:

```
proc optmodel;
   number p{1..3};
   p[1]=5;
   p[2]=7;
   p[3]=9;
   put p[*]=;
```

The preceding statements produce a line such as the one shown in Figure 5.31 in the log.

Figure 5.31 Indexed Parameter Output

```
p[1]=5 p[2]=7 p[3]=9
```

Index sets can also specify local dummy parameters. A dummy parameter can be used as an operand in the expressions that are controlled by the index set. For example, the assignment statements in the preceding statements could be replaced by an initialization in the parameter declaration, as follows:

```
number p{i in 1..3} init 3 + 2*i;
```

The initialization value of the parameter location p[1] is evaluated with the value of the local dummy parameter i equal to 1. So the initialization expression 3 + 2*i evaluates to 5. Similarly for location p[2], the value of i is 2 and the initialization expression evaluates to 7.

The OPTMODEL modeling language supports aggregation operators that combine values of an expression where a local dummy parameter (or parameters) ranges over the members of a set. For example, the SUM aggregation operator combines expression values by adding them together. The following statements output 21, since p[1] + p[2] + p[3] = 5 + 7 + 9 = 21:

```
proc optmodel;
   number p{i in 1..3} init 3 + 2*i;
   put (sum{i in 1..3} p[i]);
```

Aggregation operators like SUM are especially useful in objective expressions because they can combine a large number of similar expressions into a compact representation. As an example, the following statements define a trivial least squares problem:

```
proc optmodel;
   number n init 100000;
   var x{1..n};
   min z = sum{i in 1..n}(x[i] - log(i))**2;
   solve;
```

The objective function in this case is

$$z = \sum_{i=1}^{n} (x_i - \log i)^2$$

Effectively, the objective expression expands to the following large expression:

```
min z = (x[1] - log(1))**2
      + (x[2] - log(2))**2
      . . .
      + (x[99999] - log(99999))**2
      + (x[100000] - log(100000))**2;
```

Even though the problem has 100,000 variables, the aggregation operator SUM enables a compact objective expression.

NOTE: PROC OPTMODEL classifies as mathematically impure any function that returns a different value each time it is called. The RAND function, for example, falls into this category. PROC OPTMODEL disallows impure functions inside array index sets, objectives, and constraint expressions. The values of expressions that are specified in the declaration of a parameter are resolved in a nondeterministic order during threaded problem generation. Therefore, the values are also nondeterministic when these expressions use impure functions.

Types

In PROC OPTMODEL, parameters and expressions can have numeric or character values. These correspond to the elementary types named NUMBER and STRING, respectively. The NUMBER type is the same as the SAS data set numeric type. The NUMBER type includes support for missing values. The STRING type is like the SAS VARCHAR(*) character type, which allows lengths to vary as needed up to $2^{31} - 1$ bytes. However, string values are stored independently of parameters, and value references can be shared. The NUMBER and STRING types together are called the *scalar types*. You can abbreviate the type names as NUM and STR, respectively.

PROC OPTMODEL also supports set types for parameters and expressions. Sets represent collections of values of a member type, which can be a NUMBER, a STRING, or a vector of scalars (the latter is called a *tuple* and described in the following paragraphs). Members of a set all have the same member type. Members that have the same value are stored only once. For example, PROC OPTMODEL stores the set 2, 2, 2 as the set 2.

Specify a set of numbers with SET<NUMBER>. Similarly, specify a set of strings as SET<STRING>.

A set can also contain a collection of tuples, all of the same fixed length. A *tuple* is an ordered collection that contains a fixed number of elements. Each element in a tuple contains a scalar value. In PROC OPTMODEL, tuples of length 1 are equivalent to scalars. Two tuples have equal values if the elements at corresponding positions in each tuple have the same value. Within a set of tuples, the element type at a particular position in each tuple is the same for all set members. The element types are part of the set type. For example, the following statement declares parts as a set of tuples that have a string in the first element position and a number in the second element position and then initializes its elements to be <R 1>, <R 2>, <C 1>, and <C 2>.

```
set<string,number> parts = /<R 1> <R 2> <C 1> <C 2>/;
```

To create a compact model, use sets to take advantage of the structure of the problem being modeled. For example, a model might contain various values that specify attributes for each member of a group of suppliers. You could create a set that contains members that represent each supplier. You can then model the attribute values by using arrays that are indexed by members of the set.

The section "Parameters" on page 97 has more details and examples.

Names

Names are used in the OPTMODEL modeling language to refer to various entities such as parameters or variables. Names must follow the usual rules for SAS names. Names can be up to 32 bytes long and are not case sensitive. They must be declared before they are used.

Avoid declarations with names that begin with an underscore (_). These names can have special uses in PROC OPTMODEL.

Parameters

In the OPTMODEL modeling language, parameters are named locations that hold constant values. Parameter declarations specify the parameter type followed by a list of parameter names to declare. For example, the following statement declares numeric parameters named a and b:

```
number a, b;
```

Similarly, the following statements declare a set s of strings, a set n of numbers, and a set sn of tuples:

```
set<string> s;
set<number> n;
set<string, number> sn;
```

You can assign values to parameters in various ways. A parameter can be assigned a value with an assignment statement. For example, the following statements assign values to the parameter s, n, and sn in the preceding declaration:

```
s = {'a', 'b', 'c'};
n = {1, 2, 3};
sn = {<'a',1>, <'b',2>, <'c',3>};
```

Parameter values can also be assigned using a READ DATA statement (see the section "READ DATA Statement" on page 81).

A parameter declaration can provide an explicit value. To specify the value, follow the parameter name with an equal sign (=) and an expression. The value expression can be written in terms of other parameters. The declared parameter takes on a new value each time a parameter that is used in the expression changes. This automatic value update is shown in the following example:

```
proc optmodel;
   number pi=4*atan(1);
   number r;
   number circum=2*pi*r;
   r=1;
   put circum;       /* prints 6.2831853072 */
   r=2;
   put circum;       /* prints 12.566370614 */
```

The automatic update of parameter values makes it easy to perform "what if" analysis since, after the solver finds a solution, you can change parameters and reinvoke the solver. You can easily examine the effects of the changes on the optimal values.

If you declare a set parameter that has only the SET type specifier, then the element type is determined from the initialization expression. If the initialization expression is omitted or if the expression is an empty set, then the set type defaults to SET<NUMBER>. For example, the following statement implicitly declares s1 as a set of numbers:

```
set s1;
```

The following statement declares s2 as a set of strings:

```
set s2 = {'A'};
```

You can declare an array parameter by following the parameter name with an index set specification (see the section "Index Sets" on page 103). For example, declare an array of 10 numbers as follows:

```
number c{1..10};
```

Individual locations of a parameter array can be referred to with an indexing expression. For example, you can refer to the third location of parameter c as c[3]. Array index sets *cannot* be specified using a function such as RAND that returns a different value each time it is called.

Parameter names must be declared before they are used. Nonarray names become available at the end of the parameter declaration item. Array names become available after the index set specification. The latter case permits some forms of recursion in the optional initialization expression that can be supplied for a parameter.

You do not need to assign values to parameters before they are referenced. Most information in PROC OPTMODEL is stored symbolically and resolved when necessary. Values are resolved in certain statements.

For example, PROC OPTMODEL resolves a parameter used in the objective during the execution of a SOLVE statement. If no value is available during resolution, then an error is diagnosed.

Expressions

Expressions are grouped into three categories based on the types of values they can produce: logical, set, and scalar (that is, numeric or character).

Logical expressions test for a Boolean (true or false) condition. As in the DATA step, logical operators produce a value equal to either 0 or 1. A value of 0 represents a false condition, while a value of 1 represents a true condition.

Logical expression operators are not allowed in certain contexts due to syntactic considerations. For example, in the VAR statement a logical operator might indicate the start of an option. Enclose a logical expression in parentheses to use it in such contexts. The difference is illustrated by the output (Figure 5.32) of the following statements, where two variables, x and y, are declared with initial values. The PRINT statement and the EXPAND statement are used to check the initial values and the variable bounds, respectively.

```
proc optmodel;
   var x init 0.5 >= 0 <= 1;
   var y init (0.5 >= 0) <= 1;
   print x y;
   expand;
```

Figure 5.32 Logical Expression in the VAR Statement

x	y
0.5	1

```
Var x >= 0 <= 1
Var y <= 1
```

Contexts that expect a logical expression also accept numeric expressions. In such cases zero or missing values are interpreted as false, and all nonzero nonmissing numeric values are interpreted as true.

Set expressions return a set value. PROC OPTMODEL supports a number of operators that create and manipulate sets. See the section "OPTMODEL Expression Extensions" on page 105 for a description of the various set expressions. Index-set syntax is described in the section "Index Sets" on page 103.

Scalar expressions are similar to the expressions in the DATA step except for PROC OPTMODEL extensions. PROC OPTMODEL provides an IF expression (described in the section "IF-THEN/ELSE Expression" on page 107). String lengths are assigned dynamically, so there is generally no padding or truncation of string values.

Table 5.10 shows the expression operators from lower to higher precedence (a higher precedence is given a larger number). Operators that have higher precedence are applied in compound expressions before operators that have lower precedence. The table also gives the order of evaluation that is applied when multiple operators of the same precedence are used together. Operators available in both PROC OPTMODEL and the DATA step have compatible precedences, except that in PROC OPTMODEL the NOT operator has a lower precedence than the relational operators. This means that, for example, `NOT 1 < 2` is equal to `NOT (1 < 2)` (which is 0), rather than `(NOT 1) < 2` (which is 1).

Table 5.10 Expression Operator Table

Precedence	Associativity	Operator	Alternates
Logic Expression Operators			
1	Left to right	`OR`	`| !`
2	Unary	`OR{`*index-set*`}`	
		`AND{`*index-set*`}`	
3	Left to right	`AND`	`&`
4	Unary	`NOT`	`~ ^ ¬`
5	Left to right	`<`	`LT`
		`>`	`GT`
		`<=`	`LE`
		`>=`	`GE`
		`=`	`EQ`
		`~=`	`NE ^= ¬=`
6	Left to right	`IN`	
		`NOT IN`	
7	Left to right	`WITHIN`	
		`NOT WITHIN`	
Set Expression Operators			
11		`IF l THEN s1 ELSE s2`	
12	Left to right	`UNION`	
		`DIFF`	
		`SYMDIFF`	
13	Unary	`UNION{`*index-set*`}`	
14	Left to right	`INTER`	
15	Unary	`INTER{`*index-set*`}`	
16	Left to right	`CROSS`	
17	Unary	`SETOF{`*index-set*`}`	
	Right to left	`..`	`TO`
		`.. e BY`	`TO e BY`
Scalar Expression Operators			
21		`IF l THEN e`	
		`IF l THEN e1 ELSE e2`	
22	Left to right	`||`	`!!`
23	Left to right	`+ -`	
24	Unary	`SUM{`*index-set*`}`	
		`PROD{`*index-set*`}`	
		`MIN{`*index-set*`}`	
		`MAX{`*index-set*`}`	
25	Left to right	`* /`	
26	Unary	`+ -`	
	Right to left	`><`	
		`<>`	
		`**`	`^`

Primary expressions are the individual operands that are combined using the expression operators. Simple primary expressions can represent constants or named parameter and variable values. More complex primary expressions can be used to call functions or construct sets.

Table 5.11 Primary Expression Table

Expression	Description
identifier-expression	Parameter/variable reference; see the section "Identifier Expressions" on page 101
name (*argument-list*)	Function call; *argument-list* is 0 or more expressions separated by commas
n	Numeric constant
. or **.***c*	Missing value constant
"*string*" or '*string*'	String constant
{ *member-list* **}**	Set constructor; *member-list* is 0 or more scalar expressions or tuple expressions separated by commas
{ *index-set* **}**	Index set expression; returns the set of all index set members
/ *members* /	Set literal expression; compactly specifies a simple set value
(*expression* **)**	Expression enclosed in parentheses
< *element-list* **>**	Tuple expression; used with set operations; contains one or more scalar expressions separated by commas

Identifier Expressions

Use an *identifier-expression* to refer to a variable, objective, constraint, parameter or problem location in expressions or initializations. This is the syntax for *identifier-expressions*:

name [**[** *expression-1* [**,** *. . . expression-n*] **]**] [**.** *suffix* [**[** *expression* **]**]]

To refer to a location in an array, follow the array *name* with a list of scalar expressions in square brackets ([ ]). The expression values are compared to the index set that was used to declare *name*. If there is more than one expression, then the values are formed into a tuple. The expression values for a valid array location must match a member of the array's index set. For example, the following statements define a parameter array A that has two valid indices that match the tuples <1,2> and <3,4>:

```
proc optmodel;
   set<number, number> ISET = {<1,2>, <3,4>};
   number A{ISET};
   a[1,2] = 0;  /* OK */
   a[3,2] = 0;  /* invalid index */
```

The first assignment is valid with this definition of the index set, but the second fails because <3,2> is not a member of the set parameter ISET.

Specify a *suffix* to refer to auxiliary locations for variables or objectives. For more information, see the section "Suffixes" on page 135. Certain suffixes can be followed by a numeric index *expression* that selects

a particular solution saved by the SOLVE statement. For more information about solution indices, see the section "Multiple Solutions" on page 152.

Function Expressions

Most functions that can be invoked from the DATA step or the %SYSFUNC macro can be used in PROC OPTMODEL expressions. Certain functions are specific to the DATA step and cannot be used in PROC OPTMODEL. Functions specific to the DATA step include these:

- functions in the LAG, DIF, and DIM families
- functions that access the DATA step program data vector
- functions that access symbol attributes

The CALL statement can invoke SAS library subroutines. These subroutines can read and update the values of the parameters and variables that are used as arguments. See the section "CALL Statement" on page 53 for an example.

OPTMODEL arrays can be passed to SAS library functions and subroutines using one of the following argument syntax forms:

OF *array-name***[*]** [**.** *suffix* [**[** *expression* | ***** **]**]]
OF *name* **.** *suffix* **[*]**
OF *array-name* **[** *expression-1* [**,** *. . . expression-n*] **]** **.** *suffix* **[*]**

The *array-name* is the name of an array symbol. The optional *suffix* allows auxiliary values to be referenced, as described in section "Suffixes" on page 135. You can also use an *identifier-expression* with a suffix that allows a solution index, specifying an asterisk (*) in place of the index *expression*.

The OF array form is a compact alternative to listing the array or solution elements explicitly. The OF argument form is resolved into a sequence of arguments, one for each index in the array or for each solution index. The array elements appear in order of the array's index set. Solutions appear in solution index order. If you use an asterisk for both the array and solution index lists, an argument appears for each combination of array and solution indices, and the solutions for each array index are placed in contiguous arguments.

As an example, the following statements use the CALL SORTN function to sort the elements of a numeric array:

```
proc optmodel;
   number original{i in 1..8} = sin(i);
   number sorted{i in 1..8} init original[i];
   call sortn(of sorted[*]);
   print original sorted;
```

The output is shown in Figure 5.33. Eight arguments are passed to the SORTN routine. The original column shows the original order, and the sorted column has the sorted order.

Figure 5.33 Sorting Using an OF Array Argument

[1]	original	sorted
1	0.84147	-0.95892
2	0.90930	-0.75680
3	0.14112	-0.27942
4	-0.75680	0.14112
5	-0.95892	0.65699
6	-0.27942	0.84147
7	0.65699	0.90930
8	0.98936	0.98936

Index Sets

An index set represents a set of combinations of members from the component set expressions. The index set notation is used in PROC OPTMODEL to describe collections of valid array indices and to specify sets of values with which to perform an operation. Index sets can declare local dummy parameters and can further restrict the set of combinations by a selection expression.

In an index-set specification, the index set consists of one or more *index-set-items* that are separated by commas. Each *index-set-item* can include local dummy parameter declarations. An optional selection expression follows the list of *index-set-items*. The following syntax, which describes an index set, usually appears in braces ({}):

index-set-item [**,** *. . . index-set-item*] [**:** *logic-expression*]

index-set-item has these forms:

set-expression
name **IN** *set-expression*
< *name-1* [**,** *. . . name-n*] **>** **IN** *set-expression*

Names that precede the IN keyword in *index-set-items* declare local dummy parameter names. Dummy parameters correspond to the dummy index variables in mathematical expressions. For example, the following statements output the number 385:

```
proc optmodel;
   put (sum{i in 1..10} i**2);
```

The preceding statements evaluate this summation:

$$\sum_{i=1}^{10} i^2 = 385$$

In both the statements and the summation, the index name is *i*.

The last form of *index-set-item* in the list can be modified to use the SLICE expression implicitly. See the section "More on Index Sets" on page 160 for details.

Array index sets cannot be defined using functions that return different values each time the functions are called. See the section "Indexing" on page 95 for details.

Nil Values

The evaluation of certain PROC OPTMODEL expressions can produce a special value, the nil value, that is distinct from any other value of the expression's type. A nil value represents the lack of any ordinary value. You can use nil values to check for undefined parameters or to reset the effects of previous assignments or initializations.

Nil values are allowed for the following expressions:

- the operands of an equality comparison operator, = or ~=. Nil values are equal only to nil values.
- the second and third operands of an IF-THEN/ELSE expression
- the right-hand-side expression of an assignment statement. Assigning a nil value resets the effect of previous initializations or assignments of the target location.
- a parenthesized value expression for a PUT statement. Nil values are skipped on output.
- a parenthesized value expression for a PRINT statement. Nil values create an empty cell in the output.

NOTE: The *initializers* list from the INIT [*initializers*] clause for an array parameter declaration is evaluated at most once, except for any default element value. If an array element is assigned a nil value after the *initializers* are evaluated, then the default element value is used.

The _NIL_ keyword evaluates to a nil value in expressions. Nil values can also be generated when an *identifier expression* references an undefined parameter value. An *identifier expression* evaluates to a nil value instead of reporting an error when it is used as the operand of an equality comparison or as the right-hand-side expression of an assignment statement with a NUMBER, STRING, or SET parameter target. The nil value is generated when the *identifier expression* is used either directly or via the second or third operand of an IF-THEN/ELSE expression.

NOTE: Undefined parameter values are treated as errors when an *identifier expression* is used as the operand of a compound comparison (like b in (0 ~= b <= 10)) where one of the operators that are using the value is not an equality comparison.

The following code demonstrates some uses of nil values:

```
proc optmodel;
   num a{1..2};
   a[1] = 1;
   put (a[1] = _nil_)=; /* false, defined */
   put (a[2] = _nil_)=; /* true, undefined */
   put ((if a[1] then _nil_ else 2) = a[1])= /* false */;
   put ((if a[1] then _nil_ else 2) = a[2])= /* true */;
   a[1] = a[2]; /* now a[1] is undefined */
   put (a[1] = _nil_)=; /* true, undefined */

   /* array initialization list is evaluated at most once */
   num b{1..3} init [1 2 3 [*] 7];
   put b[*];     /* evaluates the initialization list */
   b[1] = _nil_; /* reset b[1] */
   put b[*];     /* b[1] replaced by the default value */
```

```
var x init (rand('UNIFORM')) >= 0 <= 10;
put x=;       /* initialized to a random value */
x = _nil_;    /* discard the current value */
put x=;       /* initialized to a new random value */
put x.ub=;    /* the declared bound */
x.ub = 1;     /* override the declared bound */
put x.ub=;    /* the overridden bound */
x.ub = _nil_; /* reverse the override */
put x.ub=;    /* the declared bound again */

/* nil values print as empty cells */
print {i in 1..3, j in 1..3}(if i+j-1=3 then _nil_ else 1/(i+j-1));
```

This code produces the output in Figure 5.34.

Figure 5.34 Nil Values

```
a[1] = _NIL_=0
a[2] = _NIL_=1
IF a[1] THEN _NIL_ ELSE 2 = a[1]=0
IF a[1] THEN _NIL_ ELSE 2 = a[2]=1
a[1] = _NIL_=1
1 2 3
7 2 3
x=0.5832970655
x=0.9936253726
x.UB=10
x.UB=1
x.UB=10
```

	1	2	3
1	1.00	0.50	
2	0.50		0.25
3		0.25	0.20

OPTMODEL Expression Extensions

PROC OPTMODEL defines several new types of expressions for the manipulation of sets. Aggregation operators combine values of an expression that is evaluated over the members of an index set. Other operators create new sets by combining existing sets, or they test relationships between sets. PROC OPTMODEL also supports an IF expression operator that can conditionally evaluate expressions. These and other such expressions are described in this section.

AND Aggregation Expression

AND { *index-set* **}** *logic-expression*

The AND aggregation operator evaluates the logical expression *logic-expression* jointly for each member of the index set *index-set*. The index set enumeration finishes early if the *logic-expression* evaluation produces a false value (zero or missing). The expression returns 0 if a false value is found or returns 1 otherwise. The following statements demonstrate both a true and a false result:

```
proc optmodel;
   put (and{i in 1..5} i < 10); /* returns 1 */
   put (and{i in 1..5} i NE 3); /* returns 0 */
```

CARD Function

CARD (*set-expression* **)**

The CARD function returns the number of members of its set operand. For example, the following statements produce the output 3 since the set has 3 members:

```
proc optmodel;
   put (card(1..3));
```

CROSS Expression

set-expression-1 **CROSS** *set-expression-2*

The CROSS expression returns the crossproduct of its set operands. The result is the set of tuples formed by concatenating the tuple value of each member of the left operand with the tuple value of each member of the right operand. Scalar set members are treated as tuples of length 1. The following statements demonstrate the CROSS operator:

```
proc optmodel;
   set s1 = 1..2;
   set<string> s2 = {'a', 'b'};
   set<number, string> s3=s1 cross s2;
   put 's3 is ' s3;
   set<number, string, number> s4 = s3 cross 4..5;
   put 's4 is ' s4;
```

This code produces the output in Figure 5.35.

Figure 5.35 CROSS Expression Output

```
s3 is {<1,'a'>,<1,'b'>,<2,'a'>,<2,'b'>}
s4 is {<1,'a',4>,<1,'a',5>,<1,'b',4>,<1,'b',5>,<2,'a',4>,<2,'a',5>,<2,'b',4>,<2,
'b',5>}
```

DIFF Expression

set-expression-1 **DIFF** *set-expression-2*

The DIFF operator returns a set that contains the set difference of the left and right operands. The result set contains values that are members of the left operand but not members of the right operand. The operands must have compatible set types. The following statements evaluate and print a set difference:

```
proc optmodel;
   put ({1,3} diff {2,3}); /* outputs {1} */
```

Equality Expression

set-expression-1 **=** *set-expression-2*

set-expression-1 **~=** *set-expression-2*

The equality comparison operators are extended to support the comparison of sets. Sets are considered equal if they have the same members. The following code demonstrates set comparison:

```
proc optmodel;
   put (/a b/ = /a b/);  /* true */
   put (/a b/ = /a/);    /* false */
   put ({} = /a b/);     /* false */
   put (1..3 = {1,3,2}); /* true, ignores order */
```

IF-THEN/ELSE Expression

IF *logic-expression* **THEN** *expression-2* [**ELSE** *expression-3*]

The IF-THEN/ELSE expression evaluates the logical expression *logic-expression* and returns the result of evaluating the second or third operand expression according to the logical test result. If the *logic-expression* is true (nonzero and nonmissing), then the result of evaluating *expression-2* is returned. If the *logic-expression* is false (zero or missing), then the result of evaluating *expression-3* is returned. The other subexpression that is not selected is not evaluated.

An ELSE clause is matched during parsing with the nearest IF-THEN clause that does not have a matching ELSE. The ELSE clause can be omitted for numeric expressions; the resulting IF-THEN is handled as if a default ELSE 0 clause were supplied.

Use the IF-THEN/ELSE expression to handle special cases in models. For example, an inventory model based on discrete time periods might require special handling for the first or last period. In the following example the initial inventory for the first period is assumed to be fixed:

```
proc optmodel;
   number T;
   var inv{1..T}, order{1..T};
   number sell{1..T};
   number inv0;
   . . .
   /* balance inventory flow */
   con iflow{i in 1..T}:
       inv[i] = order[i] - sell[i] +
       if i=1 then inv0 else inv[i-1];
   . . .
```

The IF-THEN/ELSE expression in the example models the initial inventory for a time period *i*. Usually the inventory value is the inventory at the end of the previous period, but for the first time period the inventory value is given by the inv0 parameter. The iflow constraints are linear because the IF-THEN/ELSE test subexpression does not depend on variables and the other subexpressions are linear.

IF-THEN/ELSE can be used as either a set expression or a scalar expression. The type of expression depends on the subexpression between the THEN and ELSE keywords. The type used affects the parsing of the subexpression that follows the ELSE keyword because the set form has a lower operator precedence. For example, the following two expressions are equivalent because the numeric IF-THEN/ELSE has a higher precedence than the range operator (..):

```
IF logic THEN 1 ELSE 2 .. 3

(IF logic THEN 1 ELSE 2) .. 3
```

But the set form of IF-THEN/ELSE has lower precedence than the range expression operator. So the following two expressions are equivalent:

```
IF logic THEN 1 .. 2 ELSE 3 .. 4

IF logic THEN (1 .. 2) ELSE (3 .. 4)
```

The IF-THEN and IF-THEN/ELSE operators always have higher precedence than the logic operators. So, for example, the following two expressions are equivalent:

```
IF logic THEN numeric1 < numeric2

(IF logic THEN numeric1) < numeric2
```

It is best to use parentheses when in doubt about precedence.

IN Expression

expression **IN** *set-expression*

expression **NOT IN** *set-expression*

The IN expression returns 1 if the value of the left operand is a member of the right operand set. Otherwise, the IN expression returns 0. The NOT IN operator logically negates the returned value. Unlike the DATA step, the right operand is an arbitrary set expression. The left operand can be a tuple expression. The following example demonstrates the IN and NOT IN operators:

```
proc optmodel;
   set s = 1..10;
   put (5 in s);          /* outputs 1 */
   put (-1 not in s);     /* outputs 1 */
   set<num, str> t = {<1,'a'>, <2,'b'>, <2,'c'>};
   put (<2, 'b'> in t);   /* outputs 1 */
   put (<1, 'b'> in t);   /* outputs 0 */
```

Index Set Expression

{ *index-set* **}**

The index set expression returns the set of members of an index set. This expression is distinguished from a set constructor (see the section "Set Constructor Expression" on page 111) because it contains a list of set expressions.

The following statements use an index set with a selection expression that excludes the value 3:

```
proc optmodel;
   put ({i in 1..5 : i NE 3}); /* outputs {1,2,4,5} */
```

INTER Expression

set-expression-1 **INTER** *set-expression-2*

The INTER operator returns a set that contains the intersection of the left and right operands. This is the set that contains values that are members of both operand sets. The operands must have compatible set types.

The following statements evaluate and print a set intersection:

```
proc optmodel;
   put ({1,3} inter {2,3}); /* outputs {3} */
```

INTER Aggregation Expression

INTER { *index-set* **}** *set-expression*

The INTER aggregation operator evaluates the *set-expression* for each member of the index set *index-set*. The result is the set that contains the intersection of the set of values that were returned by the *set-expression* for each member of the index set. An empty index set causes an expression evaluation error.

The following statements use the INTER aggregation operator to compute the value of $\{1,2,3,4\} \cap \{2,3,4,5\} \cap \{3,4,5,6\}$:

```
proc optmodel;
   put (inter{i in 1..3} i..i+3); /* outputs {3,4} */
```

MAX Aggregation Expression

MAX { *index-set* **}** *expression*

The MAX aggregation operator evaluates the scalar-valued expression *expression* for each member of the index set *index-set*. The result is the maximum of the values that are returned by the *expression*. Missing numeric values are handled with the SAS numeric sort order; a missing value is treated as smaller than any nonmissing numeric value. If the index set is empty and the *expression* is numeric, then the result is the negative number that has the largest absolute value representable on the machine. Expression evaluation produces an error if the index set is empty and the *expression* has a string value.

The following example produces the outputs `0.5` and `*****`:

```
proc optmodel;
   put (max{i in 2..5} 1/i);
   put (max{i in 2..5} repeat('*',i-1));
```

MIN Aggregation Expression

MIN { *index-set* **}** *expression*

The MIN aggregation operator evaluates the scalar-valued expression *expression* for each member of the index set *index-set*. The result is the minimum of the values that are returned by the *expression*. Missing numeric values are handled with the SAS numeric sort order; a missing value is treated as smaller than any nonmissing numeric value. If the index set is empty and the *expression* is numeric, then the result is the largest positive number representable on the machine. Expression evaluation produces an error if the index set is empty and the *expression* has a string value.

The following example produces the outputs `0.2` and `**`:

```
proc optmodel;
   put (min{i in 2..5} 1/i);
   put (min{i in 2..5} repeat('*',i-1));
```

Nil Constant Expression

NIL

The _NIL_ keyword evaluates to a nil value. For more information about nil values, see the section "Nil Values" on page 104.

OR Aggregation Expression

OR { *index-set* **}** *logic-expression*

The OR aggregation operator evaluates the logical expression *logic-expression* for each member of the index set *index-set*. The index set enumeration finishes early if the *logic-expression* evaluation produces a true value (nonzero and nonmissing). The result is 1 if a true value is found, or 0 otherwise. The following statements demonstrate both a true and a false result:

```
proc optmodel;
   put (or{i in 1..5} i = 2); /* returns 1 */
   put (or{i in 1..5} i = 7); /* returns 0 */
```

PROD Aggregation Expression

PROD { *index-set* **}** *expression*

The PROD aggregation operator evaluates the numeric expression *expression* for each member of the index set *index-set*. The result is the product of the values that are returned by the *expression*. This operator is analogous to the $\prod$ operator used in mathematical notation. If the index set is empty, then the result is 1.

The following example uses the PROD operator to evaluate a factorial:

```
proc optmodel;
   number n = 5;
   put (prod{i in 1..n} i); /* outputs 120 */
```

Range Expression

expression-1 .. expression-n [**BY** *expression*]

The range expression returns the set of numbers from the specified arithmetic progression. The sequence proceeds from the left operand value up to the right operand limit. The increment between numbers is 1 unless a different value is specified with a BY clause. If the increment is negative, then the progression is from the left operand down to the right operand limit. The result can be an empty set.

For compatibility with the DATA step iterative DO loop construct, the keyword TO can substitute for the range (..) operator.

The limit value is not included in the resulting set unless it belongs in the arithmetic progression. For example, the following range expression does not include 30:

```
proc optmodel;
   put (10..30 by 7); /* outputs {10,17,24} */
```

The actual numbers that the range expression "f..l by i" produces are in the arithmetic sequence

$$f, f+i, f+2i, \ldots, f+ni$$

where

$$n = \left\lfloor \frac{l-f}{i} + \sqrt{\epsilon} \right\rfloor$$

and ϵ represents the relative machine precision. The limit is adjusted to avoid arithmetic roundoff errors.

PROC OPTMODEL represents the set specified by a range expression compactly when the value is stored in a parameter location, used as a set operand of an IN or NOT IN expression, used by an iterative DO loop, or used in an index set. For example, the following expression is evaluated efficiently:

```
999998.5 IN 1..1000000000
```

Set Constructor Expression

{ [*expression-1* [**,** ... *expression-n*]] **}**

The set constructor expression returns the set of the expressions in the member list. Duplicated values are added to the set only once. A warning message is produced when duplicates are detected. The constructor expression consists of zero or more subexpressions of the same scalar type or of tuple expressions that match in length and in element types.

The following statements output a three-member set and warn about the duplicated value 2:

```
proc optmodel;
   put ({1,2,3,2}); /* outputs {1,2,3} */
```

The following example produces a three-member set of tuples, using PROC OPTMODEL parameters and variables. The output is displayed in Figure 5.36.

```
proc optmodel;
   number m = 3, n = 4;
   var x{1..4} init 1;
   string y = 'c';
   put ({<'a', x[3]>, <'b', m>, <y, m/n>});
```

Figure 5.36 Set Constructor Expression Output

```
{<'a',1>,<'b',3>,<'c',0.75>}
```

Set Literal Expression

/ *members* /

The set literal expression provides compact specification of simple set values. It is equivalent in function to the set constructor expression but minimizes typing for sets that contain numeric and string constant values.

The set members are specified by *members*, which are literal values. As with the set constructor expression, each member must have the same type.

The following statement specifies a simple numeric set:

```
/1 2.5 4/
```

The set contains the members `1`, `2.5`, and `4`. A string set could be specified as follows:

```
/Miami 'San Francisco' Seattle 'Washington, D.C.'/
```

This set contains the strings `'Miami'`, `'San Francisco'`, `'Seattle'`, and `'Washington, D.C.'`. You can specify string values in set literals without quotation marks when the text follows the rules for a SAS name. Strings that begin with a digit or contain blanks or other special characters must be specified with quotation marks.

Specify tuple members of a set by enclosing the tuple elements within angle brackets (<*elements*>). The tuple elements can be specified with numeric and string literals. The following example includes the tuple elements `<'New York', 4.5>` and `<'Chicago', -5.7>`:

```
/<'New York' 4.5> <Chicago -5.7>/
```

SETOF Aggregation Expression

SETOF { *index-set* **}** *expression*

The SETOF aggregation operator evaluates the expression *expression* for each member of the index set *index-set*. The result is the set that is formed by collecting the values returned by the operand expression. The operand can be a tuple expression. For example, the following statements produce a set of tuples of numbers with their squared and cubed values:

```
proc optmodel;
   put (setof{i in 1..3}<i, i*i, i**3>);
```

Figure 5.37 shows the displayed output.

Figure 5.37 SETOF Aggregation Expression Output

```
{<1,1,1>,<2,4,8>,<3,9,27>}
```

SLICE Expression

SLICE (< *element-1*, . . . *element-n* **>** , *set-expression* **)**

The SLICE expression produces a new set by selecting members in the operand set that match a pattern tuple. The pattern tuple is specified by the element list in angle brackets. Each *element* in the pattern tuple must specify a numeric or string expression. The expressions are used to match the values of the corresponding elements in the operand set member tuples. You can also specify an *element* by using an asterisk (*). The sequence of element values that correspond to asterisk positions in each matching tuple is combined into a tuple of the result set. At least one asterisk *element* must be specified.

The following statements demonstrate the SLICE expression:

```
proc optmodel;
   put (slice(<1,*>, {<1,3>, <1,0>, <3,1>}));
   put (slice(<*,2,*>, {<1,2,3>, <2,4,3>, <2,2,5>}));
```

These statements produce the output in Figure 5.38.

Figure 5.38 SLICE Expression Output

```
{3,0}
{<1,3>,<2,5>}
```

For the first PUT statement, <1,*> matches set members <1,3> and <1,0> but not <3,1>. The second element of each matching set tuple, corresponding to the asterisk element, becomes the value of the resulting set member. In the second PUT statement, the values of the first and third elements of the operand set member tuple are combined into a two-position tuple in the result set.

The following statements use the SLICE expression to help compute the transitive closure of a set of tuples representing a relation by using Warshall's algorithm. In these statements the set parameter dep represents a direct dependency relation.

```
proc optmodel;
   set<str,str> dep = {<'B','A'>, <'C','B'>, <'D','C'>};
   set<str,str> cl;
   set<str> cn;
   cl = dep;
   cn =  (setof{<i,j> in dep} i) inter (setof{<i,j> in dep} j);
   for {node in cn}
       cl = cl union (slice(<*,node>,cl) cross slice(<node,*>,cl));
   put cl;
```

The local dummy parameter node in the FOR statement iterates over the set cn of possible intermediate nodes that can connect relations transitively. At the end of each FOR iteration, the set parameter cl contains all tuples from the original set in addition to all transitive tuples found in the current or previous iterations.

The output in Figure 5.39 includes the indirect and direct transitive dependencies from the set dep.

Figure 5.39 Warshall's Algorithm Output

```
{<'B','A'>,<'C','B'>,<'D','C'>,<'C','A'>,<'D','B'>,<'D','A'>}
```

A special form of *index-set-item* uses the SLICE expression implicitly. See the section "More on Index Sets" on page 160 for details.

SUM Aggregation Expression

SUM { *index-set* **}** *expression*

The SUM aggregation operator evaluates the numeric expression *expression* for each member in the index set *index-set*. The result is the sum of the values that are returned by the *expression*. If the index set is empty, then the result is 0. This operator is analogous to the $\sum$ operator that is used in mathematical notation. The following statements demonstrate the use of the SUM aggregation operator:

```
proc optmodel;
   put (sum {i in 1..10} i);  /* outputs 55 */
```

SYMDIFF Expression

set-expression-1 **SYMDIFF** *set-expression-2*

The SYMDIFF expression returns the symmetric set difference of the left and right operands. The result set contains values that are members of either the left or right operand but are not members of both operands. The operands must have compatible set types.

The following example demonstrates a symmetric difference:

```
proc optmodel;
   put ({1,3} symdiff {2,3}); /* outputs {1,2} */
```

Tuple Expression

< *expression-1* [**,** . . . *expression-n*] **>**

A tuple expression represents the value of a member in a set of tuples. Each scalar subexpression inside the angle brackets represents the value of a tuple element. This form is used only with IN, SETOF, set constructor, and relational expressions.

The following statements demonstrate the tuple expression:

```
proc optmodel;
   put (<1,2,3> in setof{i in 1..2}<i,i+1,i+2>);
   put ({<1,'a'>, <2,'b'>} cross {<3,'c'>, <4,'d'>});
```

The first PUT statement checks whether the tuple <1, 2, 3> is a member of a set of tuples. The second PUT statement outputs the crossproduct of two sets of tuples that are constructed by the set constructor.

These statements produce the output in Figure 5.40.

Figure 5.40 Tuple Expression Output

```
1
{<1,'a',3,'c'>,<1,'a',4,'d'>,<2,'b',3,'c'>,<2,'b',4,'d'>}
```

You can specify tuple expressions as operands of the relational operators. For the equality operators (= and ~=), tuples are equal if their corresponding elements are equal. Ordering operators (<, <=, >, and >=) compare tuples lexicographically. That is, the corresponding tuple elements are compared left to right, and the leftmost pair of unequal elements determines the result of the comparison.

The following code demonstrates tuple comparison:

```
proc optmodel;
   put (<1,'a'> = <1,'a'>)=;  /* true, all elements equal */
   put (<1,'a'> = <1,'b'>)=;  /* false, at least one element unequal */
   put (<1,'a'> ~= <1,'b'>)=; /* true, at least one element unequal */
   put (<1,2,5> < <1,3,4>)=;  /* true, compared left to right, 2 < 3 */
```

These statements produce the output in Figure 5.41.

Figure 5.41 Tuple Comparison Output

```
<1,'a'> = <1,'a'>=1
<1,'a'> = <1,'b'>=0
<1,'a'> ~= <1,'b'>=1
<1,2,5> < <1,3,4>=1
```

UNION Expression

set-expression-1 **UNION** *set-expression-2*

The UNION expression returns the set union of the left and right operands. The result set contains values that are members of either the left or right operand. The operands must have compatible set types. The following example performs a set union:

```
proc optmodel;
   put ({1,3} union {2,3}); /* outputs {1,3,2} */
```

UNION Aggregation Expression

UNION { *index-set* **}** *set-expression*

The UNION aggregation expression evaluates the *set-expression* for each member of the index set *index-set*. The result is the set union of the values that are returned by the *set-expression*. If the index set is empty, then the result is an empty set.

The following statements demonstrate a UNION aggregation. The output is the value of {1,2,3,4} ∪ {2,3,4,5} ∪ {3,4,5,6}.

```
proc optmodel;
   put (union{i in 1..3} i..i+3); /* outputs {1,2,3,4,5,6} */
```

WITHIN Expression

set-expression-1 **WITHIN** *set-expression-2*

set-expression **NOT WITHIN** *set-expression*

The WITHIN expression returns 1 if the left operand set is a subset of the right operand set and returns 0 otherwise. (That is, the operator returns true if every member of the left operand set is a member of the right operand set.) The NOT WITHIN form logically negates the result value. The following statements demonstrate the WITHIN and NOT WITHIN operators:

```
proc optmodel;
   put ({1,3} within {2,3});     /* outputs 0 */
   put ({1,3} not within {2,3}); /* outputs 1 */
   put ({1,3} within {1,2,3});   /* outputs 1 */
```

Conditions of Optimality

Linear Programming

A standard linear program has the following formulation:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{array}$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of decision variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m\times n}$	is the matrix of constraints
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)

This formulation is called the primal problem. The corresponding dual problem (see the section "Dual Values" on page 140) is

$$\begin{array}{ll} \text{maximize} & \mathbf{b}^T\mathbf{y} \\ \text{subject to} & \mathbf{A}^T\mathbf{y} \leq \mathbf{c} \\ & \mathbf{y} \geq 0 \end{array}$$

where $\mathbf{y} \in \mathbb{R}^m$ is the vector of dual variables.

The vectors $\mathbf{x}$ and $\mathbf{y}$ are optimal to the primal and dual problems, respectively, only if there exist primal slack variables $\mathbf{s} = \mathbf{A}\mathbf{x} - \mathbf{b}$ and dual slack variables $\mathbf{w} = \mathbf{A}^T\mathbf{y} - \mathbf{c}$ such that the following *Karush-Kuhn-Tucker (KKT) conditions* are satisfied:

$$\begin{array}{rcl} \mathbf{A}\mathbf{x} + \mathbf{s} & = & \mathbf{b}, \quad \mathbf{x} \geq 0, \quad \mathbf{s} \geq 0 \\ \mathbf{A}^T\mathbf{y} + \mathbf{w} & = & \mathbf{c}, \quad \mathbf{y} \geq 0, \quad \mathbf{w} \geq 0 \\ \mathbf{s}^T\mathbf{y} & = & 0 \\ \mathbf{w}^T\mathbf{x} & = & 0 \end{array}$$

The first line of equations defines primal feasibility, the second line of equations defines dual feasibility, and the last two equations are called the complementary slackness conditions.

Nonlinear Programming

To facilitate discussion of optimality conditions in nonlinear programming, you write the general form of nonlinear optimization problems by grouping the equality constraints and inequality constraints. You also write all the general nonlinear inequality constraints and bound constraints in one form as "$\geq$" inequality constraints. Thus, you have the following formulation:

$$\begin{array}{ll} \underset{x\in\mathbb{R}^n}{\text{minimize}} & f(x) \\ \text{subject to} & c_i(x) = 0, \quad i \in \mathcal{E} \\ & c_i(x) \geq 0, \quad i \in \mathcal{I} \end{array}$$

where $\mathcal{E}$ is the set of indices of the equality constraints, $\mathcal{I}$ is the set of indices of the inequality constraints, and $m = |\mathcal{E}| + |\mathcal{I}|$.

A point x is *feasible* if it satisfies all the constraints $c_i(x) = 0, i \in \mathcal{E}$ and $c_i(x) \geq 0, i \in \mathcal{I}$. The feasible region $\mathcal{F}$ consists of all the feasible points. In unconstrained cases, the feasible region $\mathcal{F}$ is the entire $\mathbb{R}^n$ space.

A feasible point x^* is a *local solution* of the problem if there exists a neighborhood $\mathcal{N}$ of x^* such that

$$f(x) \geq f(x^*) \text{ for all } x \in \mathcal{N} \cap \mathcal{F}$$

Further, a feasible point x^* is a *strict local solution* if strict inequality holds in the preceding case; that is,

$$f(x) > f(x^*) \text{ for all } x \in \mathcal{N} \cap \mathcal{F}$$

A feasible point x^* is a *global solution* of the problem if no point in $\mathcal{F}$ has a smaller function value than $f(x^*)$; that is,

$$f(x) \geq f(x^*) \text{ for all } x \in \mathcal{F}$$

Unconstrained Optimization

The following conditions hold true for unconstrained optimization problems:

- **First-order necessary conditions**: If x^* is a local solution and $f(x)$ is continuously differentiable in some neighborhood of x^*, then

  $$\nabla f(x^*) = 0$$

- **Second-order necessary conditions**: If x^* is a local solution and $f(x)$ is twice continuously differentiable in some neighborhood of x^*, then $\nabla^2 f(x^*)$ is positive semidefinite.

- **Second-order sufficient conditions**: If $f(x)$ is twice continuously differentiable in some neighborhood of x^*, $\nabla f(x^*) = 0$, and $\nabla^2 f(x^*)$ is positive definite, then x^* is a strict local solution.

Constrained Optimization

For constrained optimization problems, the *Lagrangian function* is defined as follows:

$$L(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x)$$

where $\lambda_i, i \in \mathcal{E} \cup \mathcal{I}$, are called *Lagrange multipliers*. $\nabla_x L(x, \lambda)$ is used to denote the gradient of the Lagrangian function with respect to x, and $\nabla_x^2 L(x, \lambda)$ is used to denote the Hessian of the Lagrangian function with respect to x. The active set at a feasible point x is defined as

$$\mathcal{A}(x) = \mathcal{E} \cup \{i \in \mathcal{I} : c_i(x) = 0\}$$

You also need the following definition before you can state the first-order and second-order necessary conditions:

- **Linear independence constraint qualification and regular point**: A point x is said to satisfy the *linear independence constraint qualification* if the gradients of active constraints

  $$\nabla c_i(x), \quad i \in \mathcal{A}(x)$$

  are linearly independent. Such a point x is called a *regular point*.

You now state the theorems that are essential in the analysis and design of algorithms for constrained optimization:

- **First-order necessary conditions**: Suppose that x^* is a local minimum and also a regular point. If $f(x)$ and $c_i(x), i \in \mathcal{E} \cup \mathcal{I}$, are continuously differentiable, there exist Lagrange multipliers $\lambda^* \in \mathbb{R}^m$ such that the following conditions hold:

  $$\nabla_x L(x^*, \lambda^*) = \nabla f(x^*) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i^* \nabla c_i(x^*) = 0$$

  $$\begin{aligned} c_i(x^*) &= 0, \quad i \in \mathcal{E} \\ c_i(x^*) &\geq 0, \quad i \in \mathcal{I} \\ \lambda_i^* &\geq 0, \quad i \in \mathcal{I} \\ \lambda_i^* c_i(x^*) &= 0, \quad i \in \mathcal{I} \end{aligned}$$

  The preceding conditions are often known as the *Karush-Kuhn-Tucker conditions*, or *KKT conditions* for short.

- **Second-order necessary conditions**: Suppose that x^* is a local minimum and also a regular point. Let λ^* be the Lagrange multipliers that satisfy the KKT conditions. If $f(x)$ and $c_i(x), i \in \mathcal{E} \cup \mathcal{I}$, are twice continuously differentiable, the following conditions hold:

  $$z^T \nabla_x^2 L(x^*, \lambda^*) z \geq 0$$

  for all $z \in \mathbb{R}^n$ that satisfy

  $$\nabla c_i(x^*)^T z = 0, \quad i \in \mathcal{A}(x^*)$$

- **Second-order sufficient conditions**: Suppose there exist a point x^* and some Lagrange multipliers λ^* such that the KKT conditions are satisfied. If

  $$z^T \nabla_x^2 L(x^*, \lambda^*) z > 0$$

  for all $z \in \mathbb{R}^n$ that satisfy

  $$\nabla c_i(x^*)^T z = 0, \quad i \in \mathcal{A}(x^*)$$

  then x^* is a strict local solution.

  Note that the set of all such z's forms the null space of the matrix $\left[\nabla c_i(x^*)^T\right]_{i \in \mathcal{A}(x^*)}$. Thus, you can search for strict local solutions by numerically checking the Hessian of the Lagrangian function projected onto the null space. For a rigorous treatment of the optimality conditions, see Fletcher (1987) and Nocedal and Wright (1999).

Data Set Input/Output

You can use the CREATE DATA and READ DATA statements to exchange PROC OPTMODEL data with SAS data sets. The statements can move data into and out of PROC OPTMODEL parameters and variables. For example, the following statements use a CREATE DATA statement to save the results from an optimization into a data set:

```
proc optmodel;
   var x;
   min z = (x-5)**2;
   solve;
   create data optdata from xopt=x z;
```

These statements write a single observation into the data set OPTDATA. The data set contains two variables, xopt and z, and the values contain the optimized values of the PROC OPTMODEL variable x and objective z, respectively. The statement "xopt=x" renames the variable x to xopt.

The group of values held by a data set variable in different observations of a data set is referred to as a *column*. The READ DATA and CREATE DATA statements specify a set of columns for a data set and define how data are to be transferred between the columns and PROC OPTMODEL parameters.

Columns in square brackets ([ ]) are handled specially. Such columns are called *key columns*. Key columns specify element values that provide an implicit index for subsequent array columns. The following example uses key columns with the CREATE DATA statement to write out variable values from an array:

```
proc optmodel;
   set LOCS = {'New York', 'Washington', 'Boston'};  /* locations */
   set DOW = 1..7;  /* day of week */
   var s{LOCS, DOW} init 1;
   create data soldata from [location day_of_week]={LOCS, DOW} sale=s;
```

In this case the optimization variable s is initialized to a value of 1 and is indexed by values from the set parameters LOCS and DOW. The output data set contains an observation for each combination of values in these sets. The output data set contains three variables, location, day_of_week, and sale. The data set variables location and day_of_week save the index element values for the optimization variable s that is written in each observation. The data set created is shown in Figure 5.42.

Figure 5.42 Data Sets Created

Data Set: SOLDATA

Obs	location	day_of_week	sale
1	New York	1	1
2	New York	2	1
3	New York	3	1
4	New York	4	1
5	New York	5	1
6	New York	6	1
7	New York	7	1
8	Washington	1	1
9	Washington	2	1
10	Washington	3	1
11	Washington	4	1
12	Washington	5	1
13	Washington	6	1
14	Washington	7	1
15	Boston	1	1
16	Boston	2	1
17	Boston	3	1
18	Boston	4	1
19	Boston	5	1
20	Boston	6	1
21	Boston	7	1

Note that the key columns in the preceding example do not name existing PROC OPTMODEL variables. They create new local dummy parameters, location and day_of_week, in the same manner as dummy parameters in index sets. These local parameters can be used in subsequent columns. For example, the following statements demonstrate how to use a key column value in an expression for a later column value:

```
proc optmodel;
   create data tab
          from [i]=(1..10)
          Square=(i*i) Cube=(i*i*i);
```

These statements create a data set that has 10 observations that hold squares and cubes of the numbers from 1 to 10. The key column variable here is named i and is explicitly assigned the values from 1 to 10, while the data set variables Square and Cube hold the square and cube, respectively, of the corresponding value of i.

In the preceding example the key column values are simply the numbers from 1 to 10. The value is the same as the observation number, so the variable i is redundant. You can remove the data set variable for a key column via the DROP data set option, as follows:

```
proc optmodel;
   create data tab2 (drop=i)
          from [i] =(1..10)
          Square=(i*i) Cube=(i*i*i);
```

The local parameters declared by key columns receive their values in various ways. For a READ DATA statement, the key column values come from the data set variables for the column. In a CREATE DATA

statement, the values can be defined explicitly, as shown in the previous example. Otherwise, the CREATE DATA statement generates a set of values that combines the index sets of array columns that need implicit indexing. The statements that produce the output in Figure 5.42 demonstrate implicit indexing.

Use a suffix ("Suffixes" on page 135) to read or write auxiliary values, such as variable bounds or constraint duals. For example, consider the following statements:

```
data pdat;
   input p $ maxprod cost;
   datalines;
ABQ    12  0.7
MIA     9  0.6
CHI    14  0.5
;

proc optmodel;
   set<string> plants;
   var prod{plants} >= 0;
   number cost{plants};
   read data pdat into plants=[p] prod.ub=maxprod cost;
```

The statement "plants=[p]" in the READ DATA statement declares p as a key column and instructs PROC OPTMODEL to store the set of plant names from the data set variable p into the set parameter plants. The statement assigns the upper bound for the variable prod indexed by p to be the value of the data set variable maxprod. The cost parameter location indexed by p is also assigned to be the value of the data set variable cost.

The target variables prod and cost in the preceding example use implicit indexing. Indexing can also be performed explicitly. The following version of the READ DATA statement makes the indices explicit:

```
read data pdat into plants=[p] prod[p].ub=maxprod cost[p];
```

Explicit indexing is useful when array indices need to be transformed from the key column values in the data set. For example, the following statements reverse the order in which elements from the data set are stored in an array:

```
data abcd;
   input letter $ @@;
   datalines;
a b c d
;

proc optmodel;
   set<num> subscripts=1..4;
   string  letter{subscripts};
   read data abcd into [_N_] letter[5-_N_];
   print letter;
```

The output from this example appears in Figure 5.43.

Figure 5.43 READ DATA Statement: Explicit Indexing

[1]	letter
1	d
2	c
3	b
4	a

The following example demonstrates the use of explicit indexing to save sequential subsets of an array in individual data sets:

```
data revdata;
   input month rev @@;
   datalines;
1 200 2 345 3 362 4 958
5 659 6 804 7 487 8 146
9 683 10 732 11 652 12 469
;

proc optmodel;
   set m = 1..3;
   var revenue{1..12};
   read data revdata into [_N_] revenue=rev;
   create data qtr1 from [month]=m revenue[month];
   create data qtr2 from [month]=m revenue[month+3];
   create data qtr3 from [month]=m revenue[month+6];
   create data qtr4 from [month]=m revenue[month+9];
```

Each CREATE DATA statement generates a data set that represents one quarter of the year. Each data set contains the variables month and revenue. The data set qtr2 is shown in Figure 5.44.

Figure 5.44 CREATE DATA Statement: Explicit Indexing

Obs	month	revenue
1	1	958
2	2	659
3	3	804

Data set names and options that are used in PROC OPTMODEL statements can also be generated dynamically, by using an expression in parentheses to specify the name and options. The expression must evaluate to a string value with the following form:

[*libref* **.**] *member* [**(** *options* **)**]

Dynamic data set names are useful for processing data sets inside looping statements such as a FOR statement, particularly when the loop contains a SOLVE statement. You can replace the multiple CREATE DATA statements in the previous example with a single statement in a loop:

```
proc optmodel;
   set m = 1..3;
   var revenue{1..12};
   read data revdata into [_N_] revenue=rev;
   for {q in 1..4}
      create data ("qtr" || q)
             from [month]=m revenue[month+(q-1)*3];
```

Control Flow

Most of the control flow statements in PROC OPTMODEL are familiar to users of the DATA step or the IML procedure. PROC OPTMODEL supports the IF statement, DO blocks, the iterative DO statement, the DO WHILE statement, and the DO UNTIL statement. You can also use the CONTINUE, LEAVE, and STOP statements to modify control flow.

PROC OPTMODEL adds the FOR statement. This statement is similar in operation to an iterative DO loop. However, the iteration is performed over the members of an index set. This form is convenient for iteration over all the locations in an array, since the valid array indices are also defined using an index set. For example, the following statements initialize the array parameter A, indexed by i and j, to random values sampled from a normal distribution with mean 0 and variance 1:

```
proc optmodel;
   set R=1..10;
   set C=1..5;
   number A{R, C};
   for {i in R, j in C}
      A[i, j]=rand('NORMAL');
```

The FOR statement provides a convenient way to perform a statement such as the preceding assignment statement for each member of a set.

Formatted Output

PROC OPTMODEL provides two primary means of producing formatted output. The PUT statement provides output of data values with detailed format control. The PRINT statement handles arrays and produces formatted output in tabular form.

The PUT statement is similar in syntax to the PUT statement in the DATA step and in PROC IML. The PUT statement can output data to the SAS log, the SAS listing, or an external file. Arguments to the PUT statement specify the data to output and provide instructions for formatting. The PUT statement provides enough control to create reports within PROC OPTMODEL. However, typically the PUT statement is used to produce output for debugging or to quickly check data values.

The following example demonstrates some features of the PUT statement:

```
proc optmodel;
   number a=1.7, b=2.8;
   set s={a,b};
   put a b;         /* list output */
   put a= b=;       /* named output */
   put 'Value A: ' a 8.1 @30 'Value B: ' b 8.; /* formatted */
   string str='Ratio (A/B) is:';
   put str (a/b);  /* strings and expressions */
   put s=;          /* named set output */
```

These statements produce the output in Figure 5.45.

Figure 5.45 PUT Statement Output

```
1.7 2.8
a=1.7 b=2.8
Value A:      1.7            Value B:        3
Ratio (A/B) is: 0.6071428571
s={1.7,2.8}
```

The first PUT statement demonstrates list output. The numeric data values are output in a default format, BEST12., with leading and trailing blanks removed. A blank space is inserted after each data value is output. The second PUT statement uses the equal sign (=) to request that the variable name be output along with the regular list output.

The third PUT statement demonstrates formatted output. It uses the @ operator to position the output in a specific column. This style of output can be used in report generation. The format specification "8." causes the displayed value of parameter b to be rounded.

The fourth PUT statement shows the output of a string value, str. It also outputs the value of an expression enclosed in parentheses. The final PUT statement outputs a set along with its name.

The default destination for PUT statement output is the SAS log. The FILE and CLOSEFILE statements can be used to send output to the SAS listing or to an external data file. Multiple files can be open at the same time. The FILE statement selects the current destination for PUT statement output, and the CLOSEFILE statement closes the corresponding file. See the section "FILE Statement" on page 70 for more details.

The PRINT statement is designed to output numeric and string data in the form of tables. The PRINT statement handles the details of formatting automatically. However, the output format can be overridden by PROC OPTMODEL options and through Output Delivery System (ODS) facilities.

The PRINT statement can output array data in a table form that contains a row for each combination of array index values. This form uses columns to display the array index values for each row and uses other columns to display the value of each requested data item. The following statements demonstrate the table form:

```
proc optmodel;
   number square{i in 0..5} = i*i;
   number recip{i in 1..5} = 1/i;
   print square recip;
```

The PRINT statement produces the output in Figure 5.46.

Figure 5.46 PRINT Statement Output (List Form)

[1]	square	recip
0	0	
1	1	1.00000
2	4	0.50000
3	9	0.33333
4	16	0.25000
5	25	0.20000

The first table column, labeled "[1]," contains the index values for the parameters square and recip. The columns that are labeled square and recip contain the parameter values for each array index. For example, the last row corresponds to the index 5 and the value in the last column is 0.2, which is the value of recip[5].

Note that the first row of the table contains no value in the recip column. Parameter location recip[0] does not have a valid index, so no value is printed. The PRINT statement does not display variables that are undefined or have invalid indices. This permits arrays that have similar indexing to be printed together. The sets of defined indices in the arrays are combined to generate the set of indices shown in the table.

Also note that the PRINT statement has assigned formats and widths that differ between the square and recip columns. The PRINT statement assigns a default fixed-point format to produce the best overall output for each data column. The format that is selected depends on the PDIGITS= and PWIDTH= options.

The PDIGITS= and PWIDTH= options specify the desired significant digits and formatted width, respectively. If the range of magnitudes is large enough that no suitable format can be found, then the data item is displayed in scientific format. The table in the preceding example displays the last column with five decimal places in order to display the five significant digits that were requested by the default PDIGITS= value. The square column, on the other hand, does not need any decimal places.

The PRINT statement can also display two-dimensional arrays in matrix form. If the list following the PRINT statement contains only a single array that has two index elements, then the array is displayed in matrix form when it is sufficiently dense (otherwise the display is in table form). In this form the left-most column contains the values of the first index element. The remaining columns correspond to and are labeled by the values of the second index element. The following statements print an example of matrix form:

```
proc optmodel;
   set R=1..6;
   set C=1..4;
   number a{i in R, j in C} = 10*i+j;
   print a;
```

The PRINT statement produces the output in Figure 5.47.

Figure 5.47 PRINT Statement Output (Matrix Form)

	a			
	1	**2**	**3**	**4**
1	11	12	13	14
2	21	22	23	24
3	31	32	33	34
4	41	42	43	44
5	51	52	53	54
6	61	62	63	64

In the example the first index element ranges from 1 to 6 and corresponds to the table rows. The second index element ranges from 1 to 4 and corresponds to the table columns. Array values can be found based on the row and column values. For example, the value of parameter a[3,2] is 32. This location is found in the table in the row labeled “3” and the column labeled “2.”

ODS Table and Variable Names

PROC OPTMODEL assigns a name to each table it creates. You can use these names to reference the table when you use the Output Delivery System (ODS) to select tables and create output data sets. The names of tables common to all solvers are listed in Table 5.12. Some solvers can generate additional tables; see the

individual solver chapters for more information. For more information about ODS, see *SAS Output Delivery System: User's Guide*.

Table 5.12 ODS Tables Produced in PROC OPTMODEL

ODS Table Name	Description	Statement/Option
DerivMethods	List of derivatives used by the solver, including the method of computation	SOLVE
OptStatistics	Solver-dependent description of the resources required for solution, including function evaluations and solver time	SOLVE
PrintTable	Specified parameter or variable values	PRINT
ProblemSummary	Description of objective, variables, and constraints	SOLVE
ProfileInfo	Detailed timing of statements and declarations	PROFILE
SolutionSummary	Overview of solution, including solver-dependent solution quality values	SOLVE
SolverOptions	List of solver options and their values	SOLVE
Timing	Detailed solution timing	SOLVE

To guarantee that ODS output data sets contain information from all executed statements, use the PERSIST= option in the ODS OUTPUT statement. For details, see *SAS Output Delivery System: User's Guide*. **NOTE:** The SUBMIT statement resets ODS SELECT and EXCLUDE lists.

Table 5.13 lists the variable names of the preceding tables used in the ODS template of the OPTMODEL procedure.

Table 5.13 Variable Names for the ODS Tables Produced in PROC OPTMODEL

ODS Table Name	Variables
DerivMethods	Label1, cValue1, and nValue1
OptStatistics	Label1, cValue1, and nValue1
PrintTable (matrix form)	ROW, COL1 – COL*n*
PrintTable (table form)	COL1 – COL*n*, *identifier-expression(_suffix)*
ProblemSummary	Label1, cValue1, and nValue1
ProfileInfo	Item, Line, Column, Count, MemSize (optional), NetTime, WaitTime, and PctTime
SolutionSummary	Label1, cValue1, and nValue1
SolverOptions	Label1, cValue1, nValue1, cValue2, and nValue2
Timing	Label1, cValue1, nValue1, cValue2, and nValue2

The PRINT statement produces an ODS table named PrintTable. The variable names that are used depend on the display format used. See the section "Formatted Output" on page 123 for details about choosing the display format.

For the PRINT statement with table format, the columns that display array indices are named COL1–COL*n*, where *n* is the number of index elements. Columns that display values from identifier expressions are named using the expression's name and suffix. The identifier name becomes the output variable name if no suffix is used. Otherwise the variable name is formed by appending an underscore (_) and the suffix to the identifier name. Columns that display the value of expressions are named COL*n*, where *n* is the column number in the table.

For the PRINT statement with matrix format, the first column has the variable name ROW. The remaining columns are named COL1–COL*n*, where *n* is the number of distinct column indices. When an ODS table displays values from identifier expressions, a label is generated based on the expression's name and suffix, as described for column names in the case of table format.

The PRINTLEVEL= option controls the ODS tables produced by the SOLVE and COFOR statements. When PRINTLEVEL=0, the statements produce no ODS tables. When PRINTLEVEL=1, the SOLVE statement produces the ODS tables ProblemSummary and SolutionSummary. When PRINTLEVEL=2, the SOLVE statement produces the ODS tables ProblemSummary, SolverOptions, DerivMethods, SolutionSummary, OptStatistics, and Timing.

The following statements generate several ODS tables and write each table to a SAS data set:

```
proc optmodel printlevel=2;
   ods output PrintTable=expt ProblemSummary=exps DerivMethods=exdm
              SolverOptions=exso SolutionSummary=exss OptStatistics=exos
              Timing=exti;
   var x{1..2} >= 0;
   min z = 2*x[1] + 3 * x[2] + x[1]**2 + 10*x[2]**2
           + 2.5*x[1]*x[2] + x[1]**3;
   con c1: x[1] - x[2] <= 1;
   con c2: x[1] + 2*x[2] >= 100;
   solve;
   print x;
```

The data set expt contains the PrintTable table and is shown in Figure 5.48. The variable names are COL1 and x.

Figure 5.48 PrintTable ODS Table

PrintTable

Obs	COL1	x
1	1	10.448
2	2	44.776

The data set exps contains the ProblemSummary table and is shown in Figure 5.49. The variable names are Label1, cValue1, and nValue1. The rows describe the instance, and the description depends on the form of the problem. In most solvers, the rows describe the objective function, variables, and constraints. In the network solver, the rows describe the number of nodes, the number of edges, the directedness of the graph, and the type of problem solved over the graph.

Figure 5.49 ProblemSummary ODS Table

ProblemSummary

Obs	Label1	cValue1	nValue1
1	Objective Sense	Minimization	.
2	Objective Function	z	.
3	Objective Type	Nonlinear	.
4			.
5	Number of Variables	2	2.000000
6	Bounded Above	0	0
7	Bounded Below	2	2.000000
8	Bounded Below and Above	0	0
9	Free	0	0
10	Fixed	0	0
11			.
12	Number of Constraints	2	2.000000
13	Linear LE (<=)	1	1.000000
14	Linear EQ (=)	0	0
15	Linear GE (>=)	1	1.000000
16	Linear Range	0	0

The data set exso contains the SolverOptions table and is shown in Figure 5.50. The variable names are Label1, cValue1, nValue1, cValue2, and nValue2. The rows, which depend on the solver called by PROC OPTMODEL, list the values taken by each of the solver options. The presence of an asterisk (*) next to an option indicates that a value has been specified for that option.

Figure 5.50 SolverOptions ODS Table

SolverOptions

Obs	Label1	cValue1	nValue1	cValue2	nValue2
1	ALGORITHM	INTERIORPOINT	.		.
2	FEASTOL	1E-6	0.000001000		.
3	HESSTYPE	FULL	.		.
4	LOGFREQ	1	1.000000		.
5	MAXITER	5000	5000.000000		.
6	MAXTIME	I	I		.
7	NTHREADS	16	16.000000		.
8	OBJLIMIT	1E20	1E20		.
9	OPTTOL	1E-6	0.000001000		.
10	PRESERVEINIT	OFF	.		.
11	SOLTYPE	1	1.000000		.
12	TIMETYPE	REAL	.		.

The data set exdm contains the DerivMethods table, which displays the methods of derivative computation, and is shown in Figure 5.51. The variable names are Label1, cValue1, and nValue1. The rows, which depend on the derivatives used by the solver, specify the method used to calculate each derivative.

Figure 5.51 DerivMethods ODS Table

DerivMethods

Obs	Label1	cValue1	nValue1
1	Objective Gradient	Analytic Formulas	.
2	Objective Hessian	Analytic Formulas	.

The data set exss contains the SolutionSummary table and is shown in Figure 5.52. The variable names are Label1, cValue1, and nValue1. The rows give an overview of the solution, including the solver chosen, the objective value, and the solution status. Depending on the values returned by the solver, the SolutionSummary table might also include some solution quality values such as optimality error and infeasibility. The values in the SolutionSummary table appear in the _OROPTMODEL_ macro variable; each solver chapter has a section that describes the solver's contribution to this macro variable.

Figure 5.52 SolutionSummary ODS Table

SolutionSummary

Obs	Label1	cValue1	nValue1
1	Solver	NLP	.
2	Algorithm	Interior Point	.
3	Objective Function	z	.
4	Solution Status	Optimal	.
5	Objective Value	22623.347101	22623
6			.
7	Optimality Error	5E-7	0.000000500
8	Infeasibility	0	0
9			.
10	Iterations	5	5.000000
11	Presolve Time	0.00	0.000194
12	Solution Time	0.01	0.009540

The data set exos contains the OptStatistics table, which displays the optimization statistics, and is shown in Figure 5.53. The variable names are Label1, cValue1, and nValue1. The rows, which depend on the solver called by PROC OPTMODEL, describe the amount of time and the function evaluations that are used by the solver and associated processing. Times are displayed in seconds of clock or CPU time according to the value of the TIMETYPE= option that is used by the solver.

Figure 5.53 OptStatistics ODS Table

OptStatistics

Obs	Label1	cValue1	nValue1
1	Function Evaluations	28	28.000000
2	Gradient Evaluations	28	28.000000
3	Hessian Evaluations	6	6.000000
4	Problem Generation Time	0.00	0.001082
5	Code Generation Time	0.03	0.028820
6	Presolve Time	0.00	0.000194
7	Solution Time	0.01	0.009540
8	Total Time	0.15	0.148620

Problem generation is the process of combining the model with the data into a format that solvers can use. This includes computing equation coefficients, but it does not include reading data or evaluating other programming statements. Code generation is compiles code for nonlinear equations in the model and performs other analysis that is needed prior to solver evaluations. The time required for problem generation will be negligible if the model contains only linear equations. The presolve time in this table includes the time used by the PROC OPTMODEL presolver and any presolver that is part of the solver. Solution time is the sum of the times used by the presolvers and the solver. The presolve and solution times also appear in the SolutionSummary table. The OptStatistics table includes a total time, which is the sum of times for problem generation, code generation, solution, and overhead in the SOLVE statement. Overhead includes solver setup, postprocessing, and ODS table output.

The Timing table provides an alternate breakdown of SOLVE statement timing. Times in this table are shown in seconds of clock time. The data set exti, which is shown in Figure 5.54, contains the Timing table data and statistics. The variable names are Label1, cValue1, nValue1, cValue2, and nValue2. The values present depend on the solver and on the context of the SOLVE statement.

Figure 5.54 Timing ODS Table

Timing

Obs	Label1	cValue1	nValue1	cValue2	nValue2
1	Problem Generation	0.0010821819	0.00	0.0072783425	0.73%
2	OPTMODEL Presolver	0.0001940727	0.00	0.001305259	0.13%
3	Solver Initialization	0.1086409092	0.11	0.7306772755	73.07%
4	Code Generation	0.0288200378	0.03	0.1938325709	19.38%
5	Solver	0.0093460083	0.01	0.0628576835	6.29%
6	Solver Postprocessing	0.0006020069	0.00	0.0040488686	0.40%

Some of the Timing table values have already been described for the OptStatistics table. Solver initialization time is overhead in the SOLVE statement before the solver starts. Solver time includes execution of the solver and its associated preprocessor, if any. Solver postprocessing time is overhead in the SOLVE statement after the solver has completed. Several table values appear only for a SOLVE statement called within a COFOR loop. These values are shown in Table 5.14.

Table 5.14 Solver Timing Values for COFOR Loops

Label	Description
Waiting for Multiple Threads	Time waiting for the required number of threads to become available on the client machine when NTHREADS > 1
Waiting for Grid	Time waiting for the distributed computing environment to become ready to accept a new problem
Sending to Grid	Time to transmit a solver problem description to the distributed computing environment
Receiving from Grid	Time to receive solver results from the distributed computing environment
Waiting after Solver	Time between solver completion and the resumption of the SOLVE statement for processing the results
Grid Overhead	Time required for processing in the distributed computing environment that is not included in the preceding times

Constraints

You can add constraints to a PROC OPTMODEL model. The solver tries to satisfy the specified constraints while minimizing or maximizing the objective.

Constraints in PROC OPTMODEL have names. By using the name, you can examine various attributes of the constraint, such as the dual value that is returned by the solver (see the section "Suffixes" on page 135 for details). A constraint is not allowed to have the same name as any other model component.

PROC OPTMODEL provides a default name if none is supplied by the constraint declaration. The predefined array _ACON_ provides names for otherwise anonymous constraints. The predefined numeric parameter _NACON_ contains the number of such constraints. The constraints are assigned integer indices in sequence, so _ACON_[1] refers to the first unnamed constraint declared, while _ACON_[_NACON_] refers to the newest.

Consider the following example of a simple model that has a constraint:

```
proc optmodel;
   var x, y;
   min r = x**2 + y**2;
   con c: x+y >= 1;
   solve;
   print x y;
```

Without the constraint named c, the solver would find the point $x = y = 0$ that has an objective value of 0. However, the constraint makes this point infeasible. The resulting output is shown in Figure 5.55.

Figure 5.55 Constrained Model Solution

Problem Summary	
Objective Sense	Minimization
Objective Function	r
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	1
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	2
Hessian Diagonal Elements	2
Hessian Elements Below Diagonal	0

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	r
Solution Status	Optimal
Objective Value	0.5
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Duality Gap	0
Complementarity	0
Iterations	4
Presolve Time	0.00
Solution Time	0.01

x	y
0.5	0.5

The solver has found the point where the objective function is minimized in the region $x + y \geq 1$. This is actually on the border of the region: the constraint c is active (see the section "Dual Values" on page 140 for details).

In the preceding example the constraint c had only a lower bound. You can specify constraints that have both upper and lower bounds. For example, replacing the constraint c in the previous example would further restrict the feasible region:

```
con c: 3 >= x+y >= 1;
```

PROC OPTMODEL standardizes constraints to collect the expression terms that depend on variables and to separate the expression terms that are constant. When there is a single equality or inequality operator, the separable constant terms are moved to the right operand while the variable terms are moved to the left operand. For range constraints, the separable constant terms from the middle expression are subtracted from the lower and upper bounds. You can see the standardized constraints with the use of the EXPAND statement in the following example. Consider the following PROC OPTMODEL statements:

```
proc optmodel;
   var x{1..3};
   con b: sum{i in 1..3}(x[i] - i) = 0;
   expand b;
```

These statements produce an optimization problem with the following constraint:

```
(x[1] - 1) + (x[2] - 2) + (x[3] - 3) = 0
```

The EXPAND statement produces the output in Figure 5.56.

Figure 5.56 Expansion of a Standardized Constraint

```
Constraint b: x[1] + x[2] + x[3] = 6
```

Here the i separable constant terms in the operand of the SUM operation were moved to the right-hand side of the constraint. The sum of these i values is 6.

After standardization the constraint expression that contains all the variables is called the *body* of the constraint. You can reference the current value of the body expression by attaching the .body suffix to the constraint name. Similarly, the upper and lower bound expressions can be referenced by using the .ub and .lb suffixes, respectively. (See the section "Suffixes" on page 135 for more information.)

As a result of standardization, the value of a body expression depends on how the corresponding constraint is entered. The following example demonstrates how using equivalent relational syntax can result in different .body values:

```
proc optmodel;
    var x init 1;
    con c1: x**2 <= 5;
    con c2: 5 >= x**2;
    con c3: -x**2 >= -5;
    con c4: -5 <= -x**2;
    expand;
    print c1.body c2.body c3.body c4.body;
```

The EXPAND and PRINT statements produce the output in Figure 5.57.

Figure 5.57 Expansion and Body Values of Standardized Constraints

```
Var x
Constraint c1: x**2 <= 5
Constraint c2: -x**2 >= -5
Constraint c3: -x**2 >= -5
Constraint c4: --x**2 <= 5
```

c1.BODY	c2.BODY	c3.BODY	c4.BODY
1	-1	-1	1

CAUTION: Each constraint has an associated *dual value* (see "Dual Values" on page 140). As a result of standardization, the sign of a dual value depends in some instances on the way in which the corresponding constraint is entered into PROC OPTMODEL. In the case of a minimization objective with one-sided constraint $g(x) \geq L$, avoid entering the constraint as $L \leq g(x)$. For example, the following statements produce a value of 2:

```
proc optmodel;
   var x;
   min o1 = x**2;
   con c1: x >= 1;
   solve;
   print (c1.dual);
```

Replacing the constraint as follows results in a value of –2:

```
   con c1: 1 <= x;
```

In the case of a maximization objective with the one-sided constraint $g(x) \leq U$, avoid entering the constraint as $U \geq g(x)$.

When a constraint has variables on both sides, the sign of the dual value depends on the direction of the inequality. For example, you can enter the following constraint:

```
   con c1: x**5 - y + 8 <= 5*x + y**2;
```

This is a $\leq$ constraint, so c1.dual is nonpositive. If you enter the same constraint as follows, then c1.dual is nonnegative:

```
   con c1: 5*x + y**2 >= x**5 - y + 8;
```

It is also important to note that the signs of the dual values are negated in the case of maximization. The following statements output a value of 2:

```
proc optmodel;
   var x;
   min o1 = x**2;
   con c1: 1 <= x <= 2;
   solve;
   print (c1.dual);
```

Changing the objective function as follows yields the same value of x, but c1.dual now holds the value –2:

```
max o2 = -x**2;
solve;
print (c1.dual);
```

NOTE: A simple bound constraint on a decision variable x can be entered either by using a CONSTRAINT declaration or by assigning values to x.lb and x.ub. If you require dual values for simple bound constraints, use the CONSTRAINT declaration.

Constraints that are specified using relational operators can be linear or nonlinear. PROC OPTMODEL determines the type of constraint automatically by examining the form of the body expression. Subexpressions that do not involve variables are treated as constants. Constant subexpressions that are multiplied by or added to linear subexpressions produce new linear subexpressions. For example, constraint A in the following statements is linear:

```
proc optmodel;
   var x{1..3};
   con A: 0.5*(x[1]-x[2]) + x[3] >= 0;
```

Suffixes

You can use suffixes with *identifier-expressions* to retrieve and modify various auxiliary values maintained by the solver. The values of the suffixes can come from expressions in the declaration of the name that is suffixed. For example, the following declaration of variable v provides the values of several suffixes of v at the same time:

```
var v >= 0 <= 2 init 1 suffixes=(label='X0000000' status init 'L');
```

The preceding example sets the lower and upper bounds for variable v (that is, v.lb and v.ub, respectively) to 0 and 2. The solver label (v.label) is set to the string 'X0000000', and the initial solver status (v.status) is 'L'. The initial value of v is also set to 1. You can use more complex expressions in a declaration, as shown in the following example:

```
num ubound{1..N} init 2;
var x{i in 1..N} >= 0 <= ubound[i] suffixes(label='X'||put(i,z7.));
```

The preceding declaration makes the upper bound value for each element of x equal to the value of the element in the ubound array with a corresponding index. That is, for each i in 1 to N, x[i].ub is equal to ubound[i]. Similarly, the declaration provides a distinct .label value for each element of the x array.

The CONSTRAINT, MAX, MIN, PROBLEM, and VAR statements support a SUFFIXES=() clause that allows values for existing, modifiable suffixes (other than .lb and .ub) to be specified with each declaration. This is the syntax for a SUFFIXES=() clause:

SUFFIXES=(*values* **)**

Each *value* specifies a suffix value. The following forms are allowed:

name **=** *expression*

provides an explicit value for the suffix *name*. In this case the suffix acts like an alias for the *expression* value. When you use this form, the suffix value cannot be modified directly, such as by an assignment statement. Suffixes that are updated by the SOLVE statement, such as .status, cannot be specified using this form.

name **INIT** *expression*
: specifies a default value when the suffix *name* is first used but no other value has been supplied.

The values of the suffixes also come from the solver or from values assigned by assignment, READ DATA, or other statements (see an example in the section "Data Set Input/Output" on page 119). Note that PROC OPTMODEL allows the .lb and .ub suffixes to be assigned after the declaration even when the declaration provides a value with an expression following >= or <=.

You must use suffixes with names of the appropriate type. For example, the .dual suffix cannot be used with the name of an objective. In particular, local dummy parameter names cannot have suffixes.

Table 5.15 shows the names of the available suffixes.

Table 5.15 Suffix Names

Name Type	Suffix	Modifiable	Description
any	.name	No	Name text for any non-dummy symbol
Constraint	.active	No	Active status in the current problem
Constraint	.block	Yes	Block ID for decomposition
Constraint	.body	No	Current constraint body value
Constraint	.dual	No	Dual value from the solver
Constraint	.label	Yes	Label text for the solver
Constraint	.lb	Yes	Current lower bound
Constraint	.status	Yes	Status information from the solver
Constraint	.ub	Yes	Current upper bound
Implicit variable	.sol	No	Current solution value
Objective	.active	No	Active status in the current problem
Objective	.sol	No	Current objective value
Objective	.label	Yes	Label text for the solver
Problem	.active	No	Active status of the problem
Problem	.label	Yes	Label text for the solver
Variable	.active	No	Active status in the current problem
Variable	.direction	Yes	Branching direction for MILP
Variable	.dual	No	Alias for .rc
Variable	.fixed	No	Fixed status
Variable	.label	Yes	Label text for the solver
Variable	.lb	Yes	Lower bound
Variable	.msinit	No	Numeric value at the best starting point reported by the multistart solver
Variable	.priority	Yes	Branching priority for MILP and CLP
Variable	.rc	No	Reduced cost (LP) or gradient of the Lagrangian function
Variable	.relax	Yes	Relaxation of integrality restriction
Variable	.sol	No	Current variable value
Variable	.sol[*i*]	Yes	Saved solution value
Variable	.status	Yes	Status information from the solver
Variable	.ub	Yes	Upper bound

The .sol suffix for a variable, implicit variable, or objective can be used within a declaration to reference the current value of the symbol. It is treated as a constant in such cases. The value is independent of the current problem. When the OPTMODEL procedure processes a SOLVE statement, the value is fixed at the start of the SOLVE statement. The .sol suffix can be followed by a positive integer solution index to refer to a particular solution that the SOLVE statement returns. See the section "Multiple Solutions" on page 152 for more information about accessing multiple solutions from the solver. Each problem tracks saved solution values separately. Outside of declarations, a variable, implicit variable, or objective name with the .sol suffix and no solution index is equivalent to the unsuffixed name.

The .status suffix reports status information from the solver. Currently, only the LP solver provides status information. The .status suffix takes on the same character values that are found in the _STATUS_ variable of the PRIMALOUT and DUALOUT data sets for the OPTLP procedure, including values set by the IIS= option. See the section "Variable and Constraint Status" on page 274 and the section "Irreducible Infeasible Set" on page 275, both in Chapter 7, "The Linear Programming Solver," for more information. For other solvers, the .status values default to a single blank character.

If you choose to modify the .status suffix for a variable or constraint, the assigned suffix value can be a single character or an empty string. The LP solver rejects invalid status characters. Blank or empty strings are treated as new row or column entries for the purpose of "warm starting" the solver.

The .active suffix reports the current activity status for names in the problem. The value is 1 if the element is active or 0 otherwise. A PROBLEM name is considered active if it is the current problem (that is, it was selected by the most recent USE PROBLEM statement). A constraint is considered active if it is included in the current problem and not dropped. An objective is considered active if it is the last objective that is added to the current problem. A variable is considered active if it is included in the current problem, independent of the fixed status.

The .fixed suffix reports the fixed status of a variable. The value is 1 if the variable is fixed using the FIX statement for the current problem or 0 otherwise. Variables that are not included in the current problem are treated as unfixed.

The .msinit suffix reports the numeric value of a variable at the best starting point, as reported by the NLP solver when the MULTISTART option is specified. If the solver does not report a best starting point, then the value is missing. The value is tracked independently for each problem to support multiple subproblems. See the section "Multistart" on page 561 in Chapter 11, "The Nonlinear Programming Solver," for more information.

The .block suffix identifies the subproblem for constraints when used with the METHOD=USER option of the decomposition algorithm. The value must be numeric and is initially assigned a missing value. A constraint with a missing value for the .block suffix is part of the master problem. Otherwise constraints belong to the same subproblem if and only if they have the same .block suffix values. See Chapter 16, "The Decomposition Algorithm," for more information.

The .label suffix represents the text passed to the solver to identify a variable, constraint, or objective. Some solvers can display this label in their output. The maximum text length passed to the solver is controlled by the MAXLABLEN= option. The default text is based on the name in the model, abbreviated to fit within MAXLABLEN. For example, a model variable x[1] would be labeled "x[1]". This label text can be reassigned. The .label suffix value is also used to create MPS labels stored in the output data set for the SAVE MPS and SAVE QPS statements.

The .name suffix represents the name of a symbol as a text string. The .name suffix can be used with any declared name except for local dummy parameters. This suffix is primarily useful when applied to problem

symbols (see the section "Problem Symbols" on page 153), since the .name suffix returns the name of the referenced symbol, not the problem symbol name. The name text is based on the name in the model, abbreviated to fit in 256 characters.

Suffixed names can be used wherever a parameter name is accepted, provided only the value is required. However, you are not allowed to change the value of certain suffixes. Table 5.15 marks these suffixes as not modifiable. Suffixed names that are used as a target in an assignment or READ DATA statement must be modifiable.

The following statements formulate a trivial linear programming problem. The objective value is unbounded, which is reported after the execution of the SOLVE statement. The PRINT statements illustrate the corresponding default auxiliary values. This is shown in Figure 5.58.

```
proc optmodel;
   var x, y;
   min z = x + y;
   con c: x + 2*y <= 3;
   solve;
   print x.lb x.ub x.status x.sol;
   print y.lb y.ub y.status y.sol;
   print c.lb c.ub c.body c.dual;
```

Figure 5.58 Using a Suffix: Retrieving Auxiliary Values

x.LB	x.UB	x.STATUS	x.SOL
-1.7977E+308	1.7977E+308	I	0

y.LB	y.UB	y.STATUS	y.SOL
-1.7977E+308	1.7977E+308	I	0

c.LB	c.UB	c.BODY	c.DUAL
-1.7977E+308	3	0	.

Next, continue to submit the following statements to change the default bounds and solve again. The output is shown in Figure 5.59.

```
x.lb=0;
y.lb=0;
c.lb=1;
solve;
print x.lb x.ub x.status x.sol;
print y.lb y.ub y.status y.sol;
print c.lb c.ub c.body c.dual;
```

Figure 5.59 Using a Suffix: Modifying Auxiliary Values

x.LB	x.UB	x.STATUS	x.SOL
0	1.7977E+308	L	0

y.LB	y.UB	y.STATUS	y.SOL
0	1.7977E+308	B	0.5

Figure 5.59 *continued*

c.LB	c.UB	c.BODY	c.DUAL
1	3	1	0.5

NOTE: Spaces are significant. The form NAME.␣TAG is treated as a SAS format name followed by the tag name, not as a suffixed identifier. The forms NAME.TAG, NAME␣.␣TAG, and NAME␣.TAG (note the location of spaces) are interpreted as suffixed references.

Integer Variable Suffixes

The suffixes .relax, .priority, and .direction are applicable to integer variables.

For an integer variable x, setting x.relax to a nonzero, nonmissing value relaxes the integrality restriction. The value of x.relax is read as either 1 or 0, depending on whether or not integrality is relaxed. This suffix is ignored for noninteger variables.

The value that is contained in x.priority sets the priority of an integer variable x for use with branching in the MILP solver or selection in the CLP solvers. This value can be any nonnegative, nonmissing number. The default value is 0, which indicates default branching priority. Variables with positive .priority values are assigned greater priority than the default. Variables with the highest .priority values are assigned the highest priority. Variables with the same .priority value are assigned the same branching priority.

The value of x.direction assigns a branching direction to an integer variable x. This value should be an integer in the range –1 to 3. A noninteger value in this range is rounded on assignment. The default value is 0. The significance of each integer is found in Table 5.16.

Table 5.16 Branching Directions

Value	Direction
–1	Round down to nearest integer
0	Default
1	Round up to nearest integer
2	Round to nearest integer
3	Round to closest presolved bound

Suppose the solver branches next on an integer variable x whose last LP relaxation solution is 3.3. Suppose also that after passing through the presolver, the lower bound of x is 0 and the upper bound of x is 10. If the value in x.direction is –1 or 2, then the solver sets x to 3 for the next iteration. If the value in x.direction is 1, then the solver sets x to 4. If the value in x.direction is 3, then the solver sets x to 0.

The MPS data set created by the SAVE MPS statement ("SAVE MPS Statement" on page 86) includes a BRANCH section if any nondefault .priority or .direction values have been specified for integer variables.

Dual Values

A dual value is associated with each constraint. To access the dual value of a constraint, use the constraint name followed by the suffix .dual.

For linear programming problems, the dual value associated with a constraint is also known as the dual price (also called the shadow price). The shadow price is usually interpreted economically as the rate at which the optimal value changes with respect to a change in some right-hand side that represents a resource supply or demand requirement.

For nonlinear programming problems, the dual values correspond to the values of the optimal Lagrange multipliers. For more details about duality in nonlinear programming, see Bazaraa, Sherali, and Shetty (1993).

From the dual value associated with the constraint, you can also tell whether the constraint is active or not. A constraint is said to be active (tight at a point) if it holds with equality at that point. It can be informative to identify active constraints at the optimal point and check their corresponding dual values. Relaxing the active constraints might improve the objective value.

Background on Duality in Mathematical Programming

For a minimization problem, there exists an associated problem with the following property: any feasible solution to the associated problem provides a lower bound for the original problem, and conversely any feasible solution to the original problem provides an upper bound for the associated problem. The original and the associated problems are referred to as the primal and the dual problem, respectively. More specifically, consider the primal problem,

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & c_i(x) = 0, \quad i \in \mathcal{E} \\ & c_i(x) \le 0, \quad i \in \mathcal{L} \\ & c_i(x) \ge 0, \quad i \in \mathcal{G} \end{array}$$

where $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{G}$ denote the sets of equality, $\le$ inequality, and $\ge$ inequality constraints, respectively. Variables $x \in \mathbb{R}^n$ are called the primal variables. The Lagrangian function of the primal problem is defined as

$$L(x, \lambda, \mu, \nu) = f(x) - \sum_{i \in \mathcal{E}} \lambda_i c_i(x) - \sum_{i \in \mathcal{L}} \mu_i c_i(x) - \sum_{i \in \mathcal{G}} \nu_i c_i(x)$$

where $\lambda_i \in \mathbb{R}$, $\mu_i \le 0$, and $\nu_i \ge 0$. By convention, the Lagrange multipliers for inequality constraints have to be nonnegative. Hence λ, $-\mu$, and ν correspond to the Lagrange multipliers in the preceding Lagrangian function. It can be seen that the Lagrangian function is a linear combination of the objective function and constraints of the primal problem.

The Lagrangian function plays a fundamental role in nonlinear programming. It is used to define the optimality conditions that characterize a local minimum of the primal problem. It is also used to formulate the dual problem of the preceding primal problem. To this end, consider the following *dual* function:

$$d(\lambda, \mu, \nu) = \inf_{x} \; L(x, \lambda, \mu, \nu)$$

The dual problem is defined as

$$\begin{array}{ll} \underset{\lambda,\mu,\nu}{\text{maximize}} & d(\lambda,\mu,\nu) \\ \text{subject to} & \mu \leq 0 \\ & \nu \geq 0. \end{array}$$

The variables λ, μ, and ν are called the dual variables. Note that the dual variables associated with the equality constraints (λ) are free, whereas those associated with $\leq$ inequality constraints (μ) have to be nonpositive and those associated with $\geq$ inequality constraints (ν) have to be nonnegative.

The relation between the primal and the dual problems provides a nice connection between the optimal solutions of the problems. Suppose x^* is an optimal solution of the primal problem and $(\lambda^*, \mu^*, \nu^*)$ is an optimal solution of the dual problem. The difference between the objective values of the primal and dual problems, $\delta = f(x^*) - d(\lambda^*, \mu^*, \nu^*) \geq 0$, is called the duality gap. For some restricted class of convex nonlinear programming problems, both the primal and the dual problems have an optimal solution and the optimal objective values are equal—that is, the duality gap $\delta = 0$. In such cases, the optimal values of the dual variables correspond to the optimal Lagrange multipliers of the primal problem with the correct signs.

A maximization problem is treated analogously to a minimization problem. For the maximization problem

$$\begin{array}{lll} \underset{x}{\text{maximize}} & f(x) & \\ \text{subject to} & c_i(x) = 0, & i \in \mathcal{E} \\ & c_i(x) \leq 0, & i \in \mathcal{L} \\ & c_i(x) \geq 0, & i \in \mathcal{G}, \end{array}$$

the dual problem is

$$\begin{array}{ll} \underset{\lambda,\mu,\nu}{\text{minimize}} & d(\lambda,\mu,\nu) \\ \text{subject to} & \mu \geq 0 \\ & \nu \leq 0. \end{array}$$

where the dual function is defined as $d(\lambda,\mu,\nu) = \sup_x L(x,\lambda,\mu,\nu)$ and the Lagrangian function $L(x,\lambda,\mu,\nu)$ is defined the same as earlier. In this case, λ, μ, and $-\nu$ correspond to the Lagrange multipliers in $L(x,\lambda,\mu,\nu)$.

Minimization Problems

For inequality constraints in minimization problems, a positive optimal dual value indicates that the associated $\geq$ inequality constraint is active at the solution, and a negative optimal dual value indicates that the associated $\leq$ inequality constraint is active at the solution. In PROC OPTMODEL, the optimal dual value for a *range constraint* (a constraint with both upper and lower bounds) is the sum of the dual values associated with the upper and lower inequalities. Since only one of the two inequalities can be active, the sign of the optimal dual value, if nonzero, identifies which one is active.

For equality constraints in minimization problems, the optimal dual values are unrestricted in sign. A positive optimal dual value for an equality constraint implies that, starting close enough to the primal solution, the same optimum could be found if the equality constraint were replaced with a $\geq$ inequality constraint. A negative optimal dual value for an equality constraint implies that the same optimum could be found if the equality constraint were replaced with a $\leq$ inequality constraint.

The following is an example where simple linear programming is considered:

```
proc optmodel;
   var x, y;
   min z = 6*x + 7*y;
   con
      4*x +   y >=  5,
       -x - 3*y <= -4,
        x +   y <=  4;
   solve;
   print x y;
   expand _ACON_ ;
   print _ACON_.dual _ACON_.body;
```

The PRINT statements generate the output shown in Figure 5.60.

Figure 5.60 Dual Values in Minimization Problem: Display

Problem Summary	
Objective Sense	Minimization
Objective Function	z
Objective Type	Linear
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	3
Linear LE (<=)	2
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	6

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	z
Solution Status	Optimal
Objective Value	13
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	4
Presolve Time	0.00
Solution Time	0.01

Figure 5.60 *continued*

x	y
1	1

```
Constraint _ACON_[1]: 4*x + y >= 5
Constraint _ACON_[2]: - x - 3*y <= -4
Constraint _ACON_[3]: x + y <= 4
```

[1]	_ACON_.DUAL	_ACON_.BODY
1	1	5
2	-2	-4
3	0	2

It can be seen that the first and second constraints are active, with dual values 1 and –2. Continue to submit the following statements. Notice how the objective value is changed in Figure 5.61.

```
_ACON_[1].lb = _ACON_[1].lb - 1;
solve;
_ACON_[2].ub = _ACON_[2].ub + 1;
solve;
```

Figure 5.61 Dual Values in Minimization Problem: Interpretation

Problem Summary	
Objective Sense	Minimization
Objective Function	z
Objective Type	Linear
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	3
Linear LE (<=)	2
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	6

Figure 5.61 *continued*

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	z
Solution Status	Optimal
Objective Value	12
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	4
Presolve Time	0.00
Solution Time	0.01

Problem Summary	
Objective Sense	Minimization
Objective Function	z
Objective Type	Linear
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	3
Linear LE (<=)	2
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	6

Figure 5.61 *continued*

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	z
Solution Status	Optimal
Objective Value	10
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	4
Presolve Time	0.00
Solution Time	0.01

The change is just as the dual values imply. After the first constraint is relaxed by one unit, the objective value is improved by one unit. For the second constraint, the relaxation and improvement are one unit and two units, respectively.

NOTE: The signs of dual values produced by PROC OPTMODEL depend, in some instances, on the way in which the corresponding constraints are entered. See the section "Constraints" on page 131 for details.

Maximization Problems

For inequality constraints in maximization problems, a positive optimal dual value indicates that the associated $\leq$ inequality constraint is active at the solution, and a negative optimal dual value indicates that the associated $\geq$ inequality constraint is active at the solution. The optimal dual value for a range constraint is the sum of the dual values associated with the upper and lower inequalities. The sign of the optimal dual value identifies which inequality is active.

For equality constraints in maximization problems, the optimal dual values are unrestricted in sign. A positive optimal dual value for an equality constraint implies that, starting close enough to the primal solution, the same optimum could be found if the equality constraint were replaced with a $\leq$ inequality constraint. A negative optimal dual value for an equality constraint implies that the same optimum could be found if the equality constraint were replaced with a $\geq$ inequality constraint.

CAUTION: The signs of dual values produced by PROC OPTMODEL depend, in some instances, on the way in which the corresponding constraints are entered. See the section "Constraints" on page 131 for details.

Reduced Costs

In linear programming problems, each variable has a corresponding reduced cost. To access the reduced cost of a variable, add the suffix .rc or .dual to the variable name. These two suffixes are interchangeable.

The reduced cost of a variable is the rate at which the objective value changes when the value of that variable changes. At optimality, basic variables have a reduced cost of zero; a nonbasic variable with zero reduced cost indicates the existence of multiple optimal solutions.

In nonlinear programming problems, the reduced cost interpretation does not apply. The .dual and .rc variable suffixes represent the gradient of the Lagrangian function, computed using the values returned by the solver.

The following example illustrates the use of the .rc suffix:

```
proc optmodel;
   var x >= 0, y >= 0, z >= 0;
   max cost = 4*x + 3*y - 5*z;
   con
      -x + y + 5*z <= 15,
      3*x - 2*y - z <= 12,
      2*x + 4*y + 2*z <= 16;
   solve;
   print x y z;
   print x.rc y.rc z.rc;
```

The PRINT statements generate the output shown in Figure 5.62.

Figure 5.62 Reduced Cost in Maximization Problem: Display

x	y	z
5	1.5	0

x.RC	y.RC	z.RC
0	0	-6.5

In this example, x and y are basic variables, while z is nonbasic. The reduced cost of z is –6.5, which implies that increasing z from 0 to 1 decreases the optimal value from 24.5 to 18.

Presolver

PROC OPTMODEL includes a simple presolver that processes linear constraints to produce tighter bounds on variables. The presolver can reduce the number of variables and constraints that are presented to the solver. These changes can result in reduced solution times.

Linear constraints that involve only a single variable are converted into variable bounds. The presolver then eliminates redundant linear constraints for which variable bounds force the constraint to always be satisfied. Tightly bounded variables where upper and lower bounds are within the range specified by the VARFUZZ= option (see the section "PROC OPTMODEL Statement" on page 38) are automatically fixed to the average of the bounds. The presolver also eliminates variables that are fixed by the user or by the presolver.

The presolver can infer tighter variable bounds from linear constraints when all variables in the constraint or all but one variable have known bounds. For example, when given the following PROC OPTMODEL declarations, the presolver can use the inferred bounds $x \leq 7$ and $y \leq 4$ to remove the constraint c2:

```
proc optmodel;
   var x >= 3;
   var y >= 0;
   con c: x + y <= 7;
   con c2: x + 2*y <= 15; /* redundant */
```

The presolver makes multiple passes and rechecks linear constraints after bounds are tightened for the referenced variables. The number of passes is controlled by the PRESOLVER= option. After the passes are finished, the presolver attempts to fix the value of all variables that are not used in the updated objective and constraints. The current value of such a variable is used if the value lies between the variable's upper and lower bounds. Otherwise, the value is adjusted to the nearer bound. The value of an integer variable is rounded before being checked against its bounds.

The presolver uses the tightened variable bounds to infer the range of values for nonlinear constraint bodies. In some cases, the presolver can determine that a nonlinear constraint is infeasible. The presolver can also remove a constraint that does not restrict the feasible region of the problem.

In some cases the solver might perform better without the presolve transformations, so almost all such transformations are unavailable when the option PRESOLVER=BASIC is specified. However, the presolver still eliminates variables that have values that have been fixed by the FIX statement. To disable the OPTMODEL presolver entirely, use PRESOLVER=NONE. The solver assigns values to any unused, unfixed variables when the option PRESOLVER=NONE is specified.

Model Update

The PROC OPTMODEL modeling language provides several means of modifying a model after it is first specified. You can change the parameter values of the model. You can add new model components. The FIX and UNFIX statements can fix variables to specified values or rescind previously fixed values. The DROP and RESTORE statements can deactivate and reactivate constraints. See also the section "Multiple Subproblems" on page 151 for information on how to maintain multiple models.

To illustrate how these statements work, reconsider the following example from the section "Constraints" on page 131:

```
proc optmodel;
   var x, y;
   min r = x**2 + y**2;
   con c: x+y >= 1;
   solve;
   print x y;
```

As described previously, the solver finds the optimal point x = y = 0.5 with r = 0.5. You can see the effect of the constraint c on the solution by temporarily removing it. You can add the following statements:

```
   drop c;
   solve;
   print x y;
```

This change produces the output in Figure 5.63.

Figure 5.63 Solution with Dropped Constraint

Problem Summary	
Objective Sense	Minimization
Objective Function	r
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	0
Constraint Coefficients	0
Hessian Diagonal Elements	2
Hessian Elements Below Diagonal	0

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	r
Solution Status	Optimal
Objective Value	0
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Duality Gap	0
Complementarity	0
Iterations	0
Presolve Time	0.00
Solution Time	0.00

x	y
0	0

The optimal point is $x = y = 0$, as expected.

You can restore the constraint c with the RESTORE statement, and you can also investigate the effect of forcing the value of variable x to 0.3. This requires the following statements:

```
restore c;
fix x=0.3;
solve;
print x y c.dual;
```

This produces the output in Figure 5.64.

Figure 5.64 Solution with Fixed Variable

Problem Summary	
Objective Sense	Minimization
Objective Function	r
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	1
Fixed	1
Number of Constraints	1
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	2
Hessian Diagonal Elements	1
Hessian Elements Below Diagonal	0

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	r
Solution Status	Optimal
Objective Value	0.58
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Duality Gap	0
Complementarity	0
Iterations	0
Presolve Time	0.00
Solution Time	0.00

x	y	c.DUAL
0.3	0.7	1.4

The variable x still has the value that was defined in the FIX statement. The objective value has increased by 0.08 from its constrained optimum 0.5 (see Figure 5.55). The constraint c is active, as confirmed by the positive dual value.

You can return to the original optimization problem by allowing the solver to vary variable x with the UNFIX statement, as follows:

```
unfix x;
solve;
print x y c.dual;
```

This produces the output in Figure 5.65. The model was returned to its original conditions.

Figure 5.65 Solution with Original Model

Problem Summary	
Objective Sense	Minimization
Objective Function	r
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	1
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	2
Hessian Diagonal Elements	2
Hessian Elements Below Diagonal	0

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	r
Solution Status	Optimal
Objective Value	0.5
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Duality Gap	0
Complementarity	0
Iterations	4
Presolve Time	0.00
Solution Time	0.01

Figure 5.65 *continued*

x	y	c.DUAL
0.5	0.5	1

Multiple Subproblems

The OPTMODEL procedure enables multiple models to be manipulated easily by using named problems to switch the active model components. Problems keep track of an objective, a set of included variables and constraints, and some status information that is associated with the variables and constraints. Other data, such as parameter values, bounds, and the current value of variables, are shared by all problems.

Problems are declared with the PROBLEM declaration. You can easily switch between problems by using the USE PROBLEM statement. The USE PROBLEM statement makes the specified problem become the current problem. The various statements that generate problem data, such as SOLVE, EXPAND, and SAVE MPS, always operate using the model components included in the current problem.

A problem declaration can specify the problem's objectives by copying them from the problem that is named in a FROM clause or by including objective symbols. The objectives can be updated while the problem is current by declaring a new non-array objective or by executing programming statements that specify new objectives.

Variables can also be included when the problem is current by declaring them or by using the FIX or UNFIX statement. Similarly, constraints can be included when the problem is current by declaring them or by using the RESTORE or DROP statement. There is no way to exclude a variable or constraint item after it has been included in a problem, although the variable or constraint can be fixed or dropped.

Variables that are declared but not included in a problem are treated as constants when a problem is generated, while constraints that are declared but not included are ignored. The solver does not update the values and status for these model components.

A problem also tracks certain other status information that is associated with its included symbols, and this information can be changed without affecting other problems. This information includes the fixed status for variables, and the dropped status for constraints. The following additional data that are tracked by the problem are available through variable and constraint suffixes:

- *var*.STATUS (including IIS status)
- *var*.SOL[*i*] (for each solution *i*)
- *var*.MSINIT
- *var*.RC
- *var*.DUAL (alias of *var*.RC)
- *var*.FIXED
- *con*.STATUS (including IIS status)
- *con*.DUAL

- *con*.BLOCK

The initial problem when OPTMODEL starts is predeclared with the name _START_. This problem can be reinstated again (after other USE PROBLEM statements) with the statement

```
use problem _start_;
```

See "Example 5.5: Multiple Subproblems" on page 175 for example statements that use multiple subproblems.

Multiple Solutions

When a solver finishes, it reports zero or more solutions for the optimization variables of the current problem. Each solution assigns a value to each of the variables in the problem. The SOLVE statement saves these solutions with the current problem. The first reported solution, if any, is also copied into the optimization variables. The number of solutions is available in the predeclared numeric parameter _NSOL_.

NOTE: The network solver does not require optimization variables and has its own conventions for returning multiple solutions.

You can access the solutions that are saved with the problem by adding a solution index to the .sol suffix of the variable name. For example, x.sol[2] would reference the second solution saved for variable x in the current problem. Both the variable name and the suffix can be indexed. For example, assign[3,7].sol[1] refers to the first solution for the array variable element assign[3,7]. The solution index must be an integer in the range 1 to _NSOL_.

The following example illustrates the processing of multiple solutions from the CLP solver:

```
proc optmodel printlevel=0;
   var x{1..2} integer >= 1 <= 3;
   con c: alldiff(x);
   solve with clp / allsolns;
   print _NSOL_;
   print {j in 1..2, i in 1.._NSOL_} x[j].sol[i];
   create data solout from [sol]={i in 1.._NSOL_}
       {j in 1..2} <col("x"||j)=x[j].sol[i]> ;
```

This program produces the output in Figure 5.66. It also creates a data set, solout, which has each solution in a separate observation.

Figure 5.66 Processing Multiple Solutions

NSOL
6

	x.SOL					
	1	2	3	4	5	6
1	1	1	2	2	3	3
2	2	3	1	3	1	2

Problem Symbols

The OPTMODEL procedure declares a number of symbols that are aliases for model components in the current problem. These symbols allow the model components to be accessed uniformly. These symbols are described in Table 5.17.

Table 5.17 Problem Symbols

Symbol	Indexing	Description
NVAR		Number of variables
VAR	{1.._NVAR_}	Variable array
NCON		Number of constraints
CON	{1.._NCON_}	Constraint array
_S_NVAR_		Number of presolved variables
_S_VAR_	{1.._S_VAR_}	Presolved variable array
_S_NCON_		Number of presolved constraints
_S_CON_	{1.._S_CON_}	Presolved constraint array
NOBJ		Number of objectives
ALLOBJ	{1.._NOBJ_}	Objective array
OBJ		Current objective
PROBLEM		Current problem

If the table specifies indexing, then the corresponding symbol is accessed as an array. For example, if the problem includes two variables, x and y, then the value of _NVAR_ is 2 and the current variable values can be accessed as _var_[1] and _var_[2]. The problem variables prefixed with _S are restricted to model components in the problem after processing by the OPTMODEL presolver.

The following statements define a simple linear programming model and then use the problem symbols to print out some of the problem results. The .name suffix is used in the PRINT statements to display the actual variable and constraint names. Any of the suffixes that apply to a model component can be applied to the corresponding generic symbol.

```
proc optmodel printlevel=0;
   var x1 >= 0, x2 >= 0, x3 >= 0, x4 >= 0, x5 >= 0;

   minimize z = x1 + x2 + x3 + x4;

   con a1: x1 + x2 + x3          <= 4;
   con a2:               x4 + x5 <= 6;
   con a3: x1 +          x4      >= 5;
   con a4:      x2 +          x5 >= 2;
   con a5:           x3          >= 3;

   solve with lp;

   print _var_.name _var_ _var_.rc _var_.status;
   print _con_.name _con_.lb _con_.body _con_.ub _con_.dual _con_.status;
```

The PRINT statement output is shown in Figure 5.67.

Figure 5.67 Problem Symbol Output

[1]	_VAR_.NAME	_VAR_	_VAR_.RC	_VAR_.STATUS
1	x1	1	0	B
2	x2	0	1	L
3	x3	3	0	B
4	x4	4	0	B
5	x5	2	0	B

[1]	_CON_.NAME	_CON_.LB	_CON_.BODY	_CON_.UB	_CON_.DUAL	_CON_.STATUS
1	a1	-1.7977E308	4	4.0000E+00	0	L
2	a2	-1.7977E308	6	6.0000E+00	0	B
3	a3	5	5	1.7977E+308	1	U
4	a4	2	2	1.7977E+308	0	U
5	a5	3	3	1.7977E+308	1	U

OPTMODEL Options

All PROC OPTMODEL options can be specified in the PROC statement (see the section "PROC OPTMODEL Statement" on page 38 for more information). However, it is sometimes necessary to change options after other PROC OPTMODEL statements have been executed. For example, if an optimization technique had trouble with convergence, then it might be useful to vary the PRESOLVER= option value. This can be done with the RESET OPTIONS statement.

The RESET OPTIONS statement accepts options in the same form used by the PROC OPTMODEL statement. The RESET OPTIONS statement is also able to reset option values and to change options programmatically. For example, the following statements print the value of parameter n at various precisions:

```
proc optmodel;
   number n = 1/7;
   for {i in 1..9 by 4}
   do;
      reset options pdigits=(i);
      print n;
   end;
   reset options pdigits; /* reset to default */
```

The output generated is shown in Figure 5.68. The RESET OPTIONS statement in the DO loop sets the PDIGITS option to the value of i. The final RESET OPTIONS statement restores the default option value, because the value was omitted.

Figure 5.68 Changing the PDIGITS Option Value

n
0.1

n
0.14286

Figure 5.68 *continued*

n
0.142857143

Automatic Differentiation

PROC OPTMODEL automatically generates statements to evaluate the derivatives for most objective expressions and nonlinear constraints. PROC OPTMODEL generates analytic derivatives for objective and constraint expressions written in terms of the procedure's mathematical operators and most standard SAS library functions.

NOTE: Some functions, such as ABS, FLOOR, and SIGN, and some operators, such as IF-THEN, <> (maximum operator), and >< (minimum operator), must be used carefully in modeling expressions because functions that include such components are not continuously differentiable or even continuous.

Expressions that reference user-defined functions or some SAS library functions might require numerical approximation of derivatives. PROC OPTMODEL uses either forward-difference approximation or central-difference approximation as specified by the FD= option (see the section "PROC OPTMODEL Statement" on page 38).

NOTE: The numerical gradient approximations are significantly slower than automatically generated derivatives when the number of optimization variables is large.

Forward-Difference Approximations

The FD=FORWARD option requests the use of forward-difference derivative approximations. For a function f of n variables, the first-order derivatives are approximated by

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + e_i h_i) - f(x)}{h_i}$$

Notice that up to n additional function calls are needed here. The step lengths $h_i, i = 1, \ldots, n$, are based on the assumed function precision, *DIGITS*:

$$h_i = 10^{-DIGITS/2}(1 + |x_i|)$$

You can use the FDIGITS= option to specify the function precision, *DIGITS*, for the objective function. For constraints, use the CDIGITS= option.

The second-order derivatives are approximated by using up to $n(n + 3)/2$ extra function calls (Dennis and Schnabel 1983, pp. 80, 104):

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{f(x + h_i e_i) - 2f(x) + f(x - h_i e_i)}{h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_i h_j}$$

Notice that the diagonal of the Hessian uses a central-difference approximation (Abramowitz and Stegun 1972, p. 884). The step lengths are

$$h_i = 10^{-DIGITS/3}(1 + |x_i|)$$

Central-Difference Approximations

The FD=CENTRAL option requests the use of central-difference derivative approximations. Generally, central-difference approximations are more accurate than forward-difference approximations, but they require more function evaluations. For a function f of n variables, the first-order derivatives are approximated by

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + e_i h_i) - f(x - e_i h_i)}{2h_i}$$

Notice that up to $2n$ additional function calls are needed here. The step lengths h_i, $i = 1, \ldots, n$, are based on the assumed function precision, *DIGITS*:

$$h_i = 10^{-DIGITS/3}(1 + |x_i|)$$

You can use the FDIGITS= option to specify the function precision, *DIGITS*, for the objective function. For constraints, use the CDIGITS= option.

The second-order derivatives are approximated by using up to $2n(n + 1)$ extra function calls (Abramowitz and Stegun 1972, p. 884):

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{-f(x + 2h_i e_i) + 16f(x + h_i e_i) - 30f(x) + 16f(x - h_i e_i) - f(x - 2h_i e_i)}{12h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i - h_j e_j) - f(x - h_i e_i + h_j e_j) + f(x - h_i e_i - h_j e_j)}{4h_i h_j}$$

The step lengths are

$$h_i = 10^{-DIGITS/3}(1 + |x_i|)$$

Conversions

Numeric values are implicitly converted to strings when needed for function arguments or operands to the string concatenation operator (||). A warning message is generated when the conversion is applied to a function argument. The conversion uses BEST12. format. Unlike the DATA step, the conversion trims blanks.

Implicit conversion of strings to numbers is not permitted. Use the INPUT function to explicitly perform such conversions.

FCMP Routines

The OPTMODEL procedure can call functions and subroutines that are compiled by the FCMP procedure. You can use FCMP functions wherever a function expression is allowed in PROC OPTMODEL. Use the CALL statement to call FCMP subroutines. The following example defines a function in the FCMP procedure and calls it within PROC OPTMODEL:

```
proc fcmp outlib=work.funcs.test;
   /* arithmetic geometric mean */
   function agm(a0, b0);
      a = a0; b = b0;
      if a<=0 or b<=0 then return(0);
      do until( a - b < a/1e12 );
         a1 = 0.5*a + 0.5*b;
         b1 = sqrt(a*b);
         a = a1; b = b1;
      end;
      return(a);
   endsub;
run;

/* libraries must be specified with the CMPLIB option */
option cmplib=work.funcs;

proc optmodel;
   print (agm(1,2));

   /* find x where agm(1,x) == 23 */
   var x init 1;
   num c = 23;
   min z = (agm(1,x)-c)^2;
   solve;
   print x;
```

FCMP subroutines can return data by updating OPTMODEL numeric and string parameters, which are passed as arguments in a CALL statement. These arguments are declared using the OUTARGS statement in the PROC FCMP subroutine definition. The OPTMODEL argument must be specified with an identifier expression. The following code shows a simple example of output arguments. The maximum length of output strings from OUTARGS arguments is restricted to the argument length before the call, as described in the section "CALL Statement" on page 53.

```
proc fcmp outlib=work.funcs.test;
   subroutine do_sqr(x, sq, text $);
      outargs sq, text;
      sq = x*x;
      text = 'This is an example of output arguments';
   endsub;
run;

option cmplib=work.funcs;

proc optmodel;
   string s init repeat(' ', 79); /* reserve 80 bytes */
   number n;
   call do_sqr(7, n, s);
   print s n;
```

This code produces the output in Figure 5.69.

Figure 5.69 FCMP Output Arguments

s	n
This is an example of output arguments	49

You can pass OPTMODEL arrays to FCMP functions and subroutines that accept matrix arguments. The array must match the type and dimensions of the FCMP argument declaration. The argument in the OPTMODEL CALL statement must be specified using the following syntax:

array-name [**.** *suffix*]

The following code passes a constant matrix to an FCMP function. The array coeff contains the coefficients of a polynomial, which in this case defines a simple quadratic formula, $x^2 - 2x + 1$.

```
proc fcmp outlib=work.funcs.test;
   function evalpoly(x, coeff[*]);
      z = 0;
      do i = dim1(coeff) to 1 by -1;
         z = z * x + coeff[i];
      end;
      return (z);
   endsub;
run;

option cmplib=work.funcs;

proc optmodel;
   num coeff{1..3} = [1, -2, 1];
   var x;
   min z=evalpoly(x, coeff);
   solve;
   print x;
```

An array that is used as a matrix argument must be indexed like an FCMP matrix. In other words, the array index set must be specified as the crossproduct of one or more range expressions (such as 1..N) where the

lower bound and step size are 1. Set parameters that are used for indexing must contain a crossproduct of ranges, but the element order is not important. The following code shows some examples of suitable and unsuitable array declarations:

```
proc fcmp outlib=work.funcs.test;
   subroutine mattest(x[*]);
      put x[1]=;
   endsub;
   subroutine mattest2(x[*,*]);
      put x[1,1]=;
   endsub;
run;

option cmplib=work.funcs;

proc optmodel;
   /* the following arrays can be used as matrices */
   num N init 3;
   num mat1{1..N} init 0;
   call mattest(mat1);     /* OK */
   set S1 = 1..5;
   num mat2{S1} init 0;
   call mattest(mat2);     /* OK */
   set S2 = {S1,S1};
   num mat3{S2} init 0;
   call mattest2(mat3);    /* OK */
   num mat4{S1 cross S1} init 0;
   call mattest2(mat4);    /* OK */
   num L init 1;
   num mat5{L..N} init 0;
   call mattest(mat5);     /* OK */
   set S3 init S1;
   num mat6{S3} init 0;
   call mattest(mat6);     /* OK */

   /* some errors are detected at execution time */
   S3 = 2..5;
   call mattest(mat6);     /* ERROR: lower bound not 1 */
   S3 = {1, 3, 4, 5};
   call mattest(mat6);     /* ERROR: missing index 2 */
   L = 0;
   call mattest(mat5);     /* ERROR: lower bound not 1 */

   /* the following arrays cannot be used as matrices */
   num arr1{1..10 by 3};   /* step size is not 1 */
   call mattest(arr1);     /* ERROR */
   num arr2{i in 1..N, j in 1..N: j >= i}; /* selection expression used */
   call mattest2(arr2);    /* ERROR */
   num arr3{i in 1..N, j in 1..i};         /* index dependency on 'i' */
   call mattest2(arr3);    /* ERROR */
```

Not all PROC FCMP functionality is compatible with PROC OPTMODEL; in particular, the following FCMP functions are not supported and should not be called within your FCMP function definitions: READ_ARRAY, WRITE_ARRAY, RUN_MACRO, and RUN_SASFILE. In many cases, OPTMODEL capabilities can replace

these functions. Matrix arguments can be used in place of the READ_ARRAY function by using the READ DATA statement to load the matrix in PROC OPTMODEL. Similarly, you can replace the WRITE_ARRAY function in an FCMP subroutine by copying the matrix to an output argument and using the OPTMODEL procedure to write the matrix. You can use the SUBMIT statement in place of the RUN_MACRO and RUN_SASFILE functions.

The SAS CMPLIB= system option specifies where to look for previously compiled functions and subroutines. For more information about the CMPLIB= system option, see *SAS System Options: Reference*. FCMP functions can be used in distributed mode with the NLP multistart or LSO solver. The needed PROC FCMP compiled routines are automatically packaged and distributed. For more information about the multistart solver, see Chapter 11, "The Nonlinear Programming Solver," in this book.

PROC OPTMODEL uses derivatives values that are provided by FCMP when they are available. FCMP cannot provide derivatives with respect to array arguments, so PROC OPTMODEL must use finite differences to compute these derivatives. Also, if the CMPOPT= SAS system option specifies the FUNCDIFFERENCING value, then PROC OPTMODEL uses its own finite differencing for FCMP functions.

More on Index Sets

Dummy parameters behave like parameters but are assigned values only when an index set is evaluated. You can reference the declared dummy parameters from index set expressions that follow the index set item. You can also reference the dummy parameters in the expression or statement controlled by the index set. As the members of the set expression of an index set item are enumerated, the element values of the members are assigned to the local dummy parameters.

The number of names in a dummy parameter declaration must match the element length of the corresponding set expression in the index set item. A single name is allowed when the set member type is scalar (numeric or string). If the set members are tuples that have $n > 1$ elements, then n names are required between the angle brackets (< >) that precede the IN keyword.

Multiple index set items in an index set are nominally processed in a left-to-right order. That is, a set expression from an index set item is evaluated as though the index set items that precede it have already been evaluated. The left-hand index set items can assign values to local dummy parameters that are used by the set expressions that follow them. After each member from the set expression is enumerated, any index set items to the right are reevaluated as needed. The actual order in which index set items are evaluated can vary, if necessary, to allow more efficient enumeration. PROC OPTMODEL generates the same set of values in any case, although possibly in a different order than strict left-to-right evaluation.

You can view the element combinations that are generated from an index set as tuples. This is especially true for index set expressions (see the section "Index Set Expression" on page 108). However, in most cases no tuple set is actually formed, and the element values are assigned only to local dummy parameters.

You can specify a selection expression following a colon (:). The index set generates only those combinations of values for which the selection expression is true. For example, the following statements produce a set of upper triangular indices:

```
proc optmodel;
   put (setof {i in 1..3, j in 1..3 : j >= i} <i, j>);
```

These statements produce the output in Figure 5.70.

Figure 5.70 Upper Triangular Index Set

```
{<1,1>,<1,2>,<1,3>,<2,2>,<2,3>,<3,3>}
```

You can use the left-to-right evaluation of index set items to express the previous set more compactly. The following statements produce the same output as the previous statements:

```
proc optmodel;
   put ({i in 1..3, i..3});
```

In this example, the first time the second index set item is evaluated, the value of the dummy parameter i is 1, so the item produces the set {1,2,3}. At the second evaluation the value of i is 2, so the second item produces the set {2,3}. At the final evaluation the value of i is 3, so the second item produces the set {3}.

In many cases it is useful to combine the SLICE operator with index sets. A special form of index set item uses the SLICE operator implicitly. Normally an index set item that is applied to a set of tuples of length greater than one must be of the form

< *name-1* [**,** . . . *name-n*] **>** **IN** *set-expression*

In the special form, one or more of the name elements are replaced by expressions. The expressions select tuple elements by using the SLICE operator. An expression that consists of a single name must be enclosed in parentheses to distinguish it from a dummy parameter. The remaining names are the dummy parameters for the index set item that is applied to the SLICE result. The following example demonstrates the use of implicit set slicing:

```
proc optmodel;
   number N = 3;
   set<num,str> S = {<1,'a'>,<2,'b'>,<3,'a'>,<4,'b'>};
   put ({i in 1..N, <(i),j> in S});
   put ({i in 1..N, j in slice(<i,*>, S)});
```

The two PUT statements in this example are equivalent.

Threaded and Distributed Processing

The OPTMODEL procedure can take advantage of the multiple CPUs that are available in many computers. PROC OPTMODEL automatically uses multithreaded execution to divide problem generation among the multiple CPUs of the computer that is running the procedure. Hessian and Jacobian matrix evaluation is automatically parallelized across threads of execution on multiple CPUs. The COFOR statement enables solvers to concurrently execute in background threads on multiple CPUs, overlapping with PROC OPTMODEL statement processing. Parallel execution can decrease the amount of clock time required to perform a task, although the total CPU time required might increase.

When you do not request distributed computing, threaded processing is performed on the client. The NTHREADS= option controls the number of threads that are used for processing. When you do not specify the NTHREADS= option, threading in the OPTMODEL procedure is controlled by the following SAS system options:

CPUCOUNT=*number* | **ACTUAL**
specifies the maximum number of CPUs that can be used.

THREADS | **NOTHREADS**
enables or disables the use of threading.

Good performance is usually obtained with the default option settings (THREADS and CPUCOUNT=ACTUAL). See the option descriptions in *SAS System Options: Reference* for more details.

When distributed computing is used, the NTHREADS= option is ignored by problem generation because it is performed on the client. In this case, the maximum number of threads that are used for problem generation is based on the SAS CPUCOUNT= and THREADS options.

The NTHREADS= option and the SAS system options set the maximum number of threads. The number of threads that PROC OPTMODEL actually uses depends on the characteristics of the problem that is being solved and the requirements of the distributed computing environment. In particular, threading is not used when the problem is simple enough that threading offers no advantage.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure creates a macro variable named _OROPTMODEL_. You can inspect the execution of the most recently invoked solver from the value of the macro variable. The macro variable is defined at the start of the procedure and updated after each SOLVE statement is executed. The OPTMODEL procedure also updates the macro variable when an error is detected.

The _OROPTMODEL_ value is a string that consists of several "KEYWORD=*value*" items in sequence, separated by blanks; for example:

```
STATUS=OK ALGORITHM=DS SOLUTION_STATUS=OPTIMAL OBJECTIVE=119302.04331
PRIMAL_INFEASIBILITY=3.552714E-13 DUAL_INFEASIBILITY=2.273737E-13
BOUND_INFEASIBILITY=0 ITERATIONS=82 PRESOLVE_TIME=0.02 SOLUTION_TIME=0.05
```

The information contained in _OROPTMODEL_ varies according to which solver was last called. For lists of keywords and possible values, see the individual solver chapters.

If a value has not been computed, then the corresponding element is not included in the value of the macro variable. When PROC OPTMODEL starts, for example, the macro variable value is set to "STATUS=OK" because no SOLVE statement has been executed. If the STATUS= indicates an error, then the other values from the solver might not be available, depending on when the error occurred.

Solver Status Parameters

In addition to creating the macro variable _OROPTMODEL_, the OPTMODEL procedure creates several predeclared parameters to provide simple access to solver status values. These parameters are declared as follows:

```
string _STATUS_;
string _SOLUTION_STATUS_;
set<string> _OROPTMODEL_STR_KEYS_;
set<string> _OROPTMODEL_NUM_KEYS_;
string _OROPTMODEL_STR_{_OROPTMODEL_STR_KEYS_};
```

```
number _OROPTMODEL_NUM_{_OROPTMODEL_NUM_KEYS_};
```

The value of _STATUS_ is equal to the STATUS= component of the _OROPTMODEL_ macro variable. The value of _STATUS_ is initially "OK". The value is updated during the SOLVE statement and after statement execution errors.

The value of _SOLUTION_STATUS_ is equal to the SOLUTION_STATUS= component of the _OROPTMODEL_ macro variable. The value is initially an empty string. The value is updated during the SOLVE statement.

You can use the remaining status parameters to access all the components of the _OROPTMODEL_ macro variable. The following statements demonstrate these parameters:

```
proc optmodel printlevel=0;
   var x init 1 >= 0.001;
   min z=sin(x)/x;
   solve;
   for {k in /STATUS SOLUTION_STATUS ALGORITHM/}
      put _OROPTMODEL_STR_[k]=;
   for {k in /OBJECTIVE ITERATIONS/}
      put _OROPTMODEL_NUM_[k]=;
```

These statements produce the output in Figure 5.71.

Figure 5.71 Solver Status Parameters

```
_OROPTMODEL_STR_[STATUS]=OK
_OROPTMODEL_STR_[SOLUTION_STATUS]=OPTIMAL
_OROPTMODEL_STR_[ALGORITHM]=IP
_OROPTMODEL_NUM_[OBJECTIVE]=-0.217233628
_OROPTMODEL_NUM_[ITERATIONS]=3
```

The _OROPTMODEL_STR_ array contains the same character component values that are found in the _OROPTMODEL_ macro variable. Specify the component name as the array index. For example, the indices "STATUS" and "SOLUTION_STATUS" select array elements that hold the STATUS= and SOLUTION_STATUS= component values, respectively. The set _OROPTMODEL_STR_KEYS_ contains the component indices that you can use with _OROPTMODEL_STR_. The OPTMODEL procedure updates the _OROPTMODEL_STR_ and _OROPTMODEL_STR_KEYS_ parameters during the execution of the SOLVE statement and after any execution errors occur.

The _OROPTMODEL_NUM_ array contains the numeric component values that are displayed in the _OROPTMODEL_ macro variable. For example, the index "OBJECTIVE" selects the array element that holds the objective value when a solution is available. The set _OROPTMODEL_NUM_KEYS_ contains the component indices that you can use with _OROPTMODEL_NUM_. The OPTMODEL procedure updates the _OROPTMODEL_NUM_ and _OROPTMODEL_NUM_KEYS_ parameters during the execution of the SOLVE statement and after any execution errors occur.

Macro and Statement Evaluation Order

PROC OPTMODEL reads a complete statement, such as a DO statement, before executing any code in it. But macro language statements are processed as the code is read. So you must be careful when using the _OROPTMODEL_ macro variable in code that involves SOLVE statements nested in loops or DO statements. The following statements demonstrate one example of this behavior:

```
proc optmodel;
   var x, y;
   min z=x**2 + (x*y-1)**2;
   for {n in 1..3} do;
      fix x=n;
      solve;
      %put Line 1 &_OROPTMODEL_;
      put 'Line 2 ' (symget("_OROPTMODEL_"));
   end;
quit;
```

In the preceding statements the %PUT statement is executed once, before any SOLVE statements are executed. It displays PROC OPTMODEL's initial setting of the macro variable. But the PUT statement is executed after each SOLVE statement and indicates the expected solution status.

Rewriting PROC NLP Models for PROC OPTMODEL

This section describes techniques for converting PROC NLP models to PROC OPTMODEL models. Example 5.8 also demonstrates how to rewrite a PROC NLP model for use with PROC OPTMODEL.

To illustrate the basics, consider the following first version of the PROC NLP model for the example "Simple Pooling Problem" in Chapter 6, "The NLP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*):

```
proc nlp all;
   parms amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools = 1;
   bounds 0 <= amountx amounty amounta amountb amountc,
               amountx <= 100,
               amounty <= 200,
          0 <= pooltox pooltoy ctox ctoy,
          1 <= pools <= 3;
   lincon amounta + amountb = pooltox + pooltoy,
          pooltox + ctox = amountx,
          pooltoy + ctoy = amounty,
          ctox + ctoy    = amountc;
   nlincon nlc1-nlc2 >= 0.,
           nlc3 = 0.;
   max f;
   costa = 6; costb = 16; costc = 10;
   costx = 9; costy = 15;
   f = costx * amountx + costy * amounty
       - costa * amounta - costb * amountb - costc * amountc;
   nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
   nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
   nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
run;
```

These statements define a model that has bounds, linear constraints, nonlinear constraints, and a simple objective function. The following statements are a straightforward conversion of the PROC NLP statements to PROC OPTMODEL form:

```
proc optmodel;
   var amountx init 1 >= 0 <= 100,
       amounty init 1 >= 0 <= 200;
   var amounta init 1 >= 0,
       amountb init 1 >= 0,
       amountc init 1 >= 0;
   var pooltox init 1 >= 0,
       pooltoy init 1 >= 0;
   var ctox init 1 >= 0,
       ctoy init 1 >= 0;
   var pools init 1 >=1 <= 3;
   con amounta + amountb = pooltox + pooltoy,
       pooltox + ctox = amountx,
       pooltoy + ctoy = amounty,
       ctox + ctoy    = amountc;
   number costa, costb, costc, costx, costy;
   costa = 6; costb = 16; costc = 10;
   costx = 9; costy = 15;
   max f = costx * amountx + costy * amounty
         - costa * amounta - costb * amountb - costc * amountc;
   con nlc1: 2.5 * amountx - pools * pooltox - 2. * ctox >= 0,
       nlc2: 1.5 * amounty - pools * pooltoy - 2. * ctoy >= 0,
       nlc3: 3 * amounta + amountb - pools * (amounta + amountb)
              = 0;
   solve;
   print amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools;
```

The PROC OPTMODEL variable declarations are split into individual declarations because PROC OPTMODEL does not permit name lists in its declarations. In the OPTMODEL procedure, you specify variable bounds as part of the variable declaration instead of in a separate BOUNDS statement. The PROC NLP statements are as follows:

```
parms amountx amounty amounta amountb amountc
      pooltox pooltoy ctox ctoy pools = 1;
bounds 0 <= amountx amounty amounta amountb amountc,
            amountx <= 100,
            amounty <= 200,
       0 <= pooltox pooltoy ctox ctoy,
       1 <= pools <= 3;
```

The following PROC OPTMODEL statements are equivalent to the preceding PROC NLP statements:

```
var amountx init 1 >= 0 <= 100,
    amounty init 1 >= 0 <= 200;
var amounta init 1 >= 0,
    amountb init 1 >= 0,
    amountc init 1 >= 0;
var pooltox init 1 >= 0,
    pooltoy init 1 >= 0;
var ctox init 1 >= 0,
    ctoy init 1 >= 0;
var pools init 1 >= 1 <= 3;
```

The linear constraints are declared in the PROC NLP model by using the following statement:

```
lincon amounta + amountb = pooltox + pooltoy,
       pooltox + ctox = amountx,
       pooltoy + ctoy = amounty,
       ctox + ctoy    = amountc;
```

The following linear constraint declarations in the PROC OPTMODEL model are quite similar to the PROC NLP LINCON declarations:

```
con amounta + amountb = pooltox + pooltoy,
    pooltox + ctox = amountx,
    pooltoy + ctoy = amounty,
    ctox + ctoy    = amountc;
```

But PROC OPTMODEL provides much more flexibility in defining linear constraints. For example, a coefficient can be a named parameter or any other expression that evaluates to a constant.

The cost parameters are declared explicitly in the PROC OPTMODEL model. Unlike the DATA step or the NLP procedure, PROC OPTMODEL requires names to be declared before they are used. There are multiple ways to set the values of these parameters. The preceding example uses assignments. You could make the values part of the declaration by using the INIT *expression* clause or the = *expression* clause. You could also read the values from a data set by using the READ DATA statement.

In the original PROC NLP statements, the assignment to a parameter such as costa occurs every time the objective function is evaluated. However, the assignment occurs just once in the PROC OPTMODEL statements, when the assignment statement is processed. This works because the values are constant. But the PROC OPTMODEL statements permit the parameters to be reassigned later so that you can interactively modify the model.

The following statements define the objective f in the PROC NLP model:

```
max f;
. . .
f = costx * amountx + costy * amounty
    - costa * amounta - costb * amountb - costc * amountc;
```

The PROC OPTMODEL version of the objective is defined by using the same expression text, as follows:

```
max f = costx * amountx + costy * amounty
      - costa * amounta - costb * amountb - costc * amountc;
```

But the MAX statement and the assignment to the name f in the PROC NLP statements are combined in PROC OPTMODEL. There are advantages and disadvantages to this approach. The PROC OPTMODEL formulation is much closer to the mathematical formulation of the model. However, if multiple intermediate variables are used to structure the objective, then multiple IMPVAR declarations are required.

In the PROC NLP model, the nonlinear constraints use the following syntax:

```
nlincon nlc1-nlc2 >= 0.,
        nlc3 = 0.;
. . .
nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
```

In the PROC OPTMODEL model, the equivalent statements are as follows:

```
con nlc1: 2.5 * amountx - pools * pooltox - 2. * ctox >= 0,
    nlc2: 1.5 * amounty - pools * pooltoy - 2. * ctoy >= 0,
    nlc3: 3 * amounta + amountb - pools * (amounta + amountb)
          = 0;
```

The nonlinear constraints in PROC OPTMODEL use the same syntax as linear constraints. In fact, if the variable pools were declared as a parameter, then all the preceding constraints would be linear. The nonlinear constraint in PROC OPTMODEL combines the NLINCON statement of PROC NLP with the assignment in the PROC NLP statements. Objective names can be used in nonlinear constraint expressions to structure the formula as they are in objective expressions,

The PROC OPTMODEL model does not use a RUN statement to invoke the solver. Instead the solver is invoked interactively by the SOLVE statement in PROC OPTMODEL. By default, the OPTMODEL procedure prints much less information about the optimization process. Generally this information consists of messages from the solver (such as the termination reason) and a short status display. The PROC OPTMODEL statements add a PRINT statement in order to display the variable estimates from the solver.

Examples: OPTMODEL Procedure

Example 5.1: Matrix Square Root

This example demonstrates the use of PROC OPTMODEL array parameters and variables. The following statements create a randomized positive definite symmetric matrix and define an optimization model to find the matrix square root of the generated matrix:

```
proc optmodel;
   number n = 5;  /* size of matrix */
   /* random original array */
   number A{1..n, 1..n} = 10 - 20*rand('UNIFORM');
   /* compute upper triangle of the
    * symmetric matrix A*transpose(A) */
   /* should be positive def unless A is singular */
   number P{i in 1..n, j in i..n};
   for {i in 1..n, j in i..n}
       P[i,j] = sum{k in 1..n} A[i,k]*A[j,k];
   /* coefficients of square root array
    * (upper triangle of symmetric matrix) */
   var q{i in 1..n, i..n};
   /* The default initial value q[i,j]=0 is
    * a local minimum of the objective,
    * so you must move it away from that point. */
   q[1,1] = 1;
   /* minimize difference of square of q from P */
   min r = sum{i in 1..n, j in i..n}
           (   sum{k in 1..i} q[k,i]*q[k,j]
             + sum{k in i+1..j} q[i,k]*q[k,j]
```

```
            + sum{k in j+1..n} q[i,k]*q[j,k]
            - P[i,j] )**2;
solve;
print q;
```

These statements define a random array **A** of size $n \times n$. The product **P** is defined as the matrix product AA^T. The product is symmetric, so the declaration of the parameter P gives it upper triangular indexing. The matrix represented by P should be positive definite unless **A** is singular. But singularity is unlikely because of the random generation of **A**. If **P** is positive definite, then it has a well-defined square root, **Q**, such that $P = QQ^T$.

The objective r simply minimizes the sum of squares of the coefficients as

$$r = \sum_{1 \le i \le j \le n} R_{i,j}^2$$

where $R = QQ^T - P$. (This technique for computing matrix square roots is intended only for the demonstration of PROC OPTMODEL capabilities. Better methods exist.)

Output 5.1.1 shows part of the output from running these statements. The values that are actually displayed depend on the random numbers generated.

Output 5.1.1 Matrix Square Root Results

	q				
	1	2	3	4	5
1	-0.10557	-7.03961	8.64638	1.89284	-8.28542
2		5.23609	0.64462	-6.63339	6.71074
3			-1.61894	-7.31866	1.14428
4				3.76627	0.32063
5					4.93412

Example 5.2: Reading From and Creating a Data Set

This example demonstrates how to use the READ DATA statement to read parameters from a SAS data set. The objective is the Bard function, which is the following least squares problem with $I = \{1, 2, \ldots, 15\}$:

$$f(x) = \frac{1}{2} \sum_{k \in I} \left[y_k - \left(x_1 + \frac{k}{v_k x_2 + w_k x_3} \right) \right]^2$$

$$x = (x_1, x_2, x_3), \quad y = (y_1, y_2, \ldots, y_{15})$$

where $v_k = 16 - k$, $w_k = \min(k, v_k)$ $(k \in I)$, and

$$y = (0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39, 0.37, 0.58, 0.73, 0.96, 1.34, 2.10, 4.39)$$

The minimum function value $f(x^*) = 4.107\text{E}{-3}$ is at the point $(0.08, 1.13, 2.34)$. The starting point $x^0 = (1, 1, 1)$ is used. This problem is identical to the example "Using the DATA= Option" in Chapter 6, "The NLP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The following statements use the READ DATA statement to input parameter values and the CREATE DATA statement to save the solution in a SAS data set:

```
data bard;
   input y @@;
   datalines;
.14  .18  .22  .25  .29  .32  .35  .39
.37  .58  .73  .96 1.34 2.10 4.39
;
proc optmodel;
   set I = 1..15;
   number y{I};
   read data bard into [_n_] y;
   number v{k in I} = 16 - k;
   number w{k in I} = min(k, v[k]);
   var x{1..3} init 1;
   min f = 0.5*
      sum{k in I}
         (y[k] - (x[1] + k /
                  (v[k]*x[2] + w[k]*x[3])))**2;
   solve;
   print x;
   create data xdata from [i] xd=x;
```

In these statements the values for parameter y are read from the BARD data set. The set I indexes the terms of the objective in addition to the y array.

The preceding statements define two utility parameters that contain coefficients used in the objective function. These coefficients could have been defined in the expression for the objective, f, but it was convenient to give them names and simplify the objective expression.

The result is shown in Output 5.2.1.

Output 5.2.1 Bard Function Solution

[1]	x
1	0.08241
2	1.13303
3	2.34370

The final CREATE DATA statement saves the solution values determined by the solver into the data set XDATA. The data set contains an observation for each x index. Each observation contains two variables. The output variable i contains the index, while xd contains the value for the indexed entry in the array x. The resulting data can be seen by using the PRINT procedure as follows:

```
proc print data=xdata;
run;
```

The output from PROC PRINT is shown in Output 5.2.2.

Output 5.2.2 Output Data Set Contents

Obs	i	xd
1	1	0.08241
2	2	1.13303
3	3	2.34370

Example 5.3: Model Construction

This example uses PROC OPTMODEL features to simplify the construction of a mathematically formulated model. The model is based on the example "An Assignment Problem" in Chapter 4, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). A single invocation of PROC OPTMODEL replaces several steps in the PROC LP statements.

The model assigns production of various grades of cloth to a set of machines in order to maximize profit while meeting customer demand. Each machine has different capacities to produce the various grades of cloth. (See the PROC LP example "An Assignment Problem" for more details.) The mathematical formulation, where x_{ijk} represents the amount of cloth of grade j to produce on machine k for customer i, follows:

$$\begin{array}{lll} \max & \sum_{ijk} r_{ijk} x_{ijk} & \\ \text{subject to} & \sum_{k} x_{ijk} = d_{ij} & \text{for all } i \text{ and } j \\ & \sum_{ij} c_{jk} x_{ijk} \leq a_k & \text{for all } k \\ & x_{ijk} \geq 0 & \text{for all } i, j, \text{ and } k \end{array}$$

The OBJECT, DEMAND, and RESOURCE data sets are the same as in the PROC LP example. A new data set, GRADE, is added to help separate the data from the model.

```
title 'An Assignment Problem';

data grade(drop=i);
   do i = 1 to 6;
      grade = 'grade'||put(i,1.);
      output;
   end;
run;

data object;
   input machine customer
         grade1 grade2 grade3 grade4 grade5 grade6;
   datalines;
1 1 102 140 105 105 125 148
1 2 115 133 118 118 143 166
1 3  70 108  83  83  88  86
1 4  79 117  87  87 107 105
1 5  77 115  90  90 105 148
2 1 123 150 125 124 154   .
2 2 130 157 132 131 166   .
2 3 103 130 115 114 129   .
2 4 101 128 108 107 137   .
2 5 118 145 130 129 154   .
3 1  83   .   .  97 122 147
3 2 119   .   . 133 163 180
3 3  67   .   .  91 101 101
3 4  85   .   . 104 129 129
3 5  90   .   . 114 134 179
4 1 108 121  79   . 112 132
4 2 121 132  92   . 130 150
```

```
4 3  78  91  59   .  77  72
4 4 100 113  76   . 109 104
4 5  96 109  77   . 105 145
;

data demand;
   input customer
         grade1 grade2 grade3 grade4 grade5 grade6;
   datalines;
1 100 100 150  150  175  250
2 300 125 300  275  310  325
3 400   0 400  500  340    0
4 250   0 750  750    0    0
5   0 600 300    0  210  360
;

data resource;
   input machine
         grade1 grade2 grade3 grade4 grade5 grade6 avail;
   datalines;
1 .250 .275 .300  .350  .310  .295  744
2 .300 .300 .305  .315  .320  .     244
3 .350 .    .     .320  .315  .300  790
4 .280 .275 .260  .     .250  .295  672
;
```

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called SOLUTION:

```
proc optmodel;
   /* declare index sets */
   set CUSTOMERS;
   set <str> GRADES;
   set MACHINES;

   /* declare parameters */
   num return {CUSTOMERS, GRADES, MACHINES} init 0;
   num demand {CUSTOMERS, GRADES};
   num cost {GRADES, MACHINES} init 0;
   num avail {MACHINES};

   /* read the set of grades */
   read data grade into GRADES=[grade];

   /* read the set of customers and their demands */
   read data demand
      into CUSTOMERS=[customer]
      {j in GRADES} <demand[customer,j]=col(j)>;

   /* read the set of machines, costs, and availability */
   read data resource nomiss
      into MACHINES=[machine]
      {j in GRADES} <cost[j,machine]=col(j)>
      avail;
```

```
   /* read objective data */
   read data object nomiss
      into [machine customer]
      {j in GRADES} <return[customer,j,machine]=col(j)>;

   /* declare the model */
   var AmountProduced {CUSTOMERS, GRADES, MACHINES} >= 0;
   max TotalReturn = sum {i in CUSTOMERS, j in GRADES, k in MACHINES}
      return[i,j,k] * AmountProduced[i,j,k];
   con req_demand {i in CUSTOMERS, j in GRADES}:
      sum {k in MACHINES} AmountProduced[i,j,k] = demand[i,j];
   con req_avail {k in MACHINES}:
      sum {i in CUSTOMERS, j in GRADES}
         cost[j,k] * AmountProduced[i,j,k] <= avail[k];

   /* call the solver and save the results */
   solve;
   create data solution
      from [customer grade machine] = {i in CUSTOMERS, j in GRADES,
         k in MACHINES: AmountProduced[i,j,k].sol ne 0}
      amount=AmountProduced;
quit;
```

The statements use both numeric (NUM) and character (STR) index sets, which are populated from the corresponding data set variables in the READ DATA statements. The OPTMODEL parameters can be either single-dimensional (AVAIL) or multiple-dimensional (COST, DEMAND, RETURN). The RETURN and COST parameters are given initial values of 0, and the NOMISS option in the READ DATA statement tells PROC OPTMODEL to read only the nonmissing values from the input data sets. The model declaration is nearly identical to the mathematical formulation. The logical condition **`AmountProduced[i,j,k].sol ne 0`** in the CREATE DATA statement ensures that only the nonzero parts of the solution appear in the SOLUTION data set. In the PROC LP example, the creation of this data set required postprocessing of the PROC LP output data set.

The solver produces the following problem summary and solution summary:

Output 5.3.1 LP Solver Result

An Assignment Problem

Problem Summary	
Objective Sense	Maximization
Objective Function	TotalReturn
Objective Type	Linear
Number of Variables	120
Bounded Above	0
Bounded Below	120
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	34
Linear LE (<=)	4
Linear EQ (=)	30
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	220

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	TotalReturn
Solution Status	Optimal
Objective Value	871426.03763
Primal Infeasibility	1.591616E-12
Dual Infeasibility	4.263256E-14
Bound Infeasibility	1.591616E-12
Iterations	46
Presolve Time	0.00
Solution Time	0.01

The SOLUTION data set can be processed by PROC TABULATE as follows to create a compact representation of the solution:

```
proc tabulate data=solution;
   class customer grade machine;
   var   amount;
   table (machine*customer), (grade*amount=''*sum='');
run;
```

These statements produce the table shown in Output 5.3.2.

Output 5.3.2 An Assignment Problem

An Assignment Problem

		grade					
		grade1	grade2	grade3	grade4	grade5	grade6
machine	customer						
1	1	.	100.00	150.00	150.00	175.00	250.00
	2	.	.	300.00	.	.	.
	3	.	.	256.72	210.31	.	.
	4	.	.	750.00	.	.	.
	5	.	92.27	.	.	.	.
2	1	.	.	.	.	.	-0.00
	3	.	.	143.28	.	340.00	.
	5	.	.	300.00	.	.	-0.00
3	1	.	0.00	-0.00	.	.	.
	2	.	.	-0.00	275.00	310.00	325.00
	3	.	.	-0.00	289.69	.	.
	4	.	.	0.00	750.00	.	.
	5	.	-0.00	-0.00	.	210.00	360.00
4	1	100.00	.	.	0.00	.	.
	2	300.00	125.00	.	-0.00	.	.
	3	400.00	.	.	-0.00	.	.
	4	250.00	.	.	0.00	.	.
	5	.	507.73	.	.	.	.

Example 5.4: Set Manipulation

This example demonstrates PROC OPTMODEL set manipulation operators. These operators are used to compute the set of primes up to a given limit. This example does not solve an optimization problem, but similar set manipulation could be used to set up an optimization model. Here are the statements:

```
proc optmodel;
   number maxprime;  /* largest number to consider */
   set composites =
       union {i in 3..sqrt(maxprime) by 2} i*i..maxprime by 2*i;
   set primes = {2} union (3..maxprime by 2 diff composites);
   maxprime = 500;
   put primes;
```

The set composites contains the odd composite numbers up to the value of the parameter maxprime. The even numbers are excluded here to reduce execution time and memory requirements. The UNION aggregation operation is used in the definition to combine the sets of odd multiples of i for $i = 3, 5, \ldots$. Any composite number less than the value of the parameter maxprime has a divisor $\leq \sqrt{\text{maxprime}}$, so the range of i can be limited. The set of multiples of i can also be started at $i \times i$ since smaller multiples are found in the set of multiples for a smaller index.

You can then define the set primes. The odd primes are determined by using the DIFF operator to remove the composites from the set of odd numbers no greater than the parameter maxprime. The UNION operator adds the single even prime, 2, to the resulting set of primes.

The PUT statement produces the result in Output 5.4.1.

Output 5.4.1 Primes less than or equal to 500

```
{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,
107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,211,
223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,331,
337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,433,439,443,449,
457,461,463,467,479,487,491,499}
```

Note that you were able to delay the definition of the value of the parameter maxprime until just before the PUT statement. Since the defining expressions of the SET declarations are handled symbolically, the value of maxprime is not necessary until you need the value of the set primes. Because the sets composites and primes are defined symbolically, their values reflect any changes to the parameter maxprime. You can see this update by appending the following statements to the preceding statements:

```
maxprime = 50;
put primes;
```

The additional statements produce the results in Output 5.4.2. The value of the set primes has been recomputed to reflect the change to the parameter maxprime.

Output 5.4.2 Primes less than or equal to 50

```
{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
```

Example 5.5: Multiple Subproblems

Many important optimization problems cannot be solved directly using a standard solver, either because the problem has constraints that cannot be modeled directly or because the resulting model would be too large to be practical. For these types of problems, you can use PROC OPTMODEL to synthesize solution methods by using a combination of the existing solvers and the modeling language programming constructions. This example demonstrates the use of multiple subproblems to solve the cutting stock problem.

The cutting stock problem determines how to efficiently cut raw stock into finished widths based on the demands for the final product. Consider the example from page 195 of Chvátal (1983), where raw stock has a width of 100 inches and the demands are shown in Table 5.18.

Table 5.18 Cutting Stock Demand

Finished Width	Demand
45 inches	97
35 inches	610
31 inches	395
14 inches	211

A portion of the demand can be satisfied using a cutting pattern. For example, with the demands in Table 5.18 a possible pattern cuts one final of width 35 inches, one final of width 31 inches, and two finals of width 14 inches. This gives:

$$100 = 0 * 45 + 1 * 35 + 1 * 31 + 2 * 14 + \text{waste}.$$

The cutting stock problem can be formulated as follows, where x_j represents the number of times pattern j appears, a_{ij} represents the number of times demand item i appears in pattern j, d_i is the demand for item i, w_i is the width of item i, N represents the set of patterns, M represents the set of items, and W is the width of the raw stock:

$$\begin{array}{ll} \text{minimize} & \sum_{j \in N} x_j \\ \text{subject to} & \sum_{j \in N} a_{ij} x_j \geq d_i \quad \text{for all } i \in M \\ & x_j \text{ integer}, \geq 0 \quad \text{for all } j \in N \end{array}$$

Also for each feasible pattern j you must have:

$$\sum_{i \in M} w_i a_{ij} \leq W$$

The difficulty with this formulation is that the number of patterns can be very large, with too many columns x_j to solve efficiently. But you can use column generation, as described on page 198 of Chvátal (1983), to generate a smaller set of useful patterns, starting from an initial feasible set.

The dual variables, π_i, of the demand constraints are used to price out the columns. From linear programming (LP) duality theory, a column that improves the primal solution must have a negative reduced cost. For this problem the reduced cost for column x_j is

$$1 - \sum_{i \in M} \pi_i a_{ij}$$

Using this observation produces a knapsack subproblem:

$$\begin{array}{ll} \text{minimize} & 1 - \sum_{i \in M} \pi_i a_i \\ \text{subject to} & \sum_{i \in M} w_i a_i \leq W \\ & a_i \text{ integer}, \geq 0 \quad \text{for all } j \in N \end{array}$$

where the objective is equivalent to:

$$\text{maximize} \quad \sum_{i \in M} \pi_i a_i$$

The pattern is useful if the associated reduced cost is negative:

$$1 - \sum_{i \in M} \pi_i a_i^* < 0$$

So you can use the following steps to generate the patterns and solve the cutting stock problem:

1. Initialize a set of trivial (one item) patterns.
2. Solve the problem using the LP solver.
3. Minimize the reduced cost using a knapsack solver.
4. Include the new pattern if the reduced cost is negative.
5. Repeat steps 2 through 4 until there are no more negative reduced cost patterns.

These steps are implemented in the following statements. Since adding columns preserves primal feasibility, the statements use the primal simplex solver to take advantage of a warm start. The statements also solve the LP relaxation of the problem, but you want the integer solution. So the statements finish by using the MILP solver with the integer restriction applied. The result is not guaranteed to be optimal, but lower and upper bounds can be provided for the optimal objective.

```
/* cutting-stock problem */
/* uses delayed column generation from
   Chvatal's Linear Programming (1983), page 198 */

%macro csp(capacity);
proc optmodel printlevel=0;
   /* declare parameters and sets */
   num capacity = &capacity;
   set ITEMS;
   num demand {ITEMS};
   num width  {ITEMS};
   num num_patterns init card(ITEMS);
   set PATTERNS = 1..num_patterns;
   num a {ITEMS, PATTERNS};
   num c {ITEMS} init 0;
   num epsilon = 1E-6;

   /* read input data */
   read data indata into ITEMS=[_N_] demand width;

   /* generate trivial initial columns */
   for {i in ITEMS, j in PATTERNS}
      a[i,j] = (if (i = j) then floor(capacity/width[i]) else 0);

   /* define master problem */
   var x {PATTERNS} >= 0 integer;
   minimize NumberOfRaws = sum {j in PATTERNS} x[j];
   con demand_con {i in ITEMS}:
      sum {j in PATTERNS} a[i,j] * x[j] >= demand[i];
   problem Master include x NumberOfRaws demand_con;

   /* define column generation subproblem */
   var y {ITEMS} >= 0 integer;
   maximize KnapsackObjective = sum {i in ITEMS} c[i] * y[i];
   con knapsack_con:
      sum {i in ITEMS} width[i] * y[i] <= capacity;
   problem Knapsack include y KnapsackObjective knapsack_con;
```

```
   /* main loop */
   do while (1);
      print _page_ a;

      /* master problem */
      /* minimize sum_j x[j]
         subj. to sum_j a[i,j] * x[j] >= demand[i]
                  x[j] >= 0 and integer */
      use problem Master;
      put "solve master problem";
      solve with lp relaxint /
         presolver=none solver=ps basis=warmstart printfreq=1;
      print x;
      print demand_con.dual;
      for {i in ITEMS} c[i] = demand_con[i].dual;

      /* knapsack problem */
      /* maximize sum_i c[i] * y[i]
         subj. to sum_i width[i] * y[i] <= capacity
                  y[i] >= 0 and integer */
      use problem Knapsack;
      put "solve column generation subproblem";
      solve with milp / printfreq=0;
      for {i in ITEMS} y[i] = round(y[i]);
      print y;
      print KnapsackObjective;

      if KnapsackObjective <= 1 + epsilon then leave;

      /* include new pattern */
      num_patterns = num_patterns + 1;
      for {i in ITEMS} a[i,num_patterns] = y[i];
   end;

   /* solve IP, using rounded-up LP solution as warm start */
   use problem Master;
   for {j in PATTERNS} x[j] = ceil(x[j].sol);
   put "solve (restricted) master problem as IP";
   solve with milp / primalin;

   /* cleanup solution and save to output data set */
   for {j in PATTERNS} x[j] = round(x[j].sol);
   create data solution from [pattern]={j in PATTERNS: x[j] > 0}
      raws=x {i in ITEMS} <col('i'||i)=a[i,j]>;
quit;
%mend csp;

/* Chvatal, p.199 */
data indata;
   input demand width;
   datalines;
78 25.5
40 22.5
```

```
30 20
30 15
;
%csp(91)
/* LP solution is integer */

/* Chvatal, p.195 */
data indata;
   input demand width;
   datalines;
 97 45
610 36
395 31
211 14
;
%csp(100)
/* LP solution is fractional */
```

The contents of the output data set for the second problem instance are shown in Output 5.5.1.

Output 5.5.1 Cutting Stock Solution

Obs	pattern	raws	i1	i2	i3	i4
1	1	49	2	0	0	0
2	5	206	0	2	0	2
3	6	198	0	1	2	0

Example 5.6: Traveling Salesman Problem

This example demonstrates the use of the SUBMIT statement to execute a block of SAS statements from within a PROC OPTMODEL session. In this case, the SUBMIT block calls the SGPLOT procedure to display intermediate results during the solution of an instance of the traveling salesman problem (TSP). The problem is described in Example 9.4. For an example of how to use PROC OPTNET to solve the TSP, see "Traveling Salesman Problem Applied to a Simple Directed Graph" (Chapter 2, *SAS/OR User's Guide: Network Optimization Algorithms*).

The following DATA step converts a TSPLIB instance of type EUC_2D into a SAS data set that contains the coordinates of the vertices:

```
/* convert the TSPLIB instance into a data set */
data tspData(drop=H);
   infile "&tsplib";
   input H $1. @;
   if H not in ('N','T','C','D','E');
   input @1 var1-var3;
run;
```

The following macro generates plots of the solution and objective value:

```
%macro plotTSP;
   /* create Annotate data set to draw subtours */
   data anno;
      retain drawspace 'datavalue' linethickness 1 function 'line';
```

```
      set solData;
   run;

   title1 h=2 "TSP: Iter = &i, Objective = &&obj&i";
   title2;

   proc sgplot data=tspData sganno=anno;
      scatter x=var2 y=var3 / datalabel=var1;
      xaxis display=none;
      yaxis display=none;
   run;
%mend plotTSP;
```

The following PROC OPTMODEL statements solve the TSP by using the subtour formulation and iteratively adding subtour constraints. The SUBMIT statement calls the **%plotTSP** macro to plot the solution and objective value at each stage.

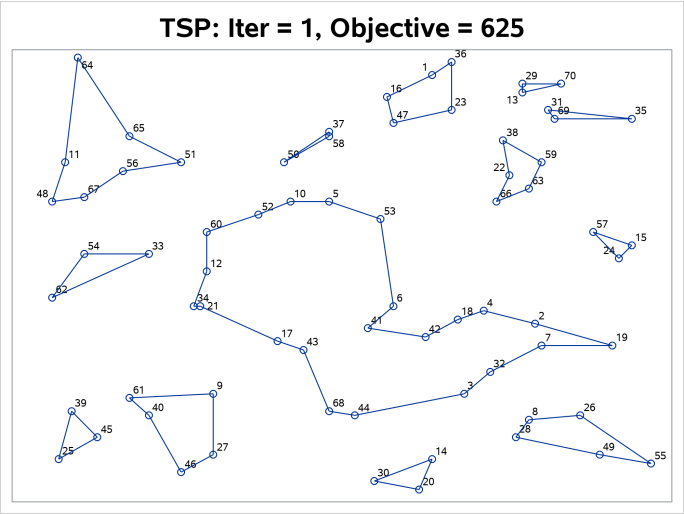
```
/* iterative solution using the subtour formulation */
proc optmodel;
   set VERTICES;
   set EDGES = {i in VERTICES, j in VERTICES: i > j};
   num xc {VERTICES};
   num yc {VERTICES};

   num numsubtour init 0;
   set SUBTOUR {1..numsubtour};

   /* read in the instance and customer coordinates (xc, yc) */
   read data tspData into VERTICES=[var1] xc=var2 yc=var3;

   /* the cost is the euclidean distance rounded to the nearest integer */
   num c {<i,j> in EDGES}
      init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);

   var x {EDGES} binary;

   /* minimize the total cost */
   min obj =
       sum {<i,j> in EDGES} c[i,j] * x[i,j];

   /* each vertex has exactly one in-edge and one out-edge */
   con two_match {i in VERTICES}:
      sum {j in VERTICES: i > j} x[i,j]
      + sum {j in VERTICES: i < j} x[j,i] = 2;

   /* no subtours (these constraints are generated dynamically) */
   con subtour_elim {s in 1..numsubtour}:
      sum {<i,j> in EDGES: (i in SUBTOUR[s] and j not in SUBTOUR[s])
         or (i not in SUBTOUR[s] and j in SUBTOUR[s])} x[i,j] >= 2;

   /* this starts the algorithm to find violated subtours */
   set <num,num> EDGES1;
   set VERTICES1 = union{<i,j> in EDGES1} {i, j};
```

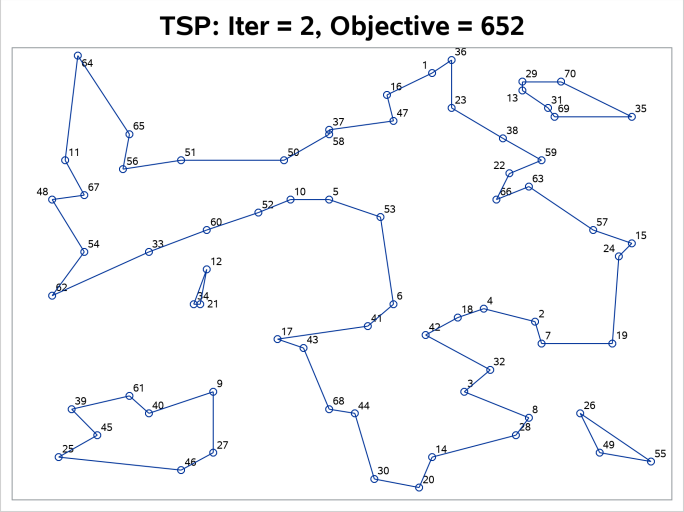
```
num component {VERTICES1};
num numcomp  init 2;
num iter     init 1;
call symput('i',trim(left(put(round(iter),best.))));
num numiters init 1;

/* initial solve with just matching constraints */
solve;
call symput(compress('obj'||put(iter,best.)),
            trim(left(put(round(obj),best.))));

/* create a data set for use by PROC SGPLOT */
create data solData from
   [i j]={<i,j> in EDGES: x[i,j].sol > 0.5}
   x1=xc[i] y1=yc[i] x2=xc[j] y2=yc[j];
submit;
   %plotTSP;
endsubmit;
/* while the solution is disconnected, continue */
do while (numcomp > 1);
   iter = iter + 1;
   call symput('i',trim(left(put(round(iter),best.))));

   /* find connected components of support graph   */
   EDGES1 = {<i,j> in EDGES: round(x[i,j].sol) = 1};
   solve with network /
      links   = (include=EDGES1)
      nodes   = (include=VERTICES1)
      concomp
      out     = (concomp=component);

   numcomp = _oroptmodel_num_["NUM_COMPONENTS"];
   if numcomp = 1 then leave;

   numiters = iter;
   numsubtour = numsubtour + numcomp;
   for {comp in 1..numcomp} do;
      SUBTOUR[numsubtour-numcomp+comp]
        = {i in VERTICES: component[i] = comp};
   end;

   solve;
   call symput(compress('obj'||put(iter,best.)),
               trim(left(put(round(obj),best.))));

   /* create a data set for use by PROC SGPLOT */
   create data solData from
      [i j]={<i,j> in EDGES: x[i,j].sol > 0.5}
      x1=xc[i] y1=yc[i] x2=xc[j] y2=yc[j];

   call symput('numiters',put(numiters,best.));
   submit;
      %plotTSP;
   endsubmit;
```

```
    end;
quit;
```

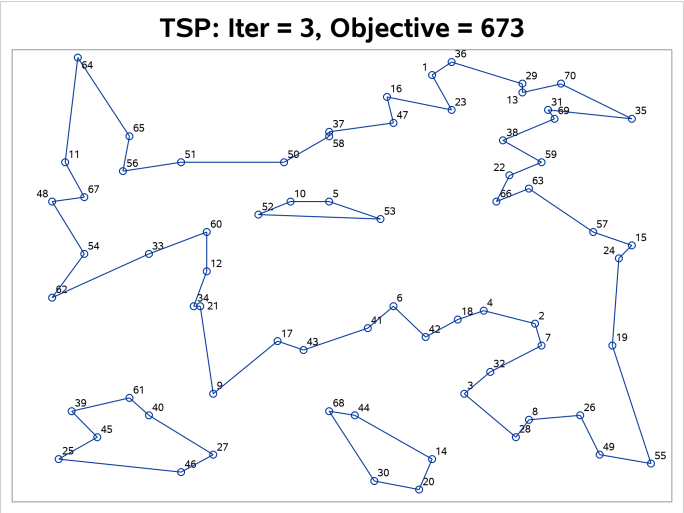
The plot in Output 5.6.1 shows the solution and objective value at each stage. Each stage restricts some subset of subtours. When you reach the final stage, you have a valid tour.

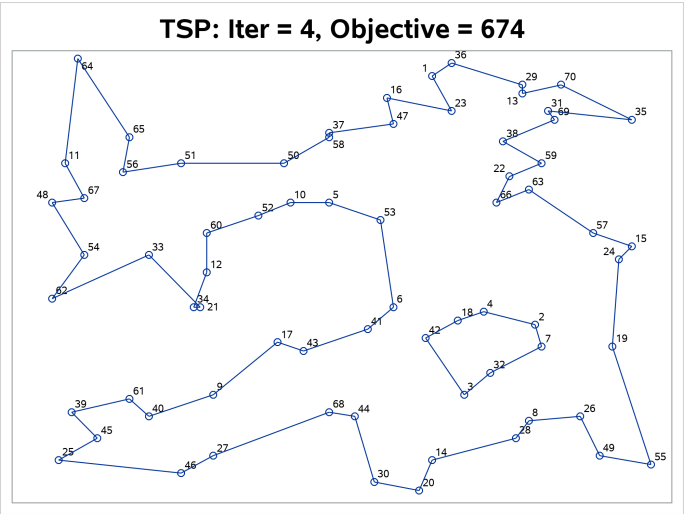
Output 5.6.1 Iterative Solution of Traveling Salesman Problem

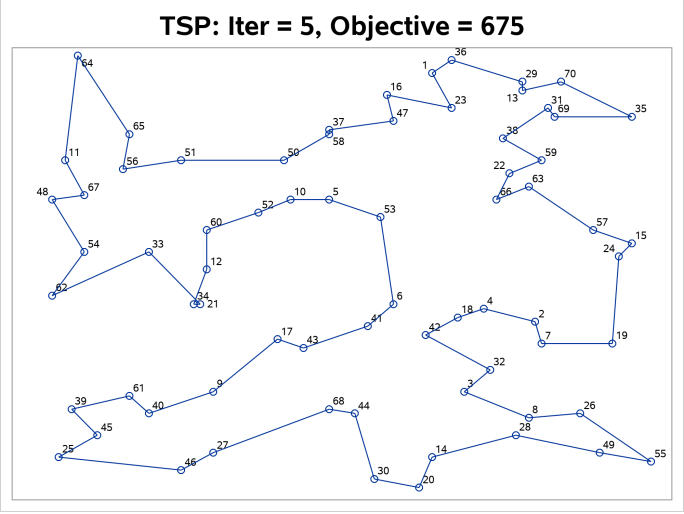




Output 5.6.1 *continued*







Example 5.7: Sparse Modeling

This example demonstrates how to rewrite certain models for more efficient processing. Sometimes optimization models that run out of memory during problem generation can be rewritten to take advantage of sparsity to use memory more efficiently. This often occurs when a large array is modeled in a dense format but most of its entries are zeros. Usually, the array provides problem coefficients or it contains optimization variables.

The model for this example solves the facility location problem that is described in Example 9.3. This example is concerned with the resources that are required for PROC OPTMODEL problem generation and solver initialization. So the size of the problem has been increased, but the model has also been modified to make it easier to solve. In order to handle the larger problem size, the model eliminates a large number of the potential assignments of customers to facilities based on distance, making the problem sparse.

The following code generates a random instance of the facility location problem:

```
%let NumCustomers  = 1500;
%let NumSites      = 250;
%let SiteCapacity  = 50;
%let MaxDemand     = 10;
%let xmax          = 200;
%let ymax          = 100;
%let seed          = 938;

/* generate random customer locations */
data cdata(drop=i);
   length name $8;
   call streaminit(&seed);
   do i = 1 to &NumCustomers;
      name = compress('C'||put(i,best.));
      x = rand('UNIFORM') * &xmax;
      y = rand('UNIFORM') * &ymax;
      demand = 1;
      output;
   end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
   length name $8;
   call streaminit(&seed);
   do i = 1 to &NumSites;
      name = compress('SITE'||put(i,best.));
      x = rand('UNIFORM') * &xmax;
      y = rand('UNIFORM') * &ymax;
      fixed_charge = 300 * (abs(&xmax/2-x)/&xmax + abs(&ymax/2-y)/&ymax);
      output;
   end;
run;
```

The following code uses a dense version of the facility location model. This model is equivalent to the model from Example 9.3 except for the added constraint distance_at_most_30. This constraint eliminates from consideration the assignment of customers to facilities over long distances by forcing the corresponding Assign variables to 0.

```
proc optmodel printlevel=2;
   profile on percent=0.1;
   set <str> CUSTOMERS;
   set <str> SITES init {};

   /* x and y coordinates of CUSTOMERS and SITES */
   num x {CUSTOMERS union SITES};
   num y {CUSTOMERS union SITES};
   num demand {CUSTOMERS};
   num fixed_charge {SITES};

   /* distance from customer i to site j */
   num dist {i in CUSTOMERS, j in SITES}
       = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

   read data cdata into CUSTOMERS=[name] x y demand;
   read data sdata into SITES=[name] x y fixed_charge;

   var Assign {CUSTOMERS, SITES} binary;
   var Build {SITES} binary;

   min CostNoFixedCharge
       = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];
   min CostFixedCharge
       = CostNoFixedCharge + sum {j in SITES} fixed_charge[j] * Build[j];

   /* each customer assigned to exactly one site */
   con assign_def {i in CUSTOMERS}:
      sum {j in SITES} Assign[i,j] = 1;

   /* if customer i assigned to site j, then facility must be built at j */
   con link {i in CUSTOMERS, j in SITES}:
      Assign[i,j] <= Build[j];

   /* each site can handle at most &SiteCapacity demand */
   con capacity {j in SITES}:
      sum {i in CUSTOMERS} demand[i] * Assign[i,j] <=
         &SiteCapacity * Build[j];

   /* do not assign customer to site more than 30 units away */
   con distance_at_most_30 {i in CUSTOMERS, j in SITES: dist[i,j] > 30}:
      Assign[i,j] = 0;

   /* solve the MILP */
   solve with milp/timetype=real;

quit;
```

If you inspect the log after running the preceding code, then you will see that the MILP presolver has pruned down the problem size considerably. If you also run the code with the SAS option FULLSTIMER enabled on a 64-bit system, then you will notice that about 1.3GB of memory is required for the PROC OPTMODEL step when you are running on a single CPU.

The solution and timing results for the dense model are shown in Output 5.7.1. The PROFILE ON statement requests further timing details, including evaluation time for declarations used by problem generation.

Output 5.7.1 Dense Model Results

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	CostFixedCharge
Solution Status	Optimal within Relative Gap
Objective Value	18001.140353
Relative Gap	0.0000287924
Absolute Gap	0.5182815753
Primal Infeasibility	6.57252E-13
Bound Infeasibility	1.042499E-13
Integer Infeasibility	1.101341E-13
Best Bound	18000.622072
Nodes	12
Solutions Found	5
Iterations	33823
Presolve Time	20.77
Solution Time	268.52

Procedure Task Timing		
Task	**Time (sec.)**	**% Time**
Problem Generation	25.92	8.61%
Solver Initialization	5.51	1.83%
Code Generation	0.16	0.05%
Presolve	20.77	6.89%
Root Node Processing	212.16	70.44%
Branch And Cut	18.65	6.19%
Synchronization	2.29	0.76%
Idle	10.64	3.53%
Other Tasks	4.02	1.34%
Solver Postprocessing	1.06	0.35%

Output 5.7.1 *continued*

Profile Information

Item	Line	Col.	Execution Count	Net Time (sec.)	Wait Time (sec.)	% Total Time
SOLVE	3626	4	1	278.57	6.72	71.5%
Constraint distance_at_most_30	3622	8		55.72	0.00	14.3%
Constraint link	3613	8		29.03	0.00	7.5%
Constraint assign_def	3609	8		10.03	7.20	2.6%
Constraint capacity	3617	8		5.23	0.17	1.3%
Number dist	3594	8		5.03	0.00	1.3%
Min CostNoFixedCharge	3603	8		3.23	0.00	0.8%
Var Assign	3600	8		2.58	0.00	0.7%
Other profiled items				0.06	0.00	0.0%

Note: Total profiled time is 389.48 seconds.

The best approach for reducing the memory requirements is to eliminate the Assign variables that are always going to be 0. This is accomplished in the following sparse version of the code. Instead of indexing Assign over the crossproduct of CUSTOMERS and SITES, now the code defines a new set of pairs that satisfy the distance requirement, CUSTOMERS_SITES. This set replaces the constraint distance_at_most_30 in the dense model. The objective and constraints have been modified to use the new indexing scheme, with implicit set slicing (as described in the section "More on Index Sets" on page 160) for constraints assign_def and capacity.

```
proc optmodel printlevel=2;
   profile on percent=0.1;
   set <str> CUSTOMERS;
   set <str> SITES init {};

   /* x and y coordinates of CUSTOMERS and SITES */
   num x {CUSTOMERS union SITES};
   num y {CUSTOMERS union SITES};
   num demand {CUSTOMERS};
   num fixed_charge {SITES};

   /* distance from customer i to site j */
   num dist {i in CUSTOMERS, j in SITES}
       = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

   read data cdata into CUSTOMERS=[name] x y demand;
   read data sdata into SITES=[name] x y fixed_charge;

   set CUSTOMERS_SITES = {i in CUSTOMERS, j in SITES: dist[i,j] <= 30};
   var Assign {CUSTOMERS_SITES} binary;
   var Build {SITES} binary;

   min CostNoFixedCharge
       = sum {<i,j> in CUSTOMERS_SITES} dist[i,j] * Assign[i,j];
   min CostFixedCharge
       = CostNoFixedCharge + sum {j in SITES} fixed_charge[j] * Build[j];

   /* each customer assigned to exactly one site */
```

```
   con assign_def {i in CUSTOMERS}:
      sum {<(i),j> in CUSTOMERS_SITES} Assign[i,j] = 1;

   /* if customer i assigned to site j, then facility must be built at j */
   con link {<i,j> in CUSTOMERS_SITES}:
      Assign[i,j] <= Build[j];

   /* each site can handle at most &SiteCapacity demand */
   con capacity {j in SITES}:
      sum {<i,(j)> in CUSTOMERS_SITES} demand[i] * Assign[i,j] <=
         &SiteCapacity * Build[j];

   /* solve the MILP */
   solve with milp/timetype=real;

quit;
```

The log from running the preceding code shows that the MILP presolver does not find anything to improve with this version of the model. On a 64-bit system, the FULLSTIMER option shows that memory requirements have been reduced to about 580MB when you are running on a single CPU, less than half the requirements of the previous model.

The solution and timing results for the sparse model are shown in Output 5.7.2. Note that the dense model (Output 5.7.1) and the sparse model (Output 5.7.2) are equivalent after presolver processing and generate the same result using similar amounts of solver time. On the other hand, problem generation time is significantly reduced as are other times including presolve time. Both models used the solver option TIMETYPE=REAL so that all times are reported in seconds of real time. You can see from the "Profile Information" table that the overhead associated with problem declarations has been significantly reduced.

Output 5.7.2 Sparse Model Results

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	CostFixedCharge
Solution Status	Optimal within Relative Gap
Objective Value	18001.140353
Relative Gap	0.0000287924
Absolute Gap	0.5182815753
Primal Infeasibility	6.57252E-13
Bound Infeasibility	1.042499E-13
Integer Infeasibility	1.101341E-13
Best Bound	18000.622072
Nodes	12
Solutions Found	7
Iterations	33823
Presolve Time	37.35
Solution Time	282.70

Output 5.7.2 *continued*

Procedure Task Timing

Task	Time (sec.)	% Time
Problem Generation	10.12	3.45%
Solver Initialization	0.63	0.22%
Code Generation	0.02	0.01%
Presolve	37.35	12.72%
Root Node Processing	209.21	71.26%
Branch And Cut	20.12	6.85%
Synchronization	1.97	0.67%
Idle	9.95	3.39%
Other Tasks	4.11	1.40%
Solver Postprocessing	0.11	0.04%

Profile Information

Item	Line	Col.	Execution Count	Net Time (sec.)	Wait Time (sec.)	% Total Time
SOLVE	3675	4	1	284.11	0.39	95.0%
Number dist	3646	8		4.81	0.00	1.6%
Set CUSTOMERS_SITES	3652	8		3.98	0.00	1.3%
Constraint link	3666	8		2.54	0.00	0.8%
Constraint assign_def	3662	8		1.80	0.00	0.6%
Constraint capacity	3670	8		0.98	0.05	0.3%
Min CostNoFixedCharge	3656	8		0.37	0.00	0.1%
Other profiled items				0.34	0.00	0.1%

Note: Total profiled time is 298.94 seconds.

Example 5.8: Chemical Equilibrium

This example illustrates how to convert PROC NLP code that handles arrays into PROC OPTMODEL form. The following PROC NLP model finds an equilibrium state for a mixture of chemicals. The same model is used in "Example 7.8: Chemical Equilibrium" in Chapter 6, "The NLP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

```
proc nlp tech=tr pall;
   array c[10] -6.089 -17.164 -34.054  -5.914 -24.721
               -14.986 -24.100 -10.708 -26.662 -22.179;
   array x[10] x1-x10;
   min y;
   parms x1-x10 = .1;
   bounds 1.e-6 <= x1-x10;
   lincon 2. = x1 + 2. * x2 + 2. * x3 + x6 + x10,
          1. = x4 + 2. * x5 + x6 + x7,
          1. = x3 + x7  + x8 + 2. * x9 + x10;
   s = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
   y = 0.;
   do j = 1 to 10;
      y = y + x[j] * (c[j] + log(x[j] / s));
   end;
```

```
run;
```

The following statements show a corresponding PROC OPTMODEL model:

```
proc optmodel;
   set CMP = 1..10;
   number c{CMP} = [-6.089 -17.164 -34.054  -5.914 -24.721
                    -14.986 -24.100 -10.708 -26.662 -22.179];
   var x{CMP} init 0.1 >= 1.e-6;
   con 2. = x[1] + 2. * x[2] + 2. * x[3] + x[6] + x[10],
       1. = x[4] + 2. * x[5] + x[6] + x[7],
       1. = x[3] + x[7]  + x[8] + 2. * x[9] + x[10];
   /* replace the variable s in the PROC NLP model */
   impvar s = sum{i in CMP} x[i];
   min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
   solve;
   print x y;
```

The PROC OPTMODEL model uses the set CMP to represent the set of compounds, which are numbered 1 to 10 in the example. The array c was initialized by using the equivalent PROC OPTMODEL syntax. The individual array locations could also have been initialized by assignment or by READ DATA statements.

The VAR declaration for variable x combines the VAR and BOUNDS statements of the PROC NLP model. The index set of the array is based on the set of compounds CMP to simplify changes to the model.

The linear constraints are similar in form to the PROC NLP model. However, the PROC OPTMODEL version uses the array form of the variable names because it treats arrays as distinct variables, not as aliases of lists of scalar variables.

The implicit variable s replaces the intermediate variable of the same name in the PROC NLP model. This is an example of translating an intermediate variable from the other models to PROC OPTMODEL. An alternative way is to use an additional constraint for every intermediate variable. Instead of declaring objective s as in the preceding statements, you can use the following statements:

```
. . .
var s;
con s = sum{i in CMP} x[i];
. . .
```

Note that this alternative formulation passes an extra variable and constraint to the solver. This formulation can sometimes be solved more efficiently, depending on the characteristics of the model.

The PROC OPTMODEL version uses a SUM operator over the set CMP, which enhances the flexibility of the model to accommodate possible changes in the set of compounds.

In the PROC NLP model, the objective function y is determined by an explicit loop. The DO loop in PROC NLP is replaced by a SUM aggregation operation in PROC OPTMODEL. The accumulation in the PROC NLP model is now performed by PROC OPTMODEL by using the SUM operator.

This PROC OPTMODEL model can be generalized further. Note that the array initialization and constraints assume a fixed set of compounds. You can rewrite the model to handle an arbitrary number of compounds and chemical elements. The new model loads the linear constraint coefficients from a data set along with the objective coefficients for the parameter c, as follows:

```
data comp;
   input c a_1 a_2 a_3;
   datalines;
-6.089   1 0 0
-17.164  2 0 0
-34.054  2 0 1
-5.914   0 1 0
-24.721  0 2 0
-14.986  1 1 0
-24.100  0 1 1
-10.708  0 0 1
-26.662  0 0 2
-22.179  1 0 1
;

data atom;
   input b @@;
   datalines;
2. 1. 1.
;

proc optmodel;
   set CMP;
   set ELT;
   number c{CMP};
   number a{ELT,CMP};
   number b{ELT};
   read data atom into ELT=[_n_] b;
   read data comp into CMP=[_n_]
        c {i in ELT} < a[i,_n_]=col("a_"||i) >;
   var x{CMP} init 0.1 >= 1.e-6;
   con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
   impvar s = sum{i in CMP} x[i];
   min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
   print a b;
   solve;
   print x;
```

This version adds coefficients for the linear constraints to the COMP data set. The data set variable a_*n* represents the number of atoms in the compound for element *n*. The READ DATA statement for COMP uses the iterated column syntax to read each of the data set variables a_*n* into the appropriate location in the array a. In this example the expanded data set variable names are a_1, a_2, and a_3.

The preceding version also adds a new set, ELT, of chemical elements and a numeric parameter, b, that represents the left-hand side of the linear constraints. The data values for the parameters ELT and b are read from the data set ATOM. The model can handle varying sets of chemical elements because of this extra data set and the new parameters.

The linear constraints have been converted to a single, indexed family of constraints. One constraint is applied for each chemical element in the set ELT. The constraint expression uses a simple form that applies generally to linear constraints. The following PRINT statement in the model shows the values that are read from the data sets to define the linear constraints:

```
print a b;
```

The PRINT statements in the model produce the results shown in Output 5.8.1.

Output 5.8.1 PROC OPTMODEL Output

	a									
	1	2	3	4	5	6	7	8	9	10
1	1	2	2	0	0	1	0	0	0	1
2	0	0	0	1	2	1	1	0	0	0
3	0	0	1	0	0	0	1	1	2	1

[1]	b
1	2
2	1
3	1

[1]	x
1	0.04066848
2	0.14773067
3	0.78315260
4	0.00141459
5	0.48524616
6	0.00069358
7	0.02739955
8	0.01794757
9	0.03731444
10	0.09687143

In the preceding model, the chemical elements and compounds are designated by numbers. So in the PRINT output, for example, the row that is labeled "3" represents the amount of the compound H_2O. PROC OPTMODEL is capable of using more symbolic strings to designate array indices. The following version of the model uses strings to index arrays:

```
data comp;
   input name $ c a_h a_n a_o;
   datalines;
H     -6.089   1 0 0
H2    -17.164  2 0 0
H2O   -34.054  2 0 1
N     -5.914   0 1 0
N2    -24.721  0 2 0
NH    -14.986  1 1 0
NO    -24.100  0 1 1
O     -10.708  0 0 1
O2    -26.662  0 0 2
OH    -22.179  1 0 1
;
data atom;
   input name $ b;
   datalines;
```

```
H  2.
N  1.
O  1.
;
proc optmodel;
   set<string> CMP;
   set<string> ELT;
   number c{CMP};
   number a{ELT,CMP};
   number b{ELT};
   read data atom into ELT=[name] b;
   read data comp into CMP=[name]
        c {i in ELT} < a[i,name]=col("a_"||i) >;
   var x{CMP} init 0.1 >= 1.e-6;
   con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
   impvar s = sum{i in CMP} x[i];
   min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
   solve;
   print x;
```

In this model, the sets CMP and ELT are now sets of strings. The data sets provide the names of the compounds and elements. The names of the data set variables for atom counts in the data set COMP now include the chemical element symbol as part of their spelling. For example, the atom count for element H (hydrogen) is named a_h. Note that these changes did not require any modification to the specifications of the linear constraints or of the objective.

The PRINT statement in the preceding statements produces the results shown in Output 5.8.2. The indices of variable x are now strings that represent the actual compounds.

Output 5.8.2 PROC OPTMODEL Output with Strings

[1]	x
H	0.04066848
H2	0.14773067
H2O	0.78315260
N	0.00141459
N2	0.48524616
NH	0.00069358
NO	0.02739955
O	0.01794757
O2	0.03731444
OH	0.09687143

References

Abramowitz, M., and Stegun, I. A., eds. (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* 10th printing. New York: Dover.

Bazaraa, M. S., Sherali, H. D., and Shetty, C. M. (1993). *Nonlinear Programming: Theory and Algorithms*. New York: John Wiley & Sons.

Chvátal, V. (1983). *Linear Programming*. New York: W. H. Freeman.

Dennis, J. E., and Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall.

Fletcher, R. (1987). *Practical Methods of Optimization*. 2nd ed. Chichester, UK: John Wiley & Sons.

Nocedal, J., and Wright, S. J. (1999). *Numerical Optimization*. New York: Springer-Verlag.

Subject Index

Syntax Index