
AMPL Syntax Update

for use with ILOG AMPL CPLEX System v8.1

The AMPL Modeling System software is copyrighted by Bell Laboratories and is distributed under license by ILOG.

CPLEX[®] is a registered trademark of ILOG.

ILOG

889 Alder Avenue, Suite 200
Incline Village, NV 89451, USA
Phone (775) 831 7744
Fax (775) 831 7755

Internet:

Product information <http://www.ilog.com/products/ampl/>
Email info@ilog.com

Contents

1 INTRODUCTION	3
2 CHANGING THE MODEL AND DATA	5
2.1 READING DATA: THE DATA, UPDATE DATA AND RESET DATA COMMANDS	5
2.2 READING DATA: THE READ COMMAND	6
2.3 REMOVING OR REDEFINING MODEL COMPONENTS	8
2.4 FIXING OR UNFIXING A VARIABLE AT A NEW VALUE	9
2.5 RELAXING INTEGRALITY	10
3 RELATIONAL DATABASE ACCESS.....	11
3.1 GENERAL PRINCIPLES OF DATA CORRESPONDENCE.....	12
3.2 EXAMPLES OF AMPL TABLE-HANDLING STATEMENTS.....	15
3.3 GENERAL FORMS OF THE TABLE DECLARATION.....	22
3.4 READING DATA FROM RELATIONAL TABLES	24
3.5 WRITING DATA TO RELATIONAL TABLES	29
3.6 READING AND WRITING THE SAME TABLE.....	34
3.7 INDEXED COLLECTIONS OF TABLES AND COLUMNS.....	37
3.8 STANDARD AND BUILT-IN TABLE HANDLERS	41
4 CHARACTER STRINGS.....	47
4.1 STRING FUNCTIONS AND OPERATORS.....	48
4.2 STRING EXPRESSIONS IN AMPL COMMANDS.....	50
4.3 NUMBER-TO-STRING CONVERSIONS.....	51
4.4 STRING-TO-NUMBER CONVERSIONS.....	53
4.5 CHARACTER-CODE CONVERSIONS	54
4.6 DISPLAY FORMATS FOR STRINGS	54
5 COMPLEMENTARITY PROBLEMS	57
5.1 MOTIVATION	57
5.2 COMPLEMENTARY PAIRS OF INEQUALITIES	60
5.3 COMPLEMENTARY DOUBLE INEQUALITIES.....	61
5.4 USING THE PATH SOLVER	62
5.5 PRESOLVE.....	62
5.6 AUXILIARY SOLUTION VALUES	64
5.7 GENERIC SYNONYMS	65
6 EXAMINING MODELS AND DATA.....	69
6.1 THE SHOW COMMAND: DISPLAYING MODEL COMPONENTS	69
6.2 THE XREF COMMAND: DISPLAYING MODEL DEPENDENCIES	70
6.3 THE EXPAND COMMAND: DISPLAYING MODEL INSTANCES	71
6.4 GENERIC SYNONYMS FOR VARIABLES, CONSTRAINTS AND OBJECTIVES	73
6.5 SUMMARY MODEL INFORMATION	74

7 LOOPING AND TESTING 1: WRITING "SCRIPTS" IN THE AMPL COMMAND LANGUAGE	77
7.1 RUNNING SCRIPTS.....	77
7.2 ITERATING OVER A SET	79
7.3 ITERATING SUBJECT TO A CONDITION	83
7.4 TESTING A CONDITION	85
7.5 TERMINATING A LOOP.....	88
7.6 STEPPING THROUGH A SCRIPT	89
8 LOOPING AND TESTING 2: IMPLEMENTING ALGORITHMS THROUGH AMPL SCRIPTS	93
8.1 ALTERNATING BETWEEN NAMED PROBLEMS	93
8.2 DEFINING NAMED PROBLEMS	100
8.3 USING NAMED PROBLEMS	104
8.4 DISPLAYING NAMED PROBLEMS.....	105
8.5 DEFINING AND USING NAMED ENVIRONMENTS	106
9 PRESOLVE TOLERANCES	109
9.1 DETERMINING THAT NO FEASIBLE SOLUTION EXISTS	110
9.2 ROUNDING BOUNDS ON INTEGER VARIABLES.....	112
10 REPORTING AND DISPLAY	113
10.1 SOLVER RETURN MESSAGES.....	113
10.2 TIME AND MEMORY LISTINGS	114
10.3 OUTPUT LOGS	117
10.4 LIMITS ON MESSAGES	118
10.5 PROMPTS	120
11 STATUSES.....	121
11.1 SOLVE RESULTS	121
11.2 SOLVER STATUSES OF OBJECTIVES AND PROBLEMS.....	124
11.3 SOLVER STATUSES OF VARIABLES.....	125
11.4 SOLVER STATUSES OF CONSTRAINTS	129
11.5 AMPL STATUSES	131
12 SUFFIXES.....	135
12.1 USER-DEFINED SUFFIXES	135
12.2 SOLVER-DEFINED SUFFIXES	137
12.3 THE SUFFIX STATEMENT.....	142

1 Introduction

The AMPL modeling language and environment continue to evolve in response to users' needs. Thus, many significant features have been added since the publication of the AMPL book, AMPL, A Modeling Language For Mathematical Programming (ISBN 0-534-50983-5). This document contains the verbatim (apart from minor, primarily formatting, changes) content of the language extensions published on the AMPL web site, at

<http://www.ampl.com/ampl/>

Since this living document is updated every so often, we recommend checking the web page for changes and added material. This version is current as of August 24, 2000.

2 Changing the Model and Data

Enhancements to AMPL commands let you read new data (while leaving the model the same) in a more convenient variety of ways. The uses of `data`, `update data` and `reset data` have been extended, and a new `read` command has been provided for importing data not in the standard AMPL format (chapter 9).

Declarations of AMPL model components can be removed or redefined by use of the new `purge`, `delete`, and `redeclare` commands. Other new features let you change the value of a variable while fixing or unfixing it, and relax the integrality restrictions on variables.

2.1 Reading data: the `data`, `update data` and `reset data` commands

The command consisting of only the keyword `data` puts AMPL into data mode, where you can type the data tables and lists described by Chapter 9 of the book:

```
ampl: model diet.mod;  
ampl: data;  
  
ampl data: set NUTR := A B1 B2 C ;  
ampl data: param:      n_min  n_max :=  
ampl data?      A        700   10000  
ampl data?      C        700   10000  
ampl data?      B1       700   10000  
ampl data?      B2       700   10000 ;  
  
ampl data: display n_min, n_max;  
  
:  n_min  n_max  :=  
A    700   10000  
B1   700   10000  
B2   700   10000  
C    700   10000  
;
```

AMPL exits data mode when it sees any statement (such as `display` above) that does not begin with a keyword (such as `set` or `param`) that begins a data statement.

Normally data values are read from files rather than being typed at the AMPL prompts. A command of the form

```
data filename ;
```

reads the contents of the named file as if it had been typed at the prompts, starting in data mode. The mode then reverts to whatever it was before the data statement was executed - unless the last data statement in the file is incomplete, in which case data mode persists.

When you use `data` statements to assign values to model components, AMPL checks that no component is assigned a value more than once. Duplicate assignments are flagged as errors. In some situations, however, it is convenient to be able to change the data by issuing new `data` statements; for example, after solving for one scenario of a model, you may want to modify some of the data by reading a new data file that corresponds to a second scenario. The statements in the new file would normally be treated as erroneous duplicates, but you can tell AMPL to accept them by first giving one of the following commands:

```
reset data ;  
reset data component-list ;  
  
update data ;  
update data component-list ;
```

The `reset` commands discard specified data, which must then be re-read before another problem can be generated and solved. The `update` commands retain the current data, but allow specified data values to be overwritten (once only) by subsequent `data` commands. The *component-list*, which has the same format as in the `display` command (Chapter 10.3), indicates the model components that are to be reset or updated. Without the *component-list*, these commands apply to all the components of the model.

Previous restrictions on the location of an AMPL `data` statement have now been lifted. In, particular, a `data` statement may appear within an `if` statement to conditionally read data. A `data` statement may also be used within a `for` or `repeat` statement, in conjunction with `reset` or `update`, to iteratively read a data file that is being regenerated at each pass through a loop, typically by some external program.

2.2 Reading data: the `read` command

The new `read` command provides a way of reading unformatted data into AMPL parameters and other components. It has syntax similar to that of the `print` command:

```
read arglist redirectionopt ;  
read indexing-expr : arglist redirectionopt ;
```

As in the case of `display`, `print` and `printf`, the optional *indexing-expr* causes the `read` command to be executed separately for each member of the specified indexing set. Thus, for example, a statement beginning `read {i in ORIG} : ...` has the same effect as `for {i in ORIG} read ...`.

We describe first how `read` interprets its input, and then where it looks for the input.

Interpretation of input. The `read` command treats its input as an unformatted series of data values, separated by white space (any combination of spaces, tabs and newlines). The *arglist* specifies a series of components to which these values are assigned. As in the case of `print`, the *arglist* is a comma-separated list of *args*, which may be any of:

```

component-ref
indexing-expr component-ref
( arglist )

```

The *component-ref* must be a reference to a parameter, variable, or constraint; it is meaningless to read a value into a set member or any more general expression. As in the case of `print`, all indexing must be explicit, so that for example you must say `read { j in DEST} demand[j]` rather than `read demand`.

Values are assigned to *args* in the order that they are read. Thus it is legal to write, say,

```

param cost {ORIG,DEST} default 9999;
param ic symbolic in ORIG;
param jc symbolic in DEST;
param npairs integer;

read npairs, {1..npairs} (ic,jc,cost[ic,jc]);

```

The first input value, by being assigned to `npairs`, determines how much more data will be read. The remainder of the input comes in a series of threes; the first two values, read into `ic` and `jc`, determine where in `cost` the third value should be stored.

Sources of input. If no *redirection* is specified, values are read from the current input stream. Thus if you have typed the `read` command at an AMPL prompt, you type the values at subsequent prompts until all of the *arglist* entries have been assigned values. For example:

```

ampl: read npairs, {1..npairs} (ic,jc,cost[ic,jc]);
ampl? 3
ampl? GARY FRA 3000
ampl? GARY LAF 4500
ampl? CLEV WIN 4200

ampl: display cost;
cost :=
CLEV WIN    4200
GARY FRA    3000
GARY LAF    4500
;

```

The prompt changes from `ampl?` back to `ampl:` when all the needed input has been read. If instead you put `read` inside an AMPL script file that is read by use of `include` or `commands`, then the input is read from the same file, beginning directly after the `;` of the `read` command.

Most often the input to `read` lies in a separate file. Then you can use the optional *redirection* to specify this file; its form is `< filename`, where *filename* is a string that identifies a file on your computer. You can read more than once from the same file,

```

ampl: read npairs < pairs.dat;
ampl: read {1..npairs} (ic,jc,cost[ic,jc]) < pairs.dat;

```

in which case each `read` starts reading the file where the previous one left off. (To force reading to start at the beginning again, use the command `close filename`.)

What if you want an AMPL script to contain a `read` command that reads values typed interactively? In this case you must redirect the source of the values to the "standard input", which is accomplished by writing a `-` as the *filename*. This can be useful when you want a script to prompt the user. For example, suppose that your script contains

```
param T integer 0;

printf "\nHow many of the periods do you want to use?\n";
read T
```

and is stored in the file `read.run`. Then here's how it would look in use:

```
ampl: include read.run;

How many of the periods do you want to use?
ampl? 5
```

In this case the value 5 would be assigned to `T`.

2.3 Removing or redefining model components

The `delete` command removes a previously declared model component, provided that no other components use it in their declarations. Thus normally you can delete a constraint, but you cannot delete a variable -- because it appears in subsequent constraint declarations. The form of the command is

```
delete component-list ;
```

where *component-list* is a space-separated or comma-separated list of names of sets, parameters, variables, objectives, constraints or problems. You can also include a "name" of the form *check n* in the *component-list*, to delete the *n*th *check* statement in the current model.

The `purge` command has the same form,

```
purge component-list ;
```

It removes not only the named components, but also all components that *depend* on them either directly (by referring to them) or indirectly (by referring to their dependents). Thus for example in `diet.mod` we have

```
param f_min {FOOD} = 0;
param f_max {j in FOOD} = f_min[j];

var Buy {j in FOOD} = f_min[j], <= f_max[j];

minimize total_cost: sum {j in FOOD} cost[j] * Buy[j];
```

The command `purge f_min` deletes parameter `f_min` and the components whose declarations refer to `f_min`, including parameter `f_max` and variable `Buy`. It also

deletes objective `total_cost`, which depends indirectly on `f_min` through its reference to `Buy`.

Once a component has been removed by `delete` or `purge`, any previously hidden meaning of the component's name becomes visible again. After a constraint named `prod` is deleted, for instance, AMPL again recognizes `prod` as an iterated multiplication operator (Table 7-1 of the AMPL book).

The name of a component removed by `delete` or `purge` becomes again unused, and may subsequently be declared as the name of any new component of any type. If you only want to make some relatively small revision to a component's declaration, however, then you will probably find AMPL's new `redeclare` feature to be more convenient. You can say

```
redeclare declaration
```

where *declaration* is the complete revised declaration that you would like to substitute. Looking again at `diet.mod`, for example,

```
redeclare param f_min {FOOD} 0 integer;
```

changes only the validity conditions on `f_min`. The declarations of all components that depend on `f_min` are left unchanged, as are any values previously read for `f_min`. A `redeclare` can be applied to statements beginning with any of the following:

```
set
param
var
minimize
maximize
subject to
node
arc
problem
```

You can also redeclare the *n*th check statement by writing `redeclare check n` in front of your new check declaration.

To request a list of all components that a given component refers to, use the new [xref](#) command.

2.4 Fixing or unfixing a variable at a new value

Rather than setting a variable to zero and then fixing it to zero,

```
let Buy["beef"] := 0;
fix Buy["beef"];
```

you can fix and set the variable with one command:

```
fix Buy["beef"] := 0;
```

The `:=` phrase may also be used with the indexed form of the `fix` command:

```
fix {i in ORIG} Trans[i,"FRA"] := 100;
```

The `unfix` command works in the same way, to simultaneously unfix and reset the value of a variable.

2.5 Relaxing integrality

By changing the `relax_integrality` option from its default of 0 to any nonzero value,

```
option relax_integrality 1;
```

you tell AMPL to drop any restriction of variables to integer values. Variables declared `integer` get whatever bounds you specified for them, while variables declared `binary` are given a lower bound of zero and an upper bound of one. To restore the integrality restrictions, set the `relax_integrality` option back to 0.

Some of the solvers that work with AMPL provide their own directives for relaxing integrality, but these do not necessarily have the same effect as the AMPL's `relax` option. AMPL drops integrality restrictions *before* its presolve phase, so that the solver receives a true continuous relaxation of the original integer problem. If the relaxation is performed by the solver, however, then the integrality restrictions are still in effect during AMPL's presolve phase, and AMPL may perform some additional tightening and simplification as a result. As a simple example, when presolve sees the declarations

```
var X integer = 0;  
subj to upXbd: 5 * X <= 12;
```

it removes the constraint `upXbd` and places an upper limit of 2 on the variable `X`; this upper limit is sent to the solver, where it remains even if you specify a solver directive for integrality relaxation. If instead option `relax_integrality` is set to 1, AMPL converts the constraint to an upper limit of 2.4, which is sent to the solver.

The same situation can arise in much less obvious circumstances, and can lead to unexpected results. In general, the optimal value of an integer program under AMPL's `relax_integrality` option may be lower (for minimization) or higher (for maximization) than the optimal value reported by the solver's relaxation directive, unless AMPL's presolve phase is turned off by the command `option presolve 0`.

3 Relational Database Access

The structure of indexed data in AMPL has much in common with the structure of the relational tables widely used in database applications. The new AMPL **table** declaration lets you take advantage of this similarity to define explicit connections between sets, parameters, variables, and expressions in AMPL, and relational database tables maintained by other software. New **read table** and **write table** commands subsequently use these connections to import data values into AMPL and to export data and solution values from AMPL.

The relational tables read and written by AMPL reside in files whose names and locations you specify as part of the table declaration. To work with these files, AMPL relies on database *handlers*, which are add-ons that can be loaded as needed. Handlers may be made available by the vendors of solvers or of database software.

The initial part of this presentation is tutorial in nature. We begin by describing, through a series of examples, how AMPL entities can be put into correspondence with the columns of relational tables. We then revisit these examples to show how the same correspondences can be described and implemented by use of AMPL's table declaration.

Subsequent sections describe the use of the table declaration in detail. Following a description of the overall form of the declaration, we separately present and illustrate the alternatives for reading and for writing external relational tables; then we describe some further complications that arise when reading and writing the same table. Finally, we present extensions for working with indexed collections of tables or table columns, giving examples for automatically writing a series of tables or columns and for reading "two-dimensional" tables of spreadsheet data.

We conclude by providing detailed instructions for the standard and built-in handlers used in our examples. The standard handler supports packages, including Microsoft Excel and Access, that can communicate via the Open Database Connectivity (ODBC) standard under Windows. The built-in handlers support simple ASCII and binary table formats intended mainly for debugging and demonstration purposes.

The facility described here is a part of the standard, command-line version of AMPL. Thus it is independent of the database facilities provided by the AMPL Plus graphical user interface for AMPL under Windows.

3.1 General principles of data correspondence

We begin by explaining how several similarly-indexed AMPL parameters, variables, or expressions can be put into correspondence with one relational table. The AMPL declarations and commands to define and use these correspondences will then be introduced in the remaining sections.

As a simple first example, consider the following declarations from `diet.mod` in Chapter 1 of the AMPL book, defining the set `FOOD` and three parameters indexed over it:

```
set FOOD;
param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];
```

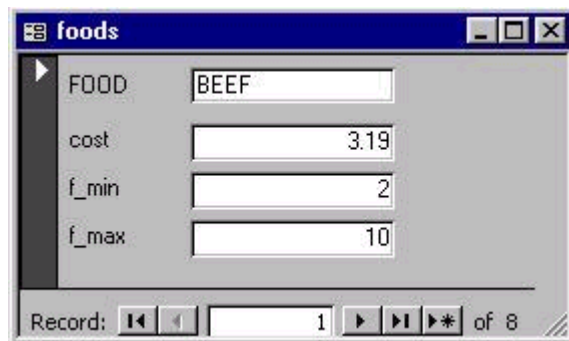
A *relational table* giving values for these components can be regarded as being laid out in the following form:

FOOD	cost	f_min	f_max
BEEF	3.19	0	100
CHK	2.59	0	100
FISH	2.29	0	100
HAM	2.89	0	100
MCH	1.89	0	100
MTL	1.99	0	100
SPG	1.99	0	100
TUR	2.49	0	100

There are 4 *columns* in this table. The column headed `FOOD` lists the members of the AMPL set also named `FOOD`. This is the table's *key* column; entries in a key column must be unique, like a set's members. Next the column headed `cost` gives the values of the like-named parameter indexed over set `FOOD`; here the value of `cost["BEEF"]` is specified as 3.19, `cost["CHK"]` as 2.59, and so forth. The remaining two columns similarly give values for the other two parameters indexed over `FOOD`.

The table has 8 *rows* of data, one for each set member. Thus each row contains all of the table's data corresponding to one member -- one food, in this example.

In the context of database software, the table rows are often viewed as data *records*, and the columns as *fields* within each record. Thus a data entry form has one entry field for each column. A form for the diet example (from Microsoft Access) might look like this:



Data records, one for each table row, can be entered or viewed one at a time by use of the controls at the bottom of the form.

Parameters are not the only entities of interest indexed over the set FOOD in this example. There are also the variables,

```
var Buy {j in FOOD} >= f_min[j], <= f_max[j];
```

and assorted result expressions that may be displayed:

```
ampl: model diet.mod;
ampl: data diet2a.dat;
ampl: solve;
```

```
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032
```

```
ampl: display Buy, Buy.rc, {j in FOOD} Buy[j]/f_max[j];
```

```

:      Buy      Buy.rc      Buy[j]/f_max[j]      :=
BEEF   5.36061   8.88178e-16   0.536061
CHK     2        1.18884      0.2
FISH    2        1.14441      0.2
HAM     10       -0.302651     1
MCH     10       -0.551151     1
MTL     10       -1.3289      1
SPG     9.30605   0           0.930605
TUR     2        2.73162      0.2
;
```

All of these can be included in the relational table for values indexed over FOOD:

FOOD	cost	f_min	f_max	Buy	BuyRC	BuyFrac
BEEF	3.19	0	100	5.36061	8.88178e-16	0.536061
CHK	2.59	0	100	2	1.18884	0.2
FISH	2.29	0	100	2	1.14441	0.2
HAM	2.89	0	100	10	-0.302651	1
MCH	1.89	0	100	10	-0.551151	1
MTL	1.99	0	100	10	-1.3289	1
SPG	1.99	0	100	9.30605	0	0.930605
TUR	2.49	0	100	2	2.73162	0.2

Whereas the first 4 columns would typically be read into AMPL from a database, the last 3 are results that would be written back from AMPL to the database. We have invented the column headings BuyRC and BuyFrac, because the AMPL expressions for the quantities in those columns are typically not valid column heading in database management systems. The AMPL table declaration provides for input/output and naming distinctions such as these, as subsequent sections will show.

Other entities of diet.mod are indexed over the set NUTR of nutrients: parameters n_min and n_max, dual prices and other values associated with constraint diet, and expressions involving these. Since nutrients are entirely distinct from foods, however, the values indexed over nutrients go into a separate relational table from the one for foods discussed above. It might look like this:

NUTR	n_min	n_max	NutrDual
A	700	20000	0

B1	700	20000	0
B2	700	20000	0.404585
C	700	20000	0
CAL	16000	24000	0
NA	0	50000	-0.00306905

As this example suggests, any AMPL model having more than one indexing set will require more than one relational table to properly hold its data and results. Databases that consist of multiple tables are a standard feature of relational data management, to be found in all but the simplest "flat file" database packages.

AMPL entities indexed over the same higher-dimensional set have a similar correspondence to a relational table, but with one key column for each dimension. In the case of `steelT.mod`, for example, the following parameters and variables are indexed over the same two-dimensional set of product-time pairs:

```

set PROD;      # products
param T > 0;    # number of weeks

param market {PROD,1..T} >= 0;
param revenue {PROD,1..T} >= 0;

var Make {PROD,1..T} >= 0;
var Sell {p in PROD, t in 1..T} >= 0, <= market[p,t];

```

A corresponding relational table thus has two key columns -- one containing members of `PROD` and the other members of `1..T` -- and then a column of values for each parameter and variable. Here's an example, corresponding to the data in `steelT.dat`:

PROD	TIME	market	revenue	Make	Sell
bands	1	6000	25	5990	6000
bands	2	6000	26	6000	6000
bands	3	4000	27	1400	1400
bands	4	6500	27	2000	2000
coils	1	4000	30	1407	307
coils	2	2500	35	1400	2500
coils	3	3500	37	3500	3500
coils	4	4200	39	4200	4200

Each ordered pair of items in the two key columns is unique in this table, just as these pairs are unique in the set `{PROD,1..T}`. Thus the `market` column of the table implies, for example, that `market["bands",1]` is 6000 and that `market["coils",3]` is 3500. Reading across the first row, we see also that `revenue["bands",1]` is 25, `Make["bands",1]` is 5990, and `Sell["bands",1]` is 6000. Again various names from the AMPL model are used as column headings, except for `TIME`, which must be invented to stand for the expression `1..T`. As in the previous example, the column headings can be any identifiers acceptable to the database software, and the `table` declaration will take care of the correspondences (in a manner to be explained below).

AMPL entities that have sufficiently similar indexing generally fit into the same relational table. We could extend our `steelT.mod` table, for instance, by adding a column for values of

```

var Inv {PROD,0..T} >= 0;

```

The table would then have the following layout:

PROD	TIME	market	revenue	Make	Sell	Inv
------	------	--------	---------	------	------	-----

bands	0	10
bands	1	6000	25	5990	6000	0
bands	2	6000	26	6000	6000	0
bands	3	4000	27	1400	1400	0
bands	4	6500	27	2000	2000	0
coils	0	0
coils	1	4000	30	1407	307	1100
coils	2	2500	35	1400	2500	0
coils	3	3500	37	3500	3500	0
coils	4	4200	39	4200	4200	0

We use a "." here to mark table entries that correspond to values not defined by the model and data. There is no `market["bands", 0]` in the data for this model, for example, although there does exist a value for `Inv["bands", 0]` in the results. Database software packages can vary somewhat in the handling of "missing" entries of this sort.

Parameters and variables may also be indexed over a set of pairs that is read as data rather than being constructed from one-dimensional sets. For instance, in `transp3.mod` we have:

```
set LINKS within {ORIG,DEST};

param cost {LINKS} >= 0;    # shipment costs per unit
var Trans {LINKS} >= 0;    # actual units to be shipped
```

A corresponding relational table has two key columns corresponding to the two components of the indexing set `LINKS`, plus a column each for the parameter and variable that are indexed over `LINKS`:

ORIG	DEST	cost	Trans
GARY	DET	14	0
GARY	LAF	8	600
GARY	LAN	11	0
GARY	STL	16	800
CLEV	DET	9	1200
CLEV	FRA	27	0
CLEV	LAF	17	400
CLEV	LAN	12	600
CLEV	STL	26	0
CLEV	WIN	9	400
PITT	FRA	24	900
PITT	FRE	99	1100
PITT	STL	28	900
PITT	WIN	13	0

The structure here is the same as in our previous example. There is a row in the table only for each origin-destination pair that is actually in the set `LINKS`, however, rather than for every possible origin-destination pair.

3.2 Examples of AMPL table-handling statements

We begin our presentation of AMPL's statements for relational tables by presenting some examples that could be used with the tables defined above. All of the features shown in these examples are explained in much more detail in the sections following.

To transfer information between an AMPL model and a relational table, you begin with a `table` declaration that establishes the correspondence between them. Certain details of this declaration depend on the software being used to create and maintain the table. In the case of the 4-column table of diet data defined above, some of the possibilities are as follows:

For a Microsoft Access table in a database file `diet.mdb`,

```
table Foods IN "ODBC" "diet.mdb": FOOD <- [FOOD], cost, f_min,
f_max;
```

For a Microsoft Excel range from a workbook file `diet.xls`,

```
table Foods IN "ODBC" "diet.xls": FOOD <- [FOOD], cost, f_min,
f_max;
```

For an ASCII text table in file `Foods.tab`,

```
table Foods IN: FOOD <- [FOOD], cost, f_min, f_max;
```

Each `table` declaration has two parts. Before the colon, the declaration provides general information. First comes the table name -- `Foods` in our examples above -- which will be the name by which the table is known within AMPL. The keyword `IN` states that the default for all non-key table columns will be read-only; AMPL will read values *in* from these columns and will not write out to them. Details for locating the table in an external database file are provided by the character strings such as `"ODBC"` and `"diet.mdb"`, with the AMPL table name (`Foods`) providing a default where needed:

For Microsoft Access, the table is to be read from database file `diet.mdb` using AMPL's ODBC handler. The table's name within the database file is taken to be `Foods` by default.

For Microsoft Excel, the table is to be read from spreadsheet file `diet.xls` using AMPL's ODBC handler. The spreadsheet range containing the table is taken to be `Foods` by default.

Where no details are given, the table is read by default from the ASCII text file `Foods.tab` using AMPL's built-in text table handler.

In general, the format of the character strings in the `table` declaration depends upon the database handler being used. The strings required by the handlers used in our examples are explained in the section on standard and built-in table handlers at the end of the writeup.

After the colon, the `table` declaration gives the details of the correspondence between AMPL entities and relational table columns. The four comma-separated entries correspond to four columns in the table, starting with the key column distinguished by surrounding `[. . .]`. In this example, the names of the table's columns -- `FOOD`, `cost`, `f_min`, `f_max` -- are the same as the names of the corresponding AMPL components. The expression `FOOD <- [FOOD]` indicates that the entries in the key column `FOOD` are to be copied into AMPL to define the members of the set `FOOD`.

The table declaration only defines a correspondence. To actually read values from columns of a relational table into AMPL sets and parameters, it is necessary to give an explicit read table command. Thus, if the data values were in a Microsoft Access relational table like this,

FOOD	cost	f_min	f_max
BEEF	3.19	2	10
CHK	2.59	2	10
FISH	2.29	2	10
HAM	2.89	2	10
MCH	1.89	2	10
MTL	1.99	2	10
SPG	1.99	2	10
TUR	2.49	2	10

then the previously shown table declaration for Access could be used together with the read table command to read the members of FOOD and values of cost, f_min and f_max into the corresponding AMPL set and parameters:

```

ampl: model diet.mod;

ampl: table Foods IN "ODBC" "diet.mdb": FOOD <- [FOOD], cost,
f_min, f_max;
ampl: read table Foods;

ampl: display cost, f_min, f_max;

:      cost f_min f_max      :=
BEEF   3.19    2    10
CHK     2.59    2    10
FISH    2.29    2    10
HAM     2.89    2    10
MCH     1.89    2    10
MTL     1.99    2    10
SPG     1.99    2    10
TUR     2.49    2    10
;

```

(The display command is shown here just to confirm that the database values were read as intended.) If the data values were instead in a Microsoft Excel worksheet range like this,

	A	B	C	D	E	F	G
1							
2		FOOD	f_min	f_max	cost		
3		BEEF	2	10	3.19		
4		CHK	2	10	2.59		
5		FISH	2	10	2.29		
6		HAM	2	10	2.89		
7		MCH	2	10	1.89		
8		MTL	2	10	1.99		
9		SPG	2	10	1.99		
10		TUR	2	10	2.49		
11							
12							

then the values would be read in the same way, but using the previously shown table declaration for Excel:

```
ampl: model diet.mod;

ampl: table Foods IN "ODBC" "diet.xls": FOOD <- [FOOD], cost,
f_min, f_max;
ampl: read table Foods;
```

And if the values were in a file `Foods.tab` containing a text table like this,

`ampl.tab 1 3`

FOOD	cost	f_min	f_max
BEEF	3.19	2	10
CHK	2.59	2	10
FISH	2.29	2	10
HAM	2.89	2	10
MCH	1.89	2	10
MTL	1.99	2	10
SPG	1.99	2	10
TUR	2.49	2	10

then the previously shown declaration for a text table would be used:

```
ampl: model diet.mod;

ampl: table Foods IN: FOOD <- [FOOD], cost, f_min, f_max;
```

```
ampl: read table Foods;
```

Because the AMPL table name is the same -- `Foods` -- in all three of these examples, the `read table` command is the same for all three: `read table Foods`. In general, the `read table` command only specifies the AMPL name of the table to be read. All information about what is to be read, and how it is to be handled, is taken from the named table's definition in the preceding table declaration.

To create the second (7-column) relational table example of the previous section, we could use a pair of table declarations,

```
table ImportFoods IN "ODBC" "diet.mdb" "Foods":  
    FOOD <- [FOOD], cost, f_min, f_max;  
  
table ExportFoods OUT "ODBC" "diet.mdb" "Foods": FOOD <- [FOOD],  
    Buy, Buy.rc ~ BuyRC, {j in FOOD} Buy[j]/f_max[j] ~ BuyFrac;
```

or a single table declaration combining the input and output information:

```
table Foods "ODBC" "diet.mdb": [FOOD] IN, cost IN, f_min IN,  
    f_max IN, Buy OUT, Buy.rc ~ BuyRC OUT, {j in FOOD}  
    Buy[j]/f_max[j] ~ BuyFrac OUT;
```

These examples show how the AMPL table name (such as `ExportFoods`) may be different from the name of the corresponding table within the external file (as indicated by the subsequent string `"Foods"`). A number of other useful options are also seen here:

`IN` and `OUT` are associated with individual columns of the table, rather than with the whole table.

`[FOOD] IN` is used as an abbreviation for `FOOD <- [FOOD]`.

Columns of the table are associated with the values of variables `Buy` and expressions `Buy.rc` and `Buy[j]/f_max[j]`.

`Buy.rc ~ BuyRC` and `{j in FOOD} Buy[j]/f_max[j] ~ BuyFrac` associate an AMPL expression (to the left of the `~` operator) with a database column heading (to the right).

To write meaningful results back to the Access database, we would need to read all of the diet model's data, then solve, and then give a `write data` command. Here's how it all might look using the separate table declarations to read and write the Access table `Foods`,

```
ampl: model diet.mod;  
  
ampl: table ImportFoods IN "ODBC" "diet.mdb" "Foods":  
ampl?    FOOD <- [FOOD], cost, f_min, f_max;  
  
ampl: table Nutrs IN "ODBC" "diet.mdb": NUTR <- [NUTR], n_min,  
n_max;  
ampl: table Amts IN "ODBC" "diet.mdb": [NUTR, FOOD], amt;  
  
ampl: read table ImportFoods;
```

```

ampl: read table Nutrs;
ampl: read table AmtS;

ampl: solve;

ampl: table ExportFoods OUT "ODBC" "diet.mdb" "Foods": FOOD <-
[FOOD],
ampl?    Buy, Buy.rc ~ BuyRC, {j in FOOD} Buy[j]/f_max[j] ~
BuyFrac;

ampl: write table ExportFoods;

```

and here is an alternative using a single declaration to both read and write Foods:

```

ampl: model diet.mod;

ampl: table Foods "ODBC" "diet.mdb":
ampl?    [FOOD] IN, cost IN, f_min IN, f_max IN,
ampl?    Buy OUT, Buy.rc ~ BuyRC OUT,
ampl?    {j in FOOD} Buy[j]/f_max[j] ~ BuyFrac OUT;

ampl: table Nutrs IN "ODBC" "diet.mdb": NUTR <- [NUTR], n_min,
n_max;
ampl: table AmtS IN "ODBC" "diet.mdb": [NUTR, FOOD], amt;

ampl: read table Foods;
ampl: read table Nutrs;
ampl: read table AmtS;

ampl: solve;

ampl: write table Foods;

```

Either way, the Access table Foods would end up having three additional columns:

Foods : Table							
	FOOD	cost	f_min	f_max	Buy	BuyRC	BuyFrac
▶	BEEF	3.19	2	10	5.360613811	3.05311E-16	0.536061381
	CHK	2.59	2	10	2	1.18884058	0.2
	FISH	2.29	2	10	2	1.144407502	0.2
	HAM	2.89	2	10	10	-0.30265132	1
	MCH	1.89	2	10	10	-0.5511509	1
	MTL	1.99	2	10	10	-1.32890026	1
	SPG	1.99	2	10	9.306052856	-2.0123E-15	0.930605286
	TUR	2.49	2	10	2	2.731619778	0.2
*							

Record: 1 of 8

The same operations are handled similarly for other types of database files. In general, the actions of a write table command are determined by the previously declared AMPL table named in the command, and by the status of the external file associated with

the AMPL table through its `table` declaration. In certain circumstances, the `write table` command may create a new external file or table, overwrite an existing table, overwrite certain columns within an existing table, or append columns to an existing table. Details may vary somewhat depending on the database handler and the file type; see for example the explanations in the section on standard and built-in table handlers at the end of this writeup.

The `table` declaration is the same for multidimensional AMPL entities, except that there must be more than one key column specified between the brackets `[` and `]`. For the steel production example discussed previously, the correspondence to a relational table could be set up like this:

```
table steelprod "ODBC" "steel.mdb":
    [PROD, TIME], market IN, revenue IN, Make OUT, Sell OUT, Inv
    OUT;
```

Here the key columns `PROD` and `TIME` are not specified as `IN`. This is because the parameters to be read in `-- market` and `revenue --` are indexed in the AMPL model over the set `{PROD, 1..T}`, whose membership would be specified by use of other, simpler tables. The `read table steelprod` command merely uses the `PROD` and `TIME` entries of each database row to determine the pair of indices (subscripts) that are to be associated with the `market` and `revenue` entries in the row.

Our transportation example also involves a relational table for two-dimensional entities, and the associated `table` declaration is similar:

```
table transLinks "ODBC" "trans.xls" "Links":
    LINKS <- [ORIG, DEST], cost IN, Trans OUT;
```

The difference here is that `LINKS`, the AMPL set of pairs over which `cost` and `Trans` are indexed, is part of the data rather than being determined from simpler sets or parameters. Thus we write `LINKS <- [ORIG, DEST]`, to request that pairs from the key columns be read into `LINKS` at the same time that the corresponding values are read into `cost`. This distinction is discussed further in the section on reading data from relational tables below.

A model's table declarations and `read table` and `write table` commands are normally used in an AMPL script, rather than being typed interactively. There is typically one `table` declaration for each set that indexes parameters to be read or variables and expressions to be written (or both). Complete sample scripts and Access or Excel files for our diet, production, and transportation examples can be accessed as shown in Figure 1 below.

Model	Scripts	Data files	Notes
diet.mod	Diet.mdb.run diet.xls.run	diet.mdb diet.xls	
	Diet.mdb2.run diet.xls2.run	diet.mdb diet.xls	Same, but using separate table declarations for reading and writing.

steelT.mod	Steel.mdb.run steel.xls.run	steel.mdb steel.xls	Reads an unindexed parameter from a database. Writes variables having slightly different indexing sets.
transp3.mod	Trans.mdb.run trans.xls.run	trans.mdb trans.xls	Reads an AMPL set of ordered pairs from a database.

Figure 1. Examples of AMPL scripts that use `table`, `read table`, and `write table` to exchange data with relational tables in external files.

3.3 General forms of the `table` declaration

To use AMPL's relational database access features, you must supply one `table` declaration for each different relational table to be read or written (or both). This and the following four sections describe aspects of the `table` declaration that are intended to be independent of external software, while the final section describes aspects of AMPL's standard implementation for Microsoft Access and Excel (and other Windows software accessible via ODBC) and for "plain" text and binary files.

If you are just getting started, you may want to skim this section and then look through the following two sections -- Reading data from relational tables and Writing data to relational tables -- for examples that can be adapted to your situation. It is usually easiest to start by doing the reading and writing separately, with separate `table` declarations, but if you will frequently be doing both then you may also want to look at the section entitled Reading and writing the same table.

In the most general terms, the syntax of the `table` declaration is

```
table table-name {indexing-expr}opt inoutopt string-listopt :
key-spec, data-spec, data-spec, data-spec, ... ;
```

The part before the colon (:) deals with the relational table as a whole, while the part after establishes correspondences with particular columns of the table.

The *table-name* is the name by which a relational table is known within an AMPL model. This name is used to identify the table in subsequent `read table` and `write table` commands.

The optional {*indexing-expr*} specifies an indexed collection of tables, in the same way that other indexed collections of entities are specified in AMPL. Since unindexed tables are expected to be much the more common case, we initially assume that no indexing is specified. The modifications necessary for indexing are described later in the section entitled Indexed collections of tables and columns.

An *inout* keyword specifies a default read/write setting for the table's data (non-key) columns. The recognized keywords and their interpretations are:

```
IN contents to be read in to AMPL from a relational table
OUT contents to be written out from AMPL to a relational table
INOUT contents to be both read and written
```

When no *inout* keyword is specified, INOUT is assumed.

The *string-list* is a sequence of AMPL character strings that specify where the external relational table is located and what external software is used to access it. The interpretation of this information is specific to the software used, but typically the first string identifies a *handler* for the AMPL/table interface, the second identifies a file to be read or written, and the third identifies a table within the file; see the final section (Standard and built-in table handlers) below for some examples. Omitted strings are typically given default values derived from the *table-name*. If the *string-list* is omitted entirely, a file called *table-name*.tab containing a single text table is assumed.

The *key-spec* associates key columns of an external relational table with AMPL sets. Its general syntactic form is one of

```
[key-col-spec, key-col-spec, ...] inoutopt
set-expr arrow [key-col-spec, key-col-spec, ...]
```

where the *key-col-spec* is either of

```
key-col-name
index ~ key-col-name
```

Each *key-col-spec* names a key column of the table; the optional *index* is for use in subsequent *data-specs*, where it takes values from the key column. The *set-expr* is an expression for a corresponding AMPL set, whose arity must equal the number of *key-col-specs*. The *inout* keyword or *arrow* symbol (either <-, ->, or <->) indicates whether set members are to be read or written (or both).

Each *data-spec* associates a data column of an external relational table with an AMPL entity, usually a parameter but possibly a variable, a suffixed variable or constraint, or an expression. Its general syntactic form is one of

```
data-col-name inoutopt
data-expr ~ data-col-name inoutopt
```

where *col-name* identifies the data column and the optional *expr* identifies the corresponding AMPL entity.

Detailed rules for forming and interpreting *key-specs* and *data-specs* are given in the following sections. To avoid inessential complications in the initial presentation, the next two sections focus on common cases in which a `table` declaration is used only in connection with reading from a database or writing to a database. Further sections then describe some less common situations, such as the use of a `table` declaration for reading and writing the same table and for defining indexed collections of tables and columns.

3.4 Reading data from relational tables

To use an external relational table for reading only, you should employ a `table` declaration that specifies a read/write status of `IN`. Thus it should have the general form

```
table table-name IN string-listopt :  
key-spec, data-spec, data-spec, data-spec, ... ;
```

where the optional *string-list* is specific to the database type and access method being used. (In the interest of brevity, most subsequent examples do not show a *string-list*.) Data values are subsequently read from the table into AMPL entities by use of the `command`

```
read table table-name ;
```

which determines the values to be read by referring back to the `table` declaration that defined *table-name*.

We begin by describing the two fundamental variants of the `table` declaration, one for reading only parameter values from data columns, and one for reading a set's members from the key columns as well as parameter values from any data columns. We then explain how a correspondence between database columns and AMPL parameters may be established in several common situations where the naming or organization of columns does not precisely match that of the parameters. Finally, we identify additional kinds of values -- relating to variables and constraints -- that can be read from relational tables in much the same ways as parameter values.

Reading parameters only. To assign values from data columns to like-named AMPL parameters, it suffices to give a bracketed list of key columns and then a list of data columns.

The simplest case, where there is only one key column, is exemplified by

```
table Foods IN: [FOOD], cost, f_min, f_max;
```

This indicates that the relational table has 4 columns, comprising a key column `FOOD` and data columns `cost`, `f_min` and `f_max`. The data columns are associated with parameters `cost`, `f_min` and `f_max` in the current AMPL model. Since there is only one key column, all of these parameters must be indexed over one-dimensional sets.

When `read table Foods` is executed, the relational table is read one row at a time. A row's entry in the key column is interpreted as a subscript to each of the parameters, and these subscripted parameters are assigned the row's entries from the associated data columns. For example, if the relational table is

FOOD	cost	f_min	f_max
BEEF	3.19	0	100
CHK	2.59	0	100
FISH	2.29	0	100
HAM	2.89	0	100
MCH	1.89	0	100
MTL	1.99	0	100
SPG	1.99	0	100
TUR	2.49	0	100

then the processing of the first row assigns 3.19 to parameter `cost['BEEF']`, 0 to `f_min['BEEF']`, and 100 to `f_max['BEEF']`; the processing of the second row assigns 2.59 to parameter `cost['CHK']`, 0 to `f_min['CHK']`, and 100 to `f_max['CHK']`; and so forth through the 6 remaining rows.

At the time that the `read table` command is executed, AMPL makes no assumption as to how the parameters are declared; they need not be indexed over a set actually named `FOOD`, and indeed the members of their indexing sets may not yet even be known. Only later, when AMPL first uses each parameter in some computation, does it actually check the entries read from key column `FOOD` to be sure that each is a valid subscript for that parameter.

The situation is analogous for multidimensional parameters. The name of each data column must also be the name of an AMPL parameter, and the dimension of the parameter's indexing set must equal the number of key columns. Thus, for example, when two key columns are listed within the brackets,

```
table SteelProd IN: [PROD, TIME], market, revenue;
```

the listed data columns, `market` and `revenue`, must correspond to AMPL parameters `market` and `revenue` that are indexed over two-dimensional sets.

When `read table SteelProd` is executed, each row's entries in the key columns are interpreted as a pair of subscripts to each of the parameters. Thus if the relational table has contents

PROD	TIME	market	revenue
bands	1	6000	25
bands	2	6000	26
bands	3	4000	27
bands	4	6500	27
coils	1	4000	30
coils	2	2500	35
coils	3	3500	37
coils	4	4200	39

then the processing of the first row assigns 6000 to `market['bands' , 1]` and 25 to `revenue['bands' , 1]`; the processing of the second row assigns 6000 to `market['bands' , 2]` and 26 to `revenue['bands' , 2]`; and so forth through all 8 rows. The pairs of subscripts given by the key column entries must be valid for `market` and `revenue` when the values of these parameters are first needed by AMPL, but the parameters need not be declared over sets named `PROD` and `TIME`. (In fact, in the model from which this example is taken, the parameters are indexed by `{PROD, 1..T}` where `T` is a previously defined parameter.)

Since a relational table has only one collection of key columns, it applies the same subscripting to each of the parameters named by the data columns. These parameters are thus usually indexed over the same set. Parameters indexed over similar sets may also be accommodated in one table, however, by leaving blank any entries in rows corresponding to invalid subscripts. The way in which a blank entry is indicated is specific to the database software being used.

Values of unindexed (scalar) parameters may be supplied by a relational table that has one row. There must be one data column for each parameter to be read, and the single row's value in such a column must be of course the value to be assigned to the associated parameter. Any key columns are ignored, and the list of key columns is left empty in the

corresponding table declaration. To read a value for the parameter `T` that gives the number of periods in `steelT.mod`, for example, the table declaration is

```
table SteelPeriods IN: [], T;
```

and the corresponding relational table has one column, also named `T`, whose one entry is a positive integer.

Reading a set and parameters. It is often convenient to read the members of a set from a table's key column or columns, at the same time that parameters indexed over that set are read from the data columns. To indicate that a set should be read from a table, the *key-spec* in the table declaration is written in the form

```
set-name <- [key-col-spec, key-col-spec, ...]
```

The `<-` symbol is intended as an "arrow" pointing in the direction that the information is moved: from the key columns to the AMPL set.

The simplest case involves reading a one-dimensional set and the parameters indexed over it, as in this example for `diet.mod`:

```
table Foods IN: FOOD <- [FoodName], cost, f_min, f_max;
```

When `read table Foods` is executed, all entries in the key column `FoodName` of the relational table are read into AMPL as members of the set `FOOD`, and the entries in the data columns `cost`, `f_min` and `f_max` will be read into the like-named AMPL parameters as previously described. If the key column is named `FOOD` like the AMPL set, then the appropriate table declaration becomes

```
table Foods IN: FOOD <- [FOOD], cost, f_min, f_max;
```

In this special case only, the *key-spec* can also be written in the abbreviated form `[FOOD] IN`.

An analogous syntax is employed for reading a multidimensional set along with parameters indexed over it. In the case of `transp3.mod`, for instance, the table declaration could be:

```
table TransLinks IN: LINKS <- [ORIG, DEST], cost;
```

When `read table TransLinks` is executed, each row of the table provides a pair of entries from key columns `ORIG` and `DEST`. All such pairs are read into AMPL as members of the 2-dimensional set `LINKS`. Finally, the entries in column `cost` are read into parameter `cost` in the usual way.

As in our previous multidimensional example, the names in brackets need not correspond to sets in the AMPL model. The bracketed names serve only to identify the key columns. The name to the left of the arrow is the only one that must name a previously declared AMPL set; this set must have been declared to have the same dimension or arity, moreover, as the number of key columns.

It makes sense to read the set `LINKS` from a relational table, because `LINKS` is specifically declared in the model in a way that leaves the corresponding data to be read separately:

```
set ORIG;
set DEST;
```

```

set LINKS within {ORIG,DEST};
param cost {LINKS} >= 0;

```

In the similar model `transp2.mod`, by contrast, `LINKS` is defined in terms of two one-dimensional sets,

```

set ORIG;
set DEST;

set LINKS := {ORIG,DEST};
param cost {LINKS} >= 0;

```

and in `transp.mod`, no named two-dimensional set is defined at all:

```

set ORIG;
set DEST;

param cost {ORIG,DEST} >= 0;

```

In these latter cases, a table declaration would still be needed for reading parameter `cost`, but it would not specify the reading of any associated set:

```

table TransLinks IN: [ORIG, DEST], cost;

```

Separate relational tables would instead be used to provide members for the one-dimensional sets `ORIG` and `DEST` and values for the parameters indexed over them.

When a table declaration specifies an AMPL set to be assigned members, its list of *data-specs* may be empty. In that case only the key columns are read, and the only action of read table is to assign the key column values as members of the specified AMPL set.

Establishing correspondences. An AMPL model's set and parameter declarations do not necessarily correspond in all respects to the organization of tables in relevant databases. Where the difference is substantial, it may be necessary to use the database's query language to derive temporary tables that have the structure required by the model -- see for example the use of SQL queries in the section on Standard and built-in table handlers. A number of common, simple differences can be handled directly, however, through features of the table declaration.

Differences in naming are perhaps the most common. A table declaration can associate a data column with a differently named AMPL parameter by use of a *data-spec* of the form *param-name ~ data-col-name*. Thus, for example, if table `Foods` were instead defined by

```

table Foods IN: [FOOD], cost, f_min ~ lowerlim, f_max ~
upperlim;

```

then the AMPL parameters `f_min` and `f_max` would be read from data columns `lowerlim` and `upperlim` in the relational table. (Parameter `cost` would be read from column `cost` as before.)

A similarly generalized form, *index ~ key-col-name*, can be used to associate a kind of dummy index with a key column. This index may then be used in a subscript to the optional *param-name* in one or more *data-specs*. Such an arrangement is useful in a number of situations where the key column entries do not exactly correspond to the subscripts of the parameters that are to receive table values. Here are three common cases:

Where a numbering of some kind in the relational table is systematically different from the corresponding numbering in the AMPL model, a simple expression involving a key column *index* can serve to translate from the one numbering scheme to the other. For example, if time periods were counted from 0 in the relational table data rather than from 1 as in the model, then an adjustment could be made in the table declaration as follows:

```
table SteelProd IN: [p ~ PROD, t ~ TIME],
  market[p,t+1] ~ market, revenue[p,t+1] ~ revenue;
```

Where AMPL parameters have subscripts from the same sets but in different orders, key column *indexes* must be employed to provide a correct index order in all cases. If market is indexed over {PROD, 1..T} but revenue is indexed over {1..T, PROD}, for example, then a table declaration to read values for these two parameters should be written as follows:

```
table SteelProd IN: [p ~ PROD, t ~ TIME],
  market, revenue[t,p] ~ revenue;
```

Where the values for an AMPL parameter are divided among several database columns, key column *indexes* can be employed to describe the values to be found in each column. For instance, if the revenue values are given in one column for "bands" and in another column for "coils", the corresponding table declaration could be written like this:

```
table SteelProd IN: [t ~ TIME],
  revenue["bands",t] ~ revbands, revenue["coils",t] ~
  revcoils;
```

It is tempting to try to shorten declarations of these kinds by dropping the *~ data-col-name*, to produce, say,

```
table SteelProd IN: [p ~ PROD, t ~ TIME], market, revenue[t,p];
# ERROR
```

This will usually be rejected as an error, however, because `revenue[t,p]` is not a valid name for a relational table column in most database software. Instead it is necessary to write `revenue[t,p] ~ revenue` to indicate that the AMPL parameters `revenue[t,p]` receive values from the column `revenue` of the table.

More generally, a *~* synonym will have to be used in any situation where the AMPL expression for the recipient of a column's data is not itself a valid name for a database column. The rules for valid column names tend to be the same as the rules for valid component names in AMPL models, but they can vary in details depending on the database software that is being used to create and maintain the tables.

Reading other values. In a table declaration used for input, an *assignable* AMPL expression may appear anywhere that a parameter name would be allowed. An expression is assignable if it can be assigned a value, such as by placing it on the left side of a `:=` in a `let` command.

Variable names are assignable expressions. Thus a `table` declaration can specify columns of data to be read into variables, for purposes of evaluating a previously stored solution or providing a good initial solution for a solver.

Constraint names are also assignable expressions. Values "read into a constraint" are interpreted as initial dual values for some solvers, such as MINOS.

Any variable or constraint name qualified by an assignable suffix is also an assignable expression. Assignable suffixes include the predefined suffix `.sstatus` as well as any user-defined suffixes. For example, if the diet problem were changed to have integer variables, the following `table` declaration could help to provide useful information for the solver:

```
table Foods IN: FOOD IN, cost, f_min, f_max, Buy, Buy.priority ~
prior;
```

An execution of `read table Foods` would supply members for set `FOOD` and values for parameters `cost`, `f_min` and `f_max` in the usual way, and would also assign initial values and branching priorities to the `Buy` variables.

3.5 Writing data to relational tables

To use an external relational table for writing only, you should employ a `table` declaration that specifies its read/write status to be `OUT`. The general form of such a declaration is

```
table table-name OUT string-listopt :
key-spec, data-spec, data-spec, data-spec, ... ;
```

where the optional *string-list* is specific to the database type and access method being used. (In the interest of brevity, most subsequent examples do not show a *string-list*.) AMPL expression values are subsequently written to the table by use of the command

```
write table table-name ;
```

which determines the information to be written by referring back to the `table` declaration that defined *table-name*.

A `table` declaration for writing specifies an external file and possibly a relational table within that file, either explicitly in the *string-list* or implicitly by default rules. Normally the named file or table is created if it does not exist, or is overwritten otherwise. To specify that certain columns are to be replaced or are to be added to a table, the `table` declaration must incorporate one or more *data-specs* that have read/write status `IN` or `INOUT`, as discussed in Reading and writing the same table. In any case, detailed rules for when files and tables are modified or overwritten depend on the database handler being used; see Standard and built-in table handlers for some examples.

The *key-specs* and *data-specs* of `table` declarations for writing external tables superficially resemble those for reading. The range of AMPL expressions allowed when writing is much broader, however, including essentially all set-valued and numerical-valued expressions. Moreover, whereas the table rows to be read are those of some existing table, the rows to be written must be determined from AMPL expressions in some part of a `table` declaration. Specifically, rows to be written can be inferred either

from the *data-specs*, using the same conventions as in AMPL display commands, or from the *key-spec*. Each of these alternatives employs a characteristic table syntax as described below.

Writing rows inferred from the data-specs: If the *key-spec* is simply a bracketed list of the names of key columns,

```
[key-col-name, key-col-name, ...]
```

then the table declaration works much like the AMPL display command. It determines the external table rows to be written by taking the union of the indexing sets stated or implied in the *data-specs*. The format of the *data-spec* list is the same as in display, except that all of the items listed must be indexed over sets of the same dimension.

In the simplest case, the *data-specs* are the names of model components indexed over the same set:

```
table Foods OUT: [FoodName], f_min, Buy, f_max;
```

When write table Foods is executed, it creates a key column FoodName and data columns f_min, Buy, and f_max. Since the AMPL components corresponding to the data columns are all indexed over the AMPL set FOOD, one row is created for each member of FOOD. In a representative row, a member of FOOD is written to the key column FoodNames, and the values of f_min, Buy, and f_max subscripted by that member are written to the like-named data columns. For the data used in our diet example, the resulting relational table would be:

FoodName	f_min	Buy	f_max
BEEF	2	5.36061	10
CHK	2	2	10
FISH	2	2	10
HAM	2	10	10
MCH	2	10	10
MTL	2	10	10
SPG	2	9.30605	10
TUR	2	2	10

Tables corresponding to higher-dimensional sets are handled analogously, with the number of bracketed key-column names listed in the *key-spec* being equal to the dimension of the items in the *data-spec*. Thus a table containing the results from steelT.mod could be defined as

```
table SteelProd OUT: [PROD, TIME], Make, Sell, Inv;
```

Make and Sell are indexed over {PROD, 1..T}, while Inv is indexed over {PROD, 0..T}. Thus a subsequent write table SteelProd command would produce a table having one row for each member of the union of these sets:

PROD	TIME	Make	Sell	Inv
bands	0	.	.	10
bands	1	5990	6000	0
bands	2	6000	6000	0
bands	3	1400	1400	0
bands	4	2000	2000	0
coils	0	.	.	0
coils	1	1407	307	1100

coils	2	1400	2500	0
coils	3	3500	3500	0
coils	4	4200	4200	0

Two rows are empty in the columns for Make and Sell, because ("bands", 0) and ("coils", 0) are not members of the index sets of Make and Sell. We use a "." here to indicate the empty table entries, but the actual appearance and handling of empty entries will vary depending on the database software being used.

If this form is applied to writing suffixed variable or constraint names, such as the dual and slack values related to the constraint diet,

```
table Nutrs OUT: [Nutrient],
    diet.lslack, diet.ldual, diet.uslack, diet.udual;    # ERROR
```

then a subsequent write table Nutrs command is likely to be rejected, because names having a "dot" in the middle are not valid column names in most databases:

```
ampl: write table Nutrs;
Error executing "write table" command:
Error writing table Nutrs with table handler ampl.odbc:
Column 2's name "diet.lslack" contains non-alphanumeric
character '.'.
```

This situation requires that each AMPL expression be followed by the operator ~ and a corresponding valid column name for use in the relational table:

```
table Nutrs OUT: [Nutrient],
    diet.lslack ~ lb_slack, diet.ldual ~ lb_dual,
    diet.uslack ~ ub_slack, diet.udual ~ ub_dual;
```

This says that the values represented by diet.lslack should be placed in a column named lb_slack, the values represented by diet.ldual should be placed in a column named lb_dual, and so forth. With the table defined in this way, a write table Nutrs command produces the intended relational table:

Nutrient	lb_slack	lb_dual	ub_slack	ub_dual
A	1256.29	0	18043.7	0
B1	336.257	0	18963.7	0
B2	0	0.404585	19300	0
C	982.515	0	18317.5	0
CAL	3794.62	0	4205.38	0
NA	50000	0	0	-0.00306905

The ~ can also be used with unsuffixed names, if it is desired to assign the database column a name different from that of the corresponding AMPL entity.

More general expressions for the values in data columns require the use of dummy indices, in the same way that they are used in the data-list of a display command. Since indexed AMPL expressions are rarely valid column names for a database, they should generally be followed by ~ *data-col-name* to provide a valid name for the corresponding relational table column that is to be written. To write a column servings containing the number of servings of each food to be bought and a column percent giving the amount bought as a percentage of the maximum allowed, for example, the table declaration could be given as either

```
table Purchases OUT: [FoodName],
    Buy ~ servings, {j in FOOD} 100*Buy[j]/f_max[j] ~ percent;
```

or

```
table Purchases OUT: [FoodName],
    {j in FOOD} (Buy[j] ~ servings, 100*Buy[j]/f_max[j] ~
    percent);
```

Either way, since both *data-specs* give expressions indexed over the AMPL set FOOD, the resulting table has one row for each member of that set:

FoodName	servings	percent
BEEF	5.36061	53.6061
CHK	2	20
FISH	2	20
HAM	10	100
MCH	10	100
MTL	10	100
SPG	9.30605	93.0605
TUR	2	20

The expression in a *data-spec* may also use operators like `sum` that define their own dummy indices. Thus a table of total production and sales by period for `steelT.mod` could be specified by

```
table SteelTotal OUT: [TIME],
    {t in 1..T} (sum {p in PROD} Make[p,t] ~ Made,
    sum {p in PROD} Sell[p,t] ~ Sold);
```

As a two-dimensional example, a table of the amounts sold and the fractions of demand met could be specified by

```
table SteelSales OUT: [PROD, TIME],
    Sell, {p in PROD, t in 1..T} Sell[p,t]/market[p,t] ~
    MeetDemand;
```

The resulting external table would have key columns `PROD` and `TIME`, and data columns `Sell` and `MeetDemand`.

Writing rows inferred from the key-spec: An alternative form of `table` declaration specifies that one table row is to be written for each member of an explicitly specified AMPL set. For the declaration to work in this way, the *key-spec* must be written as

```
set-spec -> [key-col-spec, key-col-spec, ...]
```

In contrast to the "arrow" `<-` that points from a key-column list to an AMPL set, indicating values to be read into the set, this form uses an arrow `->` that points from an AMPL set to a key column list, indicating information to be written from the set into the key columns. An explicit expression for the row index set is given by the *set-spec*, which can be any of

```
set-name
set-name[subscript-list]
{ set-expr }
{ index-list in set-expr }
```

where *set-expr* may be any AMPL set-valued expression, and the optional *index-list* specifies dummy indices running over the set. Each *key-col-spec* may be either of

```
key-col-name
index ~ key-col-name
```

where the *index* is an alternative dummy index as explained in the examples below.

The simplest case involves writing a column for each of several model components indexed over the same one-dimensional set, as in this example for `diet.mod`:

```
table Foods OUT: FOOD -> [FoodName], f_min, Buy, f_max;
```

When `write table Foods` is executed, a table row is created for each member of the AMPL set `FOOD`. In that row, the set member is written to the key column `FoodNames`, and the values of `f_min`, `Buy`, and `f_max` subscripted by the set member are written to the like-named data columns. (For the data used in our diet example, the resulting table would be the same as for `FoodName` table given previously in this section.) If the key column has the same name, `FOOD`, as the AMPL set, then the appropriate table declaration becomes

```
table Foods OUT: FOOD -> [FOOD], f_min, Buy, f_max;
```

In this special case only, the key-spec can also be written in the abbreviated form `[FOOD] OUT`.

The use of `~` with AMPL names and suffixed names is governed by the considerations previously described, so that the example of diet slack and dual values would be written

```
table Nutrs OUT: NUTR -> [Nutrient],  
    diet.lslack ~ lb_slack, diet.ldual ~ lb_dual,  
    diet.uslack ~ ub_slack, diet.udual ~ ub_dual;
```

and `write table Nutrs` would give the same table as previously shown.

More general expressions for the values in data columns require the use of dummy indices. Since the rows to be written are determined from the *key-spec*, however, the dummies are also defined there (rather than in the *data-specs* as in the alternative form above). To specify a column containing the amount of a food bought as a percentage the maximum allowed, for example, it is necessary to write `100*Buy[j]/f_max[j]`, which in turn requires that dummy index `j` be defined. The definition may appear either in a *set-spec* of the form `{ index-list in set-expr }`,

```
table Purchases OUT: {j in FOOD} -> [FoodName],  
    Buy[j] ~ servings, 100*Buy[j]/f_max[j] ~ percent;
```

or in a *key-col-spec* of the form `index ~ key-col-name`:

```
table Purchases OUT: FOOD -> [j ~ FoodName],  
    Buy[j] ~ servings, 100*Buy[j]/f_max[j] ~ percent;
```

These two forms are equivalent. Either way, as each row is written, the index `j` takes the value written to the key column, and this value is then used in interpreting the expressions that give the values for the data columns. For our example, the resulting table -- having key column `FoodName` and data columns `servings` and `percent`, is the same as previously shown. Similarly, the previous example of the table `SteelTotal` could be written as either

```
table SteelTotal OUT: {t in 1..T} -> [TIME],  
    sum {p in PROD} Make[p,t] ~ Made,  
    sum {p in PROD} Sell[p,t] ~ Sold;
```

or

```

table SteelTotal OUT: {1..T} -> [t ~ TIME],
    sum {p in PROD} Make[p,t] ~ Made,
    sum {p in PROD} Sell[p,t] ~ Sold;

```

The result will have a key column `TIME` containing the integers 1 through `T`, and data columns `Made` and `Sold` containing the values of the two summations.

Tables corresponding to higher-dimensional sets are handled analogously, with the number of *key-col-specs* listed in brackets being equal to the dimension of the *set-spec*. Thus a table containing the results from `steelT.mod` could be defined as

```

table SteelProd OUT:
    {PROD, 1..T} -> [PROD, TIME], Make, Sell, Inv;

```

and a subsequent `write table steelprod` would produce a table of the form

PROD	TIME	Make	Sell	Inv
bands	1	5990	6000	0
bands	2	6000	6000	0
bands	3	1400	1400	0
bands	4	2000	2000	0
coils	1	1407	307	1100
coils	2	1400	2500	0
coils	3	3500	3500	0
coils	4	4200	4200	0

This result is not quite the same as the table produced by the previous `SteelProd` example, because the rows to be written here correspond explicitly to the members of the AMPL set `{PROD, 1..T}`, rather than being inferred from the indexing sets of `Make`, `Sell`, and `Inv`. In particular, the values of `Inv["bands",0]` and `Inv["coils",0]` do not appear in this table.

The options for dummy indices in higher dimensions are the same as in one dimension. Thus our example `SteelSales` could be written either using dummy indices defined in the *set-spec*,

```

table SteelSales OUT:
    {p in PROD, t in 1..T} -> [PROD, TIME],
    Sell[p,t] ~ sold, Sell[p,t]/market[p,t] ~ met;

```

or with dummy indices added to the *key-col-specs*:

```

table SteelSales OUT:
    {PROD,1..T} -> [p ~ PROD, t ~ TIME],
    Sell[p,t] ~ sold, Sell[p,t]/market[p,t] ~ met;

```

If dummy indices happen to appear in both the *set-spec* and the *key-col-specs*, ones in the *key-col-specs* take precedence.

3.6 Reading and writing the same table

To read data from a relational table and then write results to the same table, you can use a pair of `table` declarations that reference the same file and table names. You may also be able to combine these declarations into one that specifies some columns to be read and others to be written. This section gives examples and instructions for both of these possibilities.

Reading and writing using two *table* declarations. A single external table can be read by use of one `table` declaration and later written by use of another. The two `table` declarations follow the rules for reading and writing, respectively, as previously stated.

In this situation, however, one usually wants `write table` to add or rewrite selected columns, rather than overwriting the entire table. This preference can be communicated to the AMPL table handler by including input as well as output columns in the `table` declaration that is to be used for writing. Columns intended for input to AMPL can be distinguished from those intended for output to the external table by specifying a read/write status column by column (rather than for the table as a whole).

As an example, an external table for `diet.mod` might consist of columns `cost`, `f_min` and `f_max` containing input for the model, and a column `Buy` containing the results. If this is maintained as a Microsoft Access table named `Diet` within a file `Diet.mdb`, then the `table` declaration for reading data into AMPL could be

```
table FoodInput IN "ODBC" "DIET.mdb" "Diet":  
  FOOD <- [FoodNames], cost, f_min, f_max;
```

The corresponding declaration for writing the results would have a different AMPL *table-name* but would refer to the same Access table and file:

```
table FoodOutput "ODBC" "DIET.mdb" "Diet":  
  [FoodNames], cost IN, f_min IN, Buy OUT, f_max IN;
```

When `read table FoodInput` is executed, only the three columns listed in the `table FoodInput` declaration are read; if there is an existing column named `Buy`, it is ignored. Later, when the problem has been solved and `write table FoodOutput` is executed, only the one column that has read/write status `OUT` in the `table FoodOutput` declaration is written to the Access table, while the table's other columns are left unmodified.

Although details may vary with the database software used, the general convention is that overwriting of any existing table or file is intended only when *all* data columns in the `table` declaration have read/write status `OUT`. Selective rewriting or addition of columns is intended otherwise. Thus if our AMPL table for output had been declared

```
table FoodOutput "ODBC" "DIET.mdb" "Diet": [FoodNames], Buy OUT;
```

then all of the data columns in Access table `Diet` would have been deleted by `write table FoodOutput`; but the alternative

```
table FoodOutput "ODBC" "DIET.mdb" "Diet": [FoodNames], Buy INOUT;
```

would have only overwritten the column `Buy`, just as in the example we originally gave, since there is a data column (namely `Buy` itself) that does not have read/write status `OUT`. (In fact `INOUT` could be omitted here, since it is the default for read/write status. Rules for specifying read/write status are summarized at the end of this section.)

Reading and writing using the same *table* declaration. In many cases, all of the information for both reading and writing an external table can be specified in the same `table` declaration:

The *key-spec* may use the arrow `<-` to write contents of the key columns into an AMPL set, `->` to write members of an AMPL set into the key columns, or `<->` to do both.

A *data-spec* may specify read/write status IN for a column that will only be read into AMPL, OUT for a column that will only be written out from AMPL, or INOUT for a column that will be both read and written.

A read table *table-name* command reads only the columns, key or data, that are specified in the declaration of *table-name* as being IN or INOUT. A write table *table-name* command analogously writes to only the columns that are specified as OUT or INOUT.

As an example, the declarations defining FoodInput and FoodOutput above could be replaced by

```
table Foods "ODBC" "DIET.mdb" "Diet":  
  FOOD <- [FoodNames], cost IN, f_min IN, Buy OUT, f_max IN;
```

A read table Foods would then read only from key column FoodNames and data columns cost, f_min and f_max. A later write table Foods would write only to the column Buy.

General rules for specifying read/write status. The keywords IN, OUT, and INOUT can be used to specify the default read/write status of a table or the read/write status of individual data columns.

A table's default read/write status may be specified by a keyword following the *table-name*. It is taken to be INOUT if no keyword is given.

A data column's read/write status may be specified by a keyword following the *data-col-name*. It is taken to be the same as the table's default read/write status if no keyword is given.

Because key columns have a special interpretation, their read/write status is handled separately. For reading, if the *key-spec* has one of the forms

```
set-spec <- [key-col-spec, key-col-spec, ...]  
set-spec <-> [key-col-spec, key-col-spec, ...]
```

then the contents of the key columns are read into the AMPL set *set-spec*. Otherwise, values are not read into any AMPL set.

For writing, if the *key-spec* has one of the forms

```
set-spec -> [key-col-spec, key-col-spec, ...]  
set-spec <-> [key-col-spec, key-col-spec, ...]
```

then the contents of the key columns are written from the AMPL set *set-spec*. Otherwise, the contents of the key columns are written from an AMPL set that is inferred from the *data-specs*, as explained in Writing data to relational tables above.

3.7 Indexed collections of tables and columns

In some circumstances, it is convenient to declare an indexed collection of tables, or to define an indexed collection of data columns within a table. This section explains how indexing of these kinds can be specified within the `table` declaration.

To illustrate indexed collections of tables, we present a script that automatically solves a series of scenarios stored separately. To illustrate indexed collections of columns, we show how a "two-dimensional" spreadsheet table can be read.

Indexed collections of tables. AMPL table declarations can be indexed in much the same way as AMPL sets, parameters, and other model components. An optional `{indexing-expr}` follows the *table-name*:

```
table table-name {indexing-expr}opt inoutopt string-listopt : ...
```

One table is defined for each member of the set specified by the *indexing-expr*. Individual tables in this collection are denoted in the usual way, by appending a bracketed subscript or subscripts to the *table-name*.

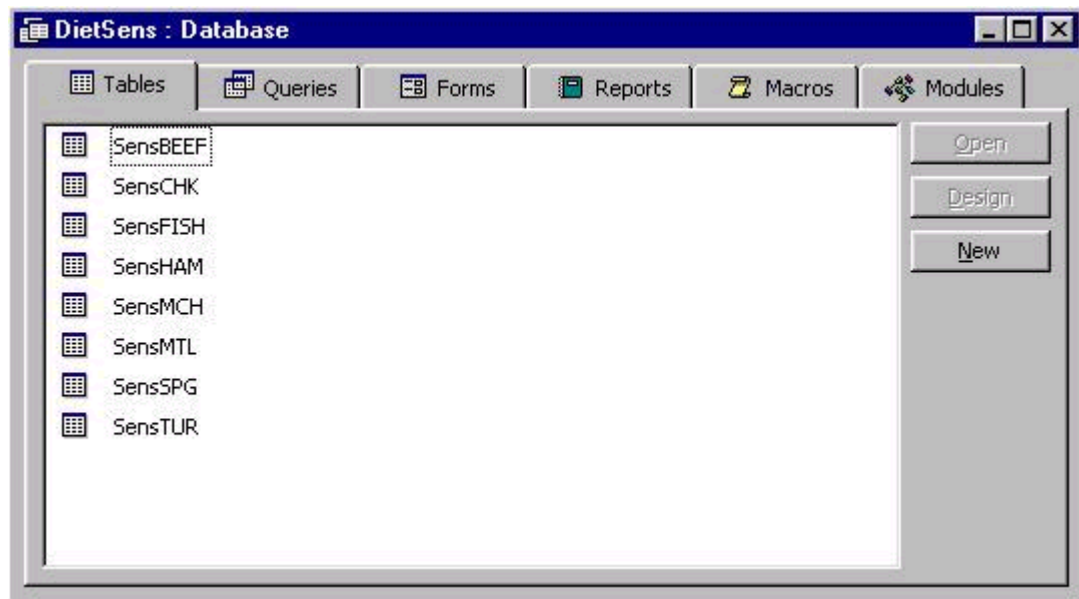
As an example, the following declaration defines a collection of AMPL tables indexed over the set of foods in `diet.mod`, each table corresponding to a different database table in the Microsoft Access file `DietSens.mdb`:

```
table DietSens {j in FOOD} OUT "ODBC" "DietSens.mdb"
("Sens" & j):
    [Food], f_min, Buy, f_max;
```

Following the rules for the standard ODBC table handler, the Access table names are given by the third item in the *string-list*, the AMPL string expression `("Sens" & j)`. Thus the AMPL table `DietSens["BEEF"]` is associated with the Access table `SensBEEF`, the AMPL table `DietSens["CHK"]` is associated with the Access table `SensCHK`, and so forth. The following AMPL script uses these tables to record the optimal diet when there is a two-for-the-price-of-one sale on each of the foods:

```
for {j in FOOD} {
    let cost[j] := cost[j] / 2;
    solve;
    write table DietSens[j];
    let cost[j] := cost[j] * 2;
}
```

For the data in `diet2a.dat`, the set `FOOD` has 8 members, so 8 tables are written in the Access database:



If instead the table declaration were to give a string expression for the second string in the *string-list*, which specifies the Access filename,

```
table DietSens {j in FOOD} OUT "ODBC" ("DietSens" & j & ".mdb"):
  [Food], f_min, Buy, f_max;
```

then 8 different Access database files, named DietSensBEEF.mdb, DietSensCHK.mdb, and so forth would be written, each containing a single table named (by default) DietSens. (These files would all need to have been created before the write table commands were executed.)

A string expression can be used in a similar way to cause each AMPL table to correspond to the same Access table, but with a different *data-col-name* for the optimal amounts:

```
table DietSens {j in FOOD} "ODBC" "DietSens.mdb":
  [Food], Buy ~ ("Buy" & j);
```

Then running the script shown above will result in the following Access table:

DietSens : Table								
	Food	BuyBEEF	BuyCHK	BuyFISH	BuyHAM	BuyMCH	BuyMTL	BuyS
	BEEF	10	8.5681492	6.5677749	5.3606138	5.3606138	5.3606138	5.778
	CHK	2	5.0738881	2	2	2	2	
	FISH	2	2	10	2	2	2	
	HAM	7.0164474	10	10	10	10	10	
	MCH	10	10	10	10	10	10	8.887
▶	MTL	10	10	10	10	10	10	
	SPG	6.6557018	2	2.7655584	9.3060529	9.3060529	9.3060529	
	TUR	2	2	2	2	2	2	
*								

Record: 6 of 8

The AMPL tables in this case were deliberately left with the default read/write status, INOUT. Had the read/write status been specified as OUT, then each write table would have overwritten the columns created by the previous one.

Indexed collections of data columns. Because there is a natural correspondence between data columns of a relational table and indexed collections of entities in an AMPL model, each *data-spec* in a table declaration normally refers to a different AMPL parameter, variable, or expression. Occasionally the values for one AMPL entity are split among multiple data columns, however. Such a case can be handled by defining a collection of data columns, one for each member of a specified indexing set.

The general form for specifying an indexed collection of table columns is

```
{indexing-expr} < data-spec, data-spec, data-spec, ... >
```

where each *data-spec* has any of the forms previously given. For each member of the set specified by the *indexing-expr*, AMPL generates one copy of each *data-spec* within the "angle brackets" <...>. As in other cases of AMPL indexing, the *indexing-expr* also defines one or more dummy indices that run over the index set; these indices are employed in the usual way within AMPL expressions in the *data-specs*, and these indices also appear within string expressions that give the names of columns in the external database.

The most common use of this feature is to read or write "two-dimensional" tables. For example, the data for the parameter

```
param amt {NUTR,FOOD} >= 0;
```

from `diet.mod` might be represented in an Excel spreadsheet as a table with nutrients labeling the rows and foods the columns:

	G	H	I	J	K	L	M	N	O	P
1										
2		NUTR	BEEF	CHK	FISH	HAM	MCH	MTL	SPG	TUR
3		A	60	8	8	40	15	70	25	60
4		B1	10	20	15	35	15	15	25	15
5		B2	15	20	10	10	15	15	15	10
6		C	20	0	10	40	35	30	50	20
7		NA	938	2180	945	278	1182	896	1329	1397
8		CAL	295	770	440	430	315	400	370	450
9										

To read this table using AMPL's external database features, we must regard it as having one key column, under the heading NUTR, and data columns headed by the names of individual foods. Thus we require a table declaration whose *key-spec* is one-dimensional and whose *data-specs* are indexed over the AMPL set FOOD:

```
table dietAmts IN "ODBC" "diet2D.xls":
    [i ~ NUTR], {j in FOOD} <amt[i,j] ~ (j)>;
```

The *key-spec* [i ~ NUTR] associates the first table column with the set NUTR in the standard way. The *data-spec* of the form {j in FOOD} <...> causes AMPL to generate an individual *data-spec* for each of the members of set FOOD. Specifically, for each j in FOOD, AMPL generates the *data-spec* amt[i,j] ~ (j), where (j) is the AMPL string expression for the heading of the external table column for food j, and amt[i,j] denotes the AMPL parameter to which the values in that column are to be written. (According to the convention used here and in other AMPL declarations and commands, the parentheses around (j) cause it to be interpreted as an expression for a string; without the parentheses it would denote a *column-name* consisting of the single character j.)

A similar approach works to write two-dimensional tables to spreadsheets. For example, after steelT.mod is solved, the results could be written to a spreadsheet using the following table declaration,

```
table Results1 OUT "ODBC" "steel2out.xls":
    {p in PROD} -> [Product],
    Inv[p,0] ~ Inv0,
    {t in 1..T} < Make[p,t] ~ ('Make' & t),
    Sell[p,t] ~ ('Sell' & t), Inv[p,t] ~ ('Inv' & t) >;
```

or, equivalently, using display-style indexing:

```
table Results2 OUT "ODBC" "steel2out.xls":
    [Product],
    {p in PROD} Inv[p,0] ~ Inv0,
    {t in 1..T} < {p in PROD} (
        Make[p,t] ~ ('Make' & t),
        Sell[p,t] ~ ('Sell' & t), Inv[p,t] ~ ('Inv' & t) ) >;
```

The key column labels the rows with product names. The data columns include one for the initial inventories, and then three representing production, sales, and inventories, respectively, for each period:

	A	B	C	D	E	F	G	H	I	J
1	Product	Inv0	Make1	Sell1	Inv1	Make2	Sell2	Inv2	Make3	Sell3
2	bands	10	5990	6000	0	6000	6000	0	1400	1400
3	coils	0	1407	307	1100	1400	2500	0	3500	3500
4										
5										
6										
7										

Conceptually, there is a symmetry between the row and column indexing of a two-dimensional table. But because the tables in these examples are being treated as relational tables, the table declaration must treat the row indexing and the column indexing in different ways. As a result, the declaration's expressions describing the row indexing are substantially different from its expressions describing the column indexing.

3.8 Standard and built-in table handlers

To work with external database files, AMPL relies on database *handlers*. These are add-ons, usually in the form of shared or dynamic link libraries, that can be loaded as needed. Handlers may be supplied by vendors of AMPL or of database software.

The initial release of the database access feature includes a "standard" Microsoft Windows database handler that communicates via the Open Database Connectivity (ODBC) application programming interface. It recognizes relational tables in the formats used by Microsoft Access, Microsoft Excel, and any other application for which an ODBC driver exists on your computer. (To see a list of ODBC drivers installed, open the ODBC or ODBC Data Sources control panel.)

In addition to any supplied handlers, minimal ASCII and binary relational table file handlers are built into AMPL for purposes of testing. Vendors may exercise the option of including other handlers as built-in.

The built-in set `_HANDLERS` gives a list of handlers currently seen by AMPL. A built-in symbolic parameter `_handler_lib` indexed over `_HANDLERS` records the shared library in which each handler was found (or `<built-in>` for built-in handlers). When the above-mentioned handlers are accessible, for example, this handler information appears as follows:

```
ampl: display _HANDLERS;
set _HANDLERS := tab bit odbc;

ampl: display _handler_lib;
_handler_lib [*] :=
```

```

    tab  '<built-in>'
    bit  '<built-in>'
    odbc  ampltabl.dll
;

```

A built-in symbolic parameter `_handler_desc` is also indexed over `_HANDLERS`. The value of `_handler_desc[h]` is normally a longer string that gives a summary of instructions for using handler `h`. For example:

```

ampl: print _handler_desc['tab'];
Builtin file.tab (ASCII table) handler: at most one string
(the file name, ending in ".tab") expected before ":[...]";
table_name.tab is assumed if there are no strings.

ampl: print _handler_desc['bit'];
Builtin file.bit (binary table) handler: exactly one string
(the file name, ending in ".bit") expected before ":[...]".

ampl: print _handler_desc['odbc'];
AMPL ODBC handler: expected 2-5 strings before ":[...]":
  'ODBC', connection_spec ['external_table_name'] ['time=...']
  ['verbose']
For IN tables, 'external_table_name' can also have the form
'SQL=sqlstmt',
where sqlstmt is a SQL statement, such as a SELECT statement.
Alternatives for connection_spec:
  'filename.ext', where ext is a registered ODBC
  extension;
  'filename.dsn' (written by the ODBC control panel's
  "File DSN");
  an explicit connection string of the form
  'DSN=...;DBQ=...';
  or an ODBC data source name (see the ODBC control
  panel).

```

As these summaries suggest, AMPL communicates with handlers through the *string-list* in the table declaration. The form and interpretation of the *string-list* are specific to each handler.

The remainder of this section describes the *string-lists* that are recognized by AMPL's standard ODBC handler. Following a general introduction, specific instructions are provided for the two applications, Access and Excel, that are used in many of the examples in preceding sections. A final subsection describes the *string-lists* recognized by the built-in binary and ASCII table handlers.

Using the standard ODBC table handler. The netlib AMPL directory provides a (gzip-compressed) Windows dynamic link library `ampltabl.dll` that supports database connections to via ODBC. To make this handler available to AMPL sessions, place it in the same directory as the AMPL program file (normally `ampl.exe`) or at another location recognized by the [ampltabl.dll](#) loading rules for Windows.

In the context of a declaration that begins `table table-name . . .`, the general form of the *string-list* for the standard ODBC table handler is

```

"ODBC" "connection-spec" "external-table-spec"opt "time=data-
column-list"opt "verbose"opt

```

The first string tells AMPL that data transfers using this `table` declaration should employ the standard ODBC handler. Subsequent strings then provide directions to that handler as follows.

The second string identifies the external database file that is to be read or written upon subsequent execution of the command `read table table-name` or `write table table-name`, respectively. There are several possibilities, depending on the form of the *connection-spec* and the configuration of Windows ODBC on your computer:

If the *connection-spec* is a Windows filename of the form *name.ext*, where *ext* is a 3-letter extension associated with an installed ODBC driver, then the named file is the database file.

If the *connection-spec* is a Windows filename of the form *name.dsn*, then the named file is treated as an ODBC File Data Source and the identity of the database file is determined from its contents.

If the *connection-spec* begins with `DSN=`, then it is interpreted as an ODBC Connection String from which the identity of the database file is determined.

If the *connection-spec* is an ODBC Data Source Name (DSN) for which a specific database file has been defined, then that file is taken to be the database file.

Information about your computer's configuration of ODBC drivers, data source names, file data sources, and related entities can be examined and changed through your ODBC control panel. The name of this control panel may be ODBC, ODBC Data Sources, or ODBC Data Sources (32bit), depending on the specifics of your installation.

The third string normally gives the name of the relational table, within the specified file, that is to be read or written upon execution of `read table table-name` or `write table table-name`. If the third string is omitted, then the name of the relational table is taken to be the same as the *table-name* of the containing `table` declaration. For writing, if the indicated table does not exist, it is created; if the table exists but all of the `table` declaration's *data-specs* have read/write status `OUT`, then it is overwritten. Otherwise, writing causes the existing table to be modified; each column written either overwrites an existing column of the same name, or becomes a new column appended to the table.

Alternatively, if the third string has the special form

```
"SQL=sql-query"
```

then the `table` declaration applies to the relational table that is (temporarily) created by a statement in the Structured Query Language, commonly abbreviated SQL. Specifically, a relational table is first constructed by executing the SQL statement given by *sql-query*, with respect to the database file given by the second string in the `table` declaration's string-list. Then the usual actions of the `table` declaration are applied to the constructed table. All columns specified in the `table` declaration should have read/write status `IN`, since it would make no sense to write to a temporary table. Normally the *sql-query* is a `SELECT` statement, which is SQL's primary device for operating on tables to create new ones.

As an example, if you wanted to read as data for `diet.mod` only those foods having a cost of \$2.49 or less, you could use an SQL query to extract the relevant records from the `Foods` table of your database:

```
table cheapFoods IN "ODBC" "diet.mdb"
  "SQL=SELECT * FROM Foods WHERE cost <= 2.49":
  FOOD <- [FOOD], cost, f_min, f_max;
```

Then to read the relevant data for parameter `amt`, which is indexed over nutrients and foods, you would want to read only those records that pertained to a food having a cost of \$2.49 or less. Here is one way that an SQL query could be used to extract the desired records:

```
option selectAmts "SELECT NUTR, Amounts.FOOD, amt FROM Amounts,
Foods "
  "WHERE Amounts.FOOD = Foods.FOOD and cost <= 2.49";

table cheapAmts IN "ODBC" "diet.mdb" ("SQL=" & $selectAmts):
  [NUTR, FOOD], amt;
```

Here we have used an AMPL option to store the string containing the SQL query. Then the table declaration's third string can be given by the relatively short string expression `"SQL=" & $selectAmts`.

Data values representing specific dates and times -- so-called timestamp data -- can be read and written by AMPL through `table` declarations in the same way as other data. When read into an AMPL numeric parameter, timestamp values are integers of the form `YYYYMMDDhhmmss`, with `YYYY` giving the year, `MM` the month, `DD` the day, and `hhmmss` the hours, minutes, and seconds. When written from a numeric parameter to an existing database column that has a timestamp format, integers of this form are automatically recognized as times and are handled accordingly. When integers of this form are written to a new column, however, it is necessary to tell the database handler that they are to be interpreted as times. This is done by adding the following string after the first three strings in the string-list:

```
"time=data-column-list"
```

The `data-column-list` is a comma-separated list of external database columns that are to have timestamp data type when created.

The string `"verbose"` after the first three strings requests diagnostic messages -- such as the `DSN=` string that ODBC reports using -- whenever the containing table declaration is used by a `read table` or `write table` command. (The ordering of strings after the first three in the string-list does not matter.)

Using the standard ODBC table handler with Access. To set up a relational table correspondence for reading or writing Microsoft Access files, specify the `ext` in the second string of the string-list as `mdb`:

```
"ODBC" "file-name.mdb" "external-table-spec" _opt "time=data-
column-list" _opt "verbose" _opt
```

The file called `"file-name.mdb"` must exist, though for writing it may be a database that does not yet contain any tables.

Using the standard ODBC table handler with Excel. To set up a relational table correspondence for reading or writing Microsoft Excel spreadsheets, specify the ext in the second string of the string-list as xls:

```
"ODBC" "file-name.xls" "external-table-name" opt "time=data-  
column-list" opt "verbose" opt
```

In this case, the second string identifies the external Excel workbook file that is to be read or written. For writing, the file specified by the second string is created if it does not exist already.

The external-table-name specified by the third string identifies a spreadsheet range, within the specified file, that is to be read or written; if this string is absent, it is taken to be the table-name given at the start of the table declaration. For reading, the specified range must exist in the Excel file. For writing, if the range does not exist, it is created, at the upper left of a new worksheet having the same name. If the range exists but all of the table declaration's data-specs have read/write status OUT, then it is overwritten.

Otherwise, writing causes the existing range to be modified. Each column written either overwrites an existing column of the same name, or becomes a new column appended to the table; each row written either overwrites entries in an existing row having the same key column entries, or becomes a new row appended to the table.

When writing causes an existing range to be extended, rows or columns are added at the bottom or right of the range, respectively. The cells of added rows or columns must be empty; otherwise, the attempt to write the table fails and the write table command elicits an error message. After a table is successfully written, the corresponding range is created or adjusted to contain exactly the cells of that table.

Built-in table handlers for text and binary files. For debugging and demonstration purposes, AMPL has built-in handlers for two very simple relational table formats. These formats store one table per file and convey equivalent information. One produces ASCII files that can be examined in any text editor, while the other creates binary files that are much faster to read and write.

For these handlers, the table declaration's string-list contains at most one string, identifying the external file that contains the relational table. If the string has the form

```
"file-name.tab"
```

then the file is taken to be an ASCII text file; if it has the form

```
"file-name.bit"
```

then it is taken to be a binary file. If no string-list is given, then a text file table-name.tab is assumed.

For reading, the indicated file must exist. For writing, if the file does not exist, it is created. If the file exists but all of the table declaration's data-specs have read/write status OUT, then it is overwritten. Otherwise, writing causes the existing file to be modified; each column written either replaces an existing column of the same name, or becomes a new column added to the table.

The format for the text files can be examined by simply writing one and viewing the results in a text editor. For example, the following AMPL session,

```
ampl: model diet.mod;  
ampl: data diet2a.dat;
```

```

ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032

ampl: table ResultList OUT "DietOpt.tab":
ampl?      [FOOD], Buy, Buy.rc, {j in FOOD} Buy[j]/f_max[j];
ampl: write table ResultList;

```

produces a file DietOpt.tab that can be seen to have the following content:

```

ampl.tab 1 3
FOOD      Buy      Buy.rc  'Buy[j]/f_max[j]'
BEEF 5.360613810741678 8.881784197001252e-16 0.5360613810741678
CHK      2          1.188840579710143      0.2
FISH     2          1.144407502131287      0.2
HAM      10         -0.3026513213981231      1
MCH      10         -0.551150895140665      1
MTL      10         -1.3289002557544745      1
SPG      9.306052855924984      0      0.9306052855924983
TUR      2          2.7316197783461194      0.2

```

In the first line, `ampl.tab` identifies this as an AMPL relational table text file, and is followed by the numbers of key and non-key columns, respectively. The second line gives the names of the table's columns, which may be any strings. (Use of the `~` operator to specify valid column-names is not necessary in this case.) Each subsequent line gives the values in one table row; numbers are written in full precision, with no special formatting or alignment.

4 Character Strings

Set members like "coils" and "BEEF" are treated by AMPL as strings of characters. Option values and filenames are also treated as strings in AMPL. New string functions, operators and conventions greatly expand the power and flexibility of the AMPL language in specifying strings.

This writeup first considers general-purpose string manipulation features, including the new string concatenation operator and a variety of new string functions. Next we describe some changes that make the display of strings more natural and flexible. The final two sections describe additional new functions and conventions for converting numbers to strings and strings to numbers.

The following table summarizes the concatenation operator and all of the functions for string handling, in the order in which they are introduced below. Arguments beginning with *s* refer to strings, with *i* refer to integers, and with *e* refer to any numeric or symbolic expressions.

Syntax	Return Type	Return Value
<i>s1</i> & <i>s2</i>	string	concatenation of <i>s1</i> and <i>s2</i>
length(<i>s1</i>)	number	number of characters in <i>s1</i>
match(<i>s1</i> , <i>s2</i>)	number	first position where <i>s2</i> matches a substring in <i>s1</i> (or 0 if it never appears)
substr(<i>s1</i> , <i>i2</i>)	string	substring of <i>s1</i> beginning at the position given by <i>i2</i> and extending to the end of <i>s1</i>
substr(<i>s1</i> , <i>i2</i> , <i>i3</i>)	string	substring of <i>s1</i> beginning at the position given by <i>i2</i> and having a length of <i>i3</i>
sub(<i>s1</i> , <i>s2</i> , <i>s3</i>)	string	result of substituting <i>s3</i> for the first match of <i>s2</i> in <i>s1</i>
gsub(<i>s1</i> , <i>s2</i> , <i>s3</i>)	string	result of substituting <i>s3</i> for all matches of <i>s2</i> in <i>s1</i>
sprintf(<i>s1</i> , <i>e2</i> , ...)	string	formatted string, the same as would be written by printf <i>s1</i> , <i>e2</i> , ...
num(<i>s1</i>)	number	decimal number represented by <i>s1</i>
num0(<i>s1</i>)	number	decimal number represented by <i>s1</i> , ignoring extraneous characters
char(<i>i1</i>)	string	string of one character whose unicode value is <i>i1</i>
ichar(<i>s1</i>)	number	unicode value of the first character in <i>s1</i>

4.1 String functions and operators

The concatenation operator & takes two strings as operands, and returns a string consisting of the left operand followed by the right operand. For example, given the sets NUTR and FOOD defined by diet.mod and diet2.dat (Figures 2-1 and 2-3), you could use concatenation to define a set NUTR_FOOD whose members represent origin-destination pairs:

```
ampl: model diet.mod;
ampl: data diet2.dat;

ampl: display NUTR, FOOD;
set NUTR := A B1 B2 C NA CAL;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR;

ampl: set NUTR_FOOD := setof {i in NUTR, j in FOOD} i & "_" & j;

ampl: display NUTR_FOOD;
set NUTR_FOOD :=
A_BEEF      B1_BEEF      B2_BEEF      C_BEEF      NA_BEEF      CAL_BEEF
A_CHK       B1_CHK       B2_CHK       C_CHK       NA_CHK       CAL_CHK
A_FISH      B1_FISH      B2_FISH      C_FISH      NA_FISH      CAL_FISH
A_HAM       B1_HAM       B2_HAM       C_HAM       NA_HAM       CAL_HAM
A_MCH       B1_MCH       B2_MCH       C_MCH       NA_MCH       CAL_MCH
A_MTL       B1_MTL       B2_MTL       C_MTL       NA_MTL       CAL_MTL
A_SPG       B1_SPG       B2_SPG       C_SPG       NA_SPG       CAL_SPG
A_TUR       B1_TUR       B2_TUR       C_TUR       NA_TUR       CAL_TUR;
```

This is not a set that you would normally want to define, but it might be useful if you have to read data in which strings like "B2_BEEF" appear (see the example below).

The length function takes a string as argument and returns the number of characters in it. The match function takes two string arguments, and returns the first position where the second appears as a substring in the first -- or zero if the second never appears as a substring in the first. For example:

```
ampl: display {j in FOOD} (length(j), match(j, "T"));

:      length(j) match(j, 'T')      :=
BEEF      4          0
CHK        3          0
FISH       4          0
HAM        3          0
MCH        3          0
MTL        3          2
SPG        3          0
TUR        3          1
;
```

The substr function takes a string and one or two integers as arguments. It returns a substring of the first argument that begins at the position given by the second argument; it has the length given by the third argument, or extends to the end of the string if no third argument is given. For instance:

```
ampl: display {j in FOOD} (substr(j,1,2), substr(j,3));

:      substr(j, 1, 2) substr(j, 3)      :=
```

BEEF	BE	EF
CHK	CH	K
FISH	FI	SH
HAM	HA	M
MCH	MC	H
MTL	MT	L
SPG	SP	G
TUR	TU	R

;

An empty string is returned if the second argument is greater than the length of the first argument, or if the third argument is less than 1.

As an example of the use of several of these functions, suppose that you want to use the model from `diet.mod` and to supply the nutrition amount data in a table like this:

```
param: NUTR_FOOD: amt_nutr :=
    A_BEEF      60
    B1_BEEF     10
    CAL_BEEF    295
    CAL_CHK     770 ...
```

Then in addition to the declarations for the parameter `amt` used in the model,

```
set NUTR;
set FOOD;
param amt {NUTR,FOOD} = 0;
```

you would declare a set and a parameter to hold the data from the "nonstandard" table:

```
set NUTR_FOOD;
param amt_nutr {NUTR_FOOD} = 0;
```

To use the model, you need to write an assignment of some kind to get the data from set `NUTR_FOOD` and parameter `amt_nutr` into sets `NUTR` and `FOOD` and parameter `amt`.

One solution is to extract the sets first, and then convert the parameters

```
set NUTR := setof {ij in NUTR_FOOD}
    substr(ij,1,match(ij,"_")-1);
set FOOD := setof {ij in NUTR_FOOD} substr(ij,match(ij,"_")+1);

param amt {i in NUTR, j in FOOD} := amt_nutr[i & "_" & j];
```

As an alternative, you can extract the sets and parameters together, by use of an AMPL script such as the following:

```
param iNUTR symbolic;
param jFOOD symbolic;
param upos 0;

let NUTR := {};
let FOOD := {};

for {ij in NUTR_FOOD} {
    let upos := match(ij,"_");
    let iNUTR := substr(ij,1,upos-1);
    let jFOOD := substr(ij,upos+1);

    let NUTR := NUTR union {iNUTR};
```

```

    let FOOD := FOOD union {jFOOD};
    let amt[iNUTR,jFOOD] := amt_nutr[ij];
}

```

Under either alternative, errors such as a missing "_" in a member of NUTR_FOOD are eventually signaled by error messages.

For completeness, AMPL also provides two functions to make substitutions in a string. Both `sub` and `gsub` take three strings as arguments. Both return the string that results when the third argument is substituted for the second in the first; `sub` substitutes only for the first occurrence of the second argument, while `gsub` substitutes for all occurrences. For example, to replace each underscore in the membership of set NUTR_FOOD above with two hyphens,

```

let NUTR_FOOD := setof {ij in NUTR_FOOD} sub(ij, '_', '--');

ampl: display NUTR_FOOD;
set NUTR_FOOD :=
A--BEEF    B1--BEEF    B2--BEEF    C--BEEF    NA--BEEF    CAL--BEEF
A--CHK     B1--CHK     B2--CHK     C--CHK     NA--CHK     CAL--CHK
A--FISH     B1--FISH    B2--FISH    C--FISH    NA--FISH    CAL--FISH
A--HAM      B1--HAM     B2--HAM     C--HAM     NA--HAM     CAL--HAM
A--MCH      B1--MCH     B2--MCH     C--MCH     NA--MCH     CAL--MCH
A--MTL      B1--MTL     B2--MTL     C--MTL     NA--MTL     CAL--MTL
A--SPG      B1--SPG     B2--SPG     C--SPG     NA--SPG     CAL--SPG
A--TUR      B1--TUR     B2--TUR     C--TUR     NA--TUR     CAL--TUR;

```

If the second argument has no occurrences in the first, then `sub` or `gsub` returns the first argument unchanged.

In `match`, `sub` and `gsub`, the second argument is actually taken to represent a "regular expression"; if it contains certain special characters, it is interpreted as a pattern that may match many sub-strings. The pattern "`^B[0-9]+_`", for example, matches any sub-string consisting of a B followed by one or more digits and then an underscore, and occurring at the beginning of a string. To use this feature, see the separate description of regular expression rules.

4.2 String expressions in AMPL commands

String-valued expressions may appear, *enclosed in parentheses*, in several AMPL contexts that previously required literal strings:

filenames that are part of commands, including `model`, `data`, and `commands`

filenames following `<` or `>` to specify redirection of input or output

values assigned to AMPL options by an `option` command

Here are two examples. The following script solves `diet.mod` with a series of different data files `dietA.dat`, `dietB.dat`, `dietC.dat`, ... and saves the solution to files `dietA.out`, `dietB.out`, `dietC.out`, ...:

```

model diet.mod;
set CASES := {"A", "B", "C"};

```

```

for {j in CASES} {
  reset data;
  data ("diet" & j & ".dat");

  solve;
  display Buy ("diet" & j & ".out");
}

```

The following script solves the same problem four times, each using a different pairing of the directives `primal` and `dual` with the directives `primalopt` and `dualopt`:

```

model sched.mod;
data sched.dat;

option solver cplex;
set DIR1 := {"primal", "dual"};
set DIR2 := {"primalopt", "dualopt"};

for {i in DIR1, j in DIR2} {
  option cplex_options (i & " " & j);
  solve;
}

```

See the next section for examples that generate consecutive numbers as parts of filenames and directives.

4.3 Number-to-string conversions

When you tell AMPL to exhibit a numerical value, you are invoking a number-to-string conversion of some kind. This conversion can be triggered and regulated by a variety of options, commands and functions, several of them new.

Options ending in `_precision` and `_round` enable you to control number-to-string conversion in specific contexts. In general terms, their values are interpreted as follows:

context_precision *n*

- n* 0: round to *n* digits of precision.
- n* = 0: show in **full precision**: the shortest decimal string representation that, when correctly rounded back to the computer's internal representation, yields the original numerical value

context_round *n*

- n* 0: round to *n* digits after the decimal point
- n* = 0: round to integer
- n* < 0: round to *-n* digits before the decimal point

The different possibilities for *context* are:

Options	Context affected
csvdisplay_precision csvdisplay_round	Numbers from the obscure <code>_display</code> and <code>csvdisplay</code> commands
display_precision display_round	Numbers from the <code>display</code> command
expand precision	Numbers from the new <code>expand</code> command

expand_round	
MD_precision	Numbers written in debug output produced by the obscure -M and -D command-line switches
objective_precision	Optimal objective from the solve command
output_precision	Numbers written by the solve command to the file that will be read by a solver
print_precision print_round	Numbers from the print command
solution_precision solution_round	Values of variables and dual variables returned by the solve or solution command

The `_round` variant (if any) takes precedence when it has a numerical value; otherwise the `precision` variant is used.

The `printf` command provides more precise control over number-to-string conversions. Its syntax is

```
printf indexingopt format-string, expression-listopt redirectionopt ;
```

Members of the *expression-list* that evaluate to numbers are converted to strings according to instructions encoded into the *format-string*. The final result of this formatting is a character string that is sent to a file specified by the optional *redirection* or else by default to standard output. A guide to format strings is provided on the separate [printf](#) rules page (and on pages 328-329 of the AMPL book). Format strings in AMPL have mostly the same interpretation as in the C programming language. The most notable exceptions are AMPL's interpretation of `%.g` or `%.0g` to specify full precision (as explained earlier in this section), and the introduction of `%q` and `%Q` to produce quoted strings (as described under "Display formats for strings" below).

The new `sprintf` function also specifies a *format-string* and an *expression-list*:

```
sprintf ( format-string, expression-listopt )
```

This function performs conversions in the same way as `printf`, except that the resulting formatted string is not sent to an output device, but instead becomes the function's return value.

Numbers are also converted automatically to strings when they appear as arguments to the string concatenation operator `&`. For example, for a multi-week model you can create a set of generically-named periods `WEEK1`, `WEEK2`, and so forth, by declaring:

```
param T integer 1;
set WEEKS ordered := setof {t in 1..T} "WEEK" & t;
```

Many useful applications of this feature occur in scripts that (as in our previous examples) process a series of files or set an option in several ways. For example, the following script solves `diet.mod` with a series of different data files `diet1.dat`, `diet2.dat`, `diet3.dat`, ... and saves the solution to files `diet1.out`, `diet2.out`, `diet3.out`, ...:

```
model diet.mod;
param Nfiles integer := 3;

for {j in 1..Nfiles} {
```

```

    reset data;
    data ("diet" & j & ".dat");

    solve;
    display Buy ("diet" & j & ".out");
}

```

The following script solves the same problem three times, each using a different value for the solver's branch directive:

```

model multmip3.mod;
data multmip3.dat;

option solver cplex;

for {i in -1 .. 1} {
    option cplex_options ("branch " & i);
    solve;
}

```

Numeric operands to & are always converted to full precision (or equivalently, to `% .0g` format) as previously defined. The conversion thus produces the expected results for concatenation of numerical constants and of indices that run over sets of integers or constants, as in our examples. Full precision conversion of computed fractional values may surprise you, however. The following variation on the preceding example would seem to solve the problem for values 0.1, 0.2, 0.3, and 0.4 of directive `linesearch_tolerance`, saving the listings from solve in files `nltrans0.1` through `nltrans0.4`:

```

model nltransd.mod;
data nltrans.dat;

for {i in 0.1 .. 0.4 by 0.1} {
    option reset_initial_guesses 1;
    option minos_options ("linesearch_tolerance=" & i);

    solve ("nltrans" & i);
}

```

The actual files created are

```

nltrans0.1
nltrans0.2
nltrans0.300000000000000004
nltrans0.4

```

Because 0.1 cannot be stored exactly in the computer's internal base-2 representation, the third member of the set `0.1 .. 0.4 by 0.1` in the `for` loop is slightly different from 0.3 in "full" precision. There is no easy way to predict this behavior, but you can prevent it by specifying the conversion explicitly. In our example, you could replace `"nltrans" & i` with `sprintf("nltrans%3.1f", i)`.

4.4 String-to-number conversions

AMPL does not automatically convert strings to numbers; if an expression or command uses a string where a number is expected, it is rejected with an error message. You can

specify conversions from strings to numbers explicitly, however, by use of the new `num` and `num0` functions. Both take a string as an argument, and return the numerical value expressed by the string. Thus, for example, suppose the strings declared by

```
param cc {j in FOOD} symbolic;
```

are "cost codes" that contain the cost per unit in the leading 4 characters: "3.19BE", "2.59CH", and so forth. The expression `substr(costcode[j],1,4) * Buy[j]` is rejected, because a string appears where a number is expected, as the left operand of `*`; but

```
num(substr(costcode[j],1,4)) * Buy[j]
```

is correct.

The `num` function accepts an argument that evaluates to a valid AMPL numerical constant written as a string, such as "-12.34" or "1.2e+6". Leading and trailing white space -- any combination of spaces, tabs and newlines -- is ignored. The use of `num` with any other argument is flagged as an error.

The `num0` function is more forgiving. It extracts the longest leading substring of its argument that would be acceptable to `num`, returns the corresponding numerical value, and disregards the rest of the argument. If no leading substring can be converted, a value of zero is returned. Thus `num0` accepts "12.34kg" or "1.2e+4?", for example, returning the numbers 12.34 and 12000. Our sample expression above, with `costcode[j]` having the form "3.19BE", can be written more concisely as

```
num0(costcode[j]) * Buy[j]
```

If the cost codes are instead of the form "BE3.19", however, then `num0` returns 0 for all of them.

4.5 Character-code conversions

Two complementary functions convert between characters and integer character codes.

The `char` function takes a nonnegative integer as an argument, and returns a string of length 1 comprising the corresponding unicode character. (There is no distinct character data type in AMPL.) Unicode is a superset of ASCII, so `char(65)` returns "A", and `display char(7)` sends a "control-G" to the AMPL output stream (where it may provoke a "beep" or other alert sound).

The `ichar` function is the inverse of `char`. It takes a string as argument, and returns the corresponding integer unicode for the string's first character. Thus `ichar("A")` and `ichar("AT&T")` both return 65.

4.6 Display formats for strings

A string is normally displayed (by `display`, `print` or `printf`) as simply the list of characters that comprise it, without surrounding delimiters of any kind. The same holds when a string is passed to any function. This is a change from some of the earlier versions of AMPL, in which the surrounding quotes were sometimes shown or passed.

Strings still do need to be delimited by quotes (' or ") when you write them as input to AMPL, so that they can be parsed correctly. The only exception is in data statements (Chapter 9 of the AMPL book) where for convenience the quotes may be omitted from strings that contain only letters, digits, and underscores and that do not represent numbers.

For the command `printf` or the function `sprintf`, the conversions specification `%s` in the format string requests the unquoted contents of the string. Two new specifiers have been added to produce quoted strings: `%q` shows quotes if and only if they would be required in a data statement, while `%Q` shows quotes around all strings.

5 Complementarity Problems

A variety of physical and economic phenomena are most naturally modeled by saying that certain pairs of inequality constraints must be *complementary*, in the sense that at least one must hold with equality. These conditions may in principle be accompanied by an objective function, but are more commonly used to construct *complementarity problems* for which a feasible solution is sought. Indeed, optimization may be viewed as a special case of complementarity, since the standard optimality conditions for linear and smooth nonlinear optimization are complementarity problems. Other kinds of complementarity problems do not arise from optimization, however, or cannot be conveniently formulated or solved as optimization problems.

A new AMPL operator, **complements**, permits complementarity conditions to be specified directly in constraint declarations. Complementarity models can thereby be formulated in a natural way, and instances of such models are easily sent to special solvers for complementarity problems.

To motivate the syntax of the new operator, we begin by considering how AMPL might be extended to express the complementary slackness conditions for standard and bounded-variable linear programs. We then give a general definition of the **complements** operator for pairs of inequalities, and for more general "mixed" complementarity conditions via double inequalities. We also comment on an AMPL interface to Dirkse and Ferris's PATH solver for "square" mixed complementarity problems. Finally, we describe extensions to accommodate complementarity constraints in several of AMPL's existing features: the presolve phase, constraint-name suffixes, and generic synonyms for constraints.

Illustrations are taken from a collection of AMPL complementarity models that use the new syntax. See the complementarity model index for links to the complete models.

5.1 Motivation

The requirements for a new complementarity constraint operator can be shown by considering one of the simplest examples: the linear complementarity problem that gives optimality conditions for a linear program. This introduction is likely to be of most value to those who are familiar with the duality theory of linear programming, but not so familiar with the various forms of complementarity problems. Others may want to skip to the next section where the new operator is defined more formally and concisely.

To begin, consider a linear program that has the following elementary primal form:

```

minimize PrimalObj: sum {j in J} c[j] * X[j];

subj to PrimalConstr {i in I}:
    sum {j in J} a[i,j] * X[j] = b[i];

subj to PrimalBounds {j in J}: X[j] = 0;

```

The corresponding "complementary slackness" conditions for optimality are easily represented like this:

```

PrimalConstr {i in I}:
    sum {j in J} a[i,j] * X[j] = b[i];

PrimalBounds {j in J}: X[j] = 0;

DualConstr {j in J}:
    sum {i in I} Y[i] * a[i,j] + Z[j] = c[j];

DualBounds {i in I}: Z[i] = 0;

Complementarity {j in J}: X[j] = 0 or Z[j] = 0;

```

Of course, AMPL does not (currently) accept the `or` operator in constraints, but the `Complementarity[j]` constraints can be written equivalently in this case as nonlinear equations:

```

Complementarity {j in J}: X[j] * Z[j] = 0;

```

Consider now the case, only slightly more complicated, of a linear program with arbitrary bounds on the variables:

```

minimize PrimalObj: sum {j in J} c[j] * X[j];

subj to PrimalConstr {i in I}:
    sum {j in J} a[i,j] * X[j] = b[i];

subj to PrimalBounds {j in J}: l[j] <= X[j] <= u[j];

```

The corresponding complementary slackness conditions are as follows:

```

PrimalConstr {i in I}:
    sum {j in J} a[i,j] * X[j] = b[i];

PrimalBounds {j in J}: l[j] <= X[j] <= u[j];

DualConstr {j in J}:
    sum {i in I} Y[i] * a[i,j] + Z[j] = c[j];

Complementarity {j in J}:

    X[j] = l[j] implies Z[j] = 0 and
    X[j] = u[j] implies Z[j] <= 0 and

    l[j] < X[j] < u[j] implies Z[j] = 0;

```

The constraints `Complementarity[j]` are again not recognized by AMPL, but in this case they also lack any concise and straightforward algebraic equivalent.

It would be desirable to be able to express these two similar forms of complementary problem in similar ways. To make this possible, we observe that there are two "elements" comprising each Complementarity[j] constraint: $X[j] = 0$ and $Z[j] = 0$ in the first case, and $l[j] < X[j] < u[j]$ and $Z[j]$ in the second. The members of each element pair "complement" each other in a sense appropriate to their form. This suggests defining a new AMPL operator, `complements`, to explicitly denote the complementarity relationship. The first form of complementary slackness then becomes

```
PrimalConstr {i in I}:
    sum {j in J} a[i,j] * X[j] = b[i];

DualConstr {j in J}:
    sum {i in I} Y[i] * a[i,j] + Z[j] = c[j];

Complementarity {j in J}:
    X[j] = 0 complements Z[j] = 0;
```

and the second form is the same except for the pair of elements that complement each other:

```
PrimalConstr {i in I}:
    sum {j in J} a[i,j] * X[j] = b[i];

DualConstr {j in J}:
    sum {i in I} Y[i] * a[i,j] + Z[j] = c[j];

Complementarity {j in J}:
    l[j] <= X[j] <= u[j] complements Z[j];
```

To be feasible for this new constraint type, a solution must satisfy the two inequalities that appear, *and* must cause the two arguments of the `complements` operator to complement each other. The meaning of "complement each other" for these two cases is exactly as given in our original statements of the complementary slackness conditions above.

The same conditions can be expressed more concisely by substituting the variables $Z[j]$ out of the constraints. We then have for the case of nonnegative variables,

```
PrimalConstr {i in I}:
    sum {j in J} a[i,j] * X[j] = b[i];

Complementarity {j in J}:
    X[j] = 0 complements
    c[j] - sum {i in I} Y[i] * a[i,j] = 0;
```

and for the case of bounded variables,

```
PrimalConstr {i in I}:
    sum {j in J} a[i,j] * X[j] = b[i];

Complementarity {j in J}:
    l[j] <= X[j] <= u[j] complements
    c[j] - sum {i in I} Y[i] * a[i,j];
```

The first form could also be written even more concisely as

```
PrimalConstr {i in I}:
    sum {j in J} a[i,j] * X[j] = b[i];
```

```
Complementarity {j in J}:
  X[j] = 0 complements
    sum {i in I} Y[i] * a[i,j] <= c[j];
```

In fact it makes sense to allow the arguments of `complements` to be any two single-inequality constraints, or any double-inequality constraint and any expression.

It turns out that these few constraint forms are sufficient to conveniently represent most linear complementarity problems of interest, regardless of whether they have any relationship to a linear program. The same is true of nonlinear complementarity problems, when these forms are extended to permit nonlinear constraints and expressions.

We next proceed to define the `complements` operator more formally and generally, first for the case of two complementary single inequalities, and then for the case of a double inequality complementing an expression.

5.2 Complementary pairs of inequalities

A complementarity relation between two inequalities is expressed by placing the `complements` operator between them in an AMPL constraint declaration. The general form for this constraint type is

```
single-inequality complements single-inequality ;
```

where a *single-inequality* is any valid constraint expression -- linear or nonlinear -- containing one `=` or `<=` operator. A constraint of this type is satisfied if both of the *single-inequality* constraints are satisfied, and at least one is satisfied with equality.

As an example, [pies.mod](#) makes the following statement about coal shipments and prices:

```
subject to delct {c in CREG, u in USERS}:
  0 <= Ct[c,u] complements ctcost[c,u] + Cv[c] = P["C",u];
```

To satisfy this constraint, the variables `Ct[c,u]`, `Cv[c]`, and `P["C",u]` must satisfy `0 <= Ct[c,u]`, `ctcost[c,u] + Cv[c] = P["C",u]`, and either `0 = Ct[cr,u]` or `ctcost[cr,u] + Cv[cr] = P["C",u]`, or both.

Even though both inequalities are linear in this case, the complementarity condition must be regarded as nonlinear. In fact, the same restrictions are often expressed as

```
Ct[cr,u] = 0,
ctcost[cr,u] + Cv[cr] - P["C",u] = 0, and
Ct[cr,u] * (ctcost[cr,u] + Cv[cr] - P["C",u]) = 0,
```

the last being explicitly a nonlinear equality. These are all readily written as AMPL arithmetic constraints, without need for the `complements` operator. They tend to obscure the nature of the complementarity restriction, however, both for people reading the model and for solvers that take advantage of complementarity. They also do not extend well to the more general "mixed" case to be discussed next.

5.3 Complementary double inequalities

A "mixed" complementarity condition is specified by placing the `complements` operator between a double inequality and an expression in an AMPL constraint declaration. The general forms for this constraint type are

double-inequality `complements` *expression* ;
expression `complements` *double-inequality* ;

where *double-inequality* is any AMPL constraint expression -- linear or nonlinear -- containing two = or two <= operators. The conditions for a constraint of this type to be satisfied are as follows:

- if the left-hand <= or the right-hand = of the *double-inequality* holds with equality, then the *expression* is greater than or equal to 0;
- if the right-hand <= or the left-hand = of the *double-inequality* holds with equality, then the *expression* is less than or equal to 0;
- if neither side of the *double-inequality* holds with equality, then the *expression* equals 0.

For the special case in which the *double-inequality* has the form $0 \leq \text{body} \leq \text{Infinity}$, these conditions reduce to the previously stated conditions for complementarity of a pair of single inequalities.

As an example, [pies.mod](#) makes the following statement about coal production:

```
subject to delc {c in CREG, t in CTYP}:  
  0 <= C[c,t] <= cmax[c,t] complements  
  ccost[c,t] + (sum {res in R} cruse[res,c,t]*Mu[res]) - Cv[c];
```

This implies that the following conditions must be satisfied:

- if $0 = C[cr,t]$, then $ccost[cr,t] + (\text{sum } \{res \text{ in } R\} cruse[res,cr,t] * Mu[res]) - Cv[cr] = 0$;
- if $C[cr,t] = cmax[cr,t]$, then $ccost[cr,t] + (\text{sum } \{res \text{ in } R\} cruse[res,cr,t] * Mu[res]) - Cv[cr] \leq 0$;
- if $0 < C[cr,t] < cmax[cr,t]$, then $ccost[cr,t] + (\text{sum } \{res \text{ in } R\} cruse[res,cr,t] * Mu[res]) - Cv[cr] = 0$.

There is no equally natural way to state these conditions in terms of arithmetic constraints alone; at the least, a transformation involving additional variables would be required.

For completeness, the special case in which the left-hand side equals the right-hand side of the double inequality may be written using one of the forms

equality `complements` *expression* ;
expression `complements` *equality* ;

A constraint of this kind is equivalent to an ordinary constraint consisting only of the *equality*; it places no restrictions on the *expression*.

5.4 Using the PATH solver

PATH, a mixed complementarity solver, takes as its input a "square" complementarity system consisting of m double-inequality complementarity constraints in m variables. An AMPL/PATH interface has been developed to accept a broader variety of AMPL constraint systems that can be converted to the form that PATH requires.

Specifically, AMPL/PATH accepts any constraint system such that the number of variables equals the number of complementarity constraints plus the number of equality constraints. There may be any number of additional inequality constraints, but there must not be any objective. The AMPL/PATH interface automatically transforms constraints from this form to the more restricted form required by PATH, and later inverts the transformation so that the results may be viewed in terms of the originally specified model.

AMPL/PATH is invoked by setting AMPL's `solver` option to `path` prior to a `solve` command:

```
ampl: model pies.mod;  
ampl: data pies.dat;  
ampl: option solver path;  
ampl: solve;  
PATH 3.0: Solution found.  
14 iterations (1 for crash); 28 pivots.  
30 function, 16 gradient evaluations.  
ampl:
```

An error message such as

```
Nonsquare complementarity system:  
    38 equations and 42 variables.  
PATH 3.0 requires a square system; this one is 38 x 42.
```

indicates that the specified problem instance cannot be solved by PATH, because the number of variables does not equal the total of equality and complementarity constraints.

5.5 Presolve

AMPL incorporates a presolve phase that can substantially simplify some linear programs. By use of an iterative algorithm, presolve deduces tighter bounds on variables and constraint expressions. It may infer from these bounds that certain variables can be fixed and constraints dropped. In the presence of complementarity constraints, several new kinds of inferences become possible.

As an example, given a constraint of the form

$expr1 = 0$ complements $expr2 = 0$,

if presolve can deduce that $expr1$ is strictly positive for all feasible points -- in other words, that it has a positive lower bound -- then it can replace the constraint by $expr2 = 0$. Similarly, in a constraint of the form

$expr1 \leq expr2 \leq expr3$ complements $expr4$,

there are various possibilities, including the following:

If presolve can deduce	Then the constraint can be replaced by
for all feasible points that	
$expr2 < expr3$	$expr1 \leq expr2$ complements $expr4 = 0$
$expr1 < expr2 < expr3$	$expr4 = 0$
$expr4 < 0$	$expr2 = expr3$

Transformations of these kinds are carried out automatically, unless you use option `presolve 0` to turn off the presolve phase. As is currently the case for ordinary constraints, results are reported in terms of the original model, so that the benefits of presolving are achieved without any special attention from the user.

By displaying a few predefined parameters,

<code>_ncons</code>	the number of ordinary constraints before presolve,
<code>_nccons</code>	the number of complementarity conditions before presolve,
<code>_sncons</code>	the number of ordinary constraints after presolve,
<code>_sncccons</code>	the number of complementarity conditions after presolve,

or by setting option `show_stats 1`, you can get some information on the number of simplifying transformations that presolve has made:

```
ampl: model pies.mod;  
ampl: data pies.dat;  
ampl: option solver path;  
ampl: option show_stats 1;  
ampl: solve;
```

```
Presolve eliminates 42 constraints.  
Presolve resolves 8 of 42 complementarity conditions.  
Adjusted problem:  
42 variables:  
    6 nonlinear variables  
    36 linear variables  
42 constraints; 142 linear nonzeros  
    34 nonlinear constraints  
    8 linear constraints  
34 complementarity conditions among the constraints:  
    28 linear, 6 nonlinear.  
0 objectives.
```

```
PATH 3.0: Solution found.  
14 iterations (1 for crash); 28 pivots.  
30 function, 16 gradient evaluations.
```

```
ampl: display _ncons, _nccons;  
_ncons = 84  
_nccons = 42  
  
ampl: display _sncons, _sncccons;  
_sncons = 42  
_sncccons = 34
```

When first instantiating the problem, AMPL counts each complementarity constraint as two ordinary constraints (the two arguments to `complements`) and also as a complementarity condition. Thus `_nccons` equals the number of complementarity constraints before presolve, and `_ncons` equals twice `_nccons` plus the number of

non-complementarity constraints before presolve. The presolve messages at the beginning of the `show_stats` output indicate how much presolve was able to reduce these numbers.

5.6 Auxiliary solution values

Solution information associated with an AMPL constraint can be represented by expressions of the form `cname.suf`, where `cname` is any constraint name and `suf` is one of several predefined suffixes. For a constraint in the general form

$$lbound \leq body \leq ubound,$$

where `lbound` and `ubound` are constants, the recognized suffixes and their values are:

<code>cname.body</code>	<code>body</code>
<code>cname.lb</code>	<code>lbound</code>
<code>cname.ub</code>	<code>ubound</code>
<code>cname.lslack</code>	<code>body - lbound</code>
<code>cname.uslack</code>	<code>ubound - body</code>
<code>cname.slack</code>	<code>min (cname.lslack, cname.uslack)</code>

Any ordinary constraint can be put into this form, possibly with `lbound` equal to `-Infinity` or `ubound` equal to `Infinity`.

This feature extends to complementarity constraints, but with two collections of suffixes, of the form `cname.Lsuf` and `cname.Rsuf`, corresponding to the left and right operands of complements, respectively. Thus for example after `pies.mod` has been solved, you can use the following `display` command to look at values associated with the constraint `delc`:

```

ampl: display delc.Llb, delc.Lbody, delc.Lub,
ampl? delc.Rlb, delc.Rbody, delc.Rub;

: delc.Llb delc.Lbody delc.Lub delc.Rlb delc.Rbody
delc.Rub :=
1 1 0 300 300 -Infinity -10.7551
Infinity
1 2 0 300 300 -Infinity -9.51119
Infinity
1 3 0 227.889 400 -Infinity -8
Infinity
2 1 0 200 200 -Infinity -10.5051
Infinity
2 2 0 300 300 -Infinity -8.91133
Infinity
2 3 0 600 600 -Infinity -8.46915
Infinity
;

```

Subscripted forms of the constraint name can also be suffixed:

```

ampl: display {t in ctyp} (delc[1,t].Lslack,
delc[1,t].Rslack);

: delc[1,t].Lslack delc[1,t].Rslack :=

```

```

1          0          Infinity
2          0          Infinity
3      172.111      Infinity
;

```

Notice that since the right operand of `delc` is an expression, it is treated as a "constraint" with infinite lower and upper bounds, and hence infinite slack.

The suffix `cname.slack` is also defined for complementarity constraints. For complementary pairs of single inequalities, it is equal to `min(cname.Lslack, cname.Rslack)`. Hence it is nonnegative if and only if both inequalities are satisfied. For complementary double inequalities of the form

```

expr complements lbound <= body <= ubound
lbound <= body <= ubound complements expr

```

`cname.slack` is defined to be

<code>min(expr, body - lbound)</code>	if <code>body <= lbound</code>
<code>min(-expr, ubound - body)</code>	if <code>body = ubound</code>
<code>-abs(expr)</code>	otherwise

Hence in this case it is always nonpositive, and is zero when one part of the double inequality is satisfied exactly.

If `cname` for a complementarity constraint appears unaffixed in an expression, it is interpreted as representing `cname.slack`.

5.7 Generic synonyms

AMPL's generic synonyms for constraints have been extended to encompass complementarity. The generic synonyms for ordinary (equality and inequality) constraints remain the same, while a parallel collection of synonyms for complementarity constraints is constructed mainly by substituting `ccon` for `con` where appropriate.

From the modeler's view (before presolve), the ordinary constraint synonyms remain

<code>_ncons</code>	the number of ordinary constraints before presolve,
<code>_conname</code>	names of the ordinary constraints before presolve,
<code>_con</code>	synonyms for the ordinary constraints before presolve,

The new complementarity constraint synonyms are

<code>_nccons</code>	the number of complementarity constraints before presolve,
<code>_cconname</code>	names of the complementarity constraints before presolve,
<code>_ccon</code>	synonyms for the complementarity constraints before presolve,

Because each complementarity constraint also gives rise to two ordinary constraints, as explained in the preceding discussion of presolve, there are two entries in `_conname` corresponding to each entry in `_cconname`:

```

ampl: model josephy.mod;
ampl: data josephy.dat;

ampl: display _conname, _cconname;

:   _conname _cconname   :=

```

```

1  'f[1].L'    'f[1]'
2  'f[1].R'    'f[2]'
3  'f[2].L'    'f[3]'
4  'f[2].R'    'f[4]'
5  'f[3].L'    .
6  'f[3].R'    .
7  'f[4].L'    .
8  'f[4].R'    .
;

```

For each complementarity constraint *cname*, the left and right arguments to the complements operator are the ordinary constraints named *cname*.L and *cname*.R. You can see this by using the synonym terminology to expand complementarity constraint *f[1]* and the corresponding two ordinary constraints from the example above:

```

ampl: expand f[1];
s.t. f[1]:
    -x[1] <= 0
    complements
        3*x[1]*x[1] + 2*x[1]*x[2] + 2*x[2]*x[2] + x[3] +
3*x[4] = 6;

ampl: expand {i in 1..2} _con[i];
s.t. f[1].L:
    -x[1] <= 0;

s.t. f[1].R:
    3*x[1]*x[1] + 2*x[1]*x[2] + 2*x[2]*x[2] + x[3] +
3*x[4] = 6;

```

From the solver's view (after presolve), a more limited collection of synonyms is defined:

<code>_sncons</code>	the number of all constraints after presolve,
<code>_sncccons</code>	the number of complementarity constraints after presolve,
<code>_sconname</code>	names of all constraints after presolve,
<code>_scon</code>	synonyms for all constraints after presolve,

Necessarily `_sncccons` is less than or equal to `_sncons`, with equality only when all constraints are complementarity constraints. By using `solexpand` in place of `expand`, you can see the form in which AMPL sent complementarity constraint *f[1]* to the solver:

```

ampl: solexpand f[1];
s.t. f[1]:
    3*x[1]*x[1] + 2*x[1]*x[2] + 2*x[2]*x[2] + -6 + x[3] +
3*x[4] = 0
    complements
        0 <= x[1];

```

To simplify the problem description that is sent to the solver, AMPL converts every complementarity constraint into one of the following canonical forms,

```

expr complements lbound <= var <= ubound,
expr <= 0 complements var <= ubound,
expr = 0 complements lbound <= var,

```

where *var* is the name of a different variable for each constraint. A predefined array of integers, `_scvar`, gives the indices of these canonical complementing variables in the

generic variable arrays `_var` and `_varname`. This terminology can be used to display a list of names of such variables:

```
AMPL: display {i in 1.._sncons} _varname[_scvar[i]];

_varname[_scvar[i]] [*] :=
1  'x[1]'
2  'x[2]'
3  'x[3]'
4  'x[4]'
;
```

When constraint `i` is an ordinary equality or inequality, `_scvar[i]` is 0.

6 Examining Models and Data

Two new commands let you more easily review an AMPL model from the command line. The `show` command displays the names of model components and the definitions of individual components. The `xref` command lists all components that depend on a given component. The `expand` command exhibits selected objectives and constraints that AMPL has generated from a model and data, or analogous information for variables. The output from any of these commands may be redirected to a file, by adding *filename* at the end.

To make a listing or a test that applies to all variables, constraints, or objectives, AMPL now also offers "generic" names for these components. New built-in sets and functions let AMPL also access additional information about model components, such as their names and dimensions.

6.1 The `show` command: displaying model components

By itself, this command lists the names of all components of the current model:

```
ampl: model multmip3.mod;
ampl: show;

parameters: demand fcost limit maxserve minload supply vcost

sets:      DEST      ORIG      PROD

variables:   Trans      Use

constraints:   Demand      Max_Serve      Min_Ship      Multi      Supply

objective:    total_cost

checks: one, called check 1.
```

This display may be restricted to components of one or more types:

```
ampl: show vars;

variables:   Trans      Use
```

```
ampl: show obj, constr;
```

```
objective:    total_cost
```

```
constraints:   Demand    Max_Serve    Min_Ship    Multi    Supply
```

The show command can also display the declarations of individual components, saving you the trouble of looking them up in the model file:

```
ampl: show total_cost;
```

```
minimize total_cost: sum{i in ORIG, j in DEST, p in PROD}
vcost[i,j,p]*Trans[i,j,p] + sum{i in ORIG, j in DEST}
fcost[i,j]*Use[i,j];
```

```
ampl: show vcost, fcost, Trans;
```

```
param vcost{ORIG, DEST, PROD} = 0;
param fcost{ORIG, DEST} = 0;
var Trans{ORIG, DEST, PROD} = 0;
```

If an item following show is the name of a component in the current model, the declaration of that component is displayed. Otherwise, the item is interpreted as a component type according to its first letter or two:

```
c  constraints
ch checks
e  environments
f  functions
o  objectives
p  parameters
pr problems
s  sets
v  variables
```

Displayed declarations may differ in inessential ways from their appearance in your model file, due to transformations that AMPL performs when the model is parsed and translated.

Since the check statements in a model do not have names, AMPL numbers them in the order that they appear. Thus to see the third check statement you would type

```
ampl: show check 1;
```

```
check{p in PROD} : sum{i in ORIG} supply[i,p] == sum{j in DEST}
demand[j,p];
```

By itself, show checks indicates the number of check statements in the model.

6.2 The xref command: displaying model dependencies

The xref command lists all model components that depend on a specified component, either directly (by referring to it) or indirectly (by referring to its dependents). If more than one component is given, the dependents are listed separately for each. Here is an example from multmip3.mod:


```

ampl: xref demand, Trans;

# 2 entities depend on demand:
check 1
Demand

# 5 entities depend on Trans:
total_cost
Supply
Demand
Multi
Min_Ship

```

In general, the form of the command is

```
xref component-list ;
```

where *component-list* is a comma-separated or space-separated list of any combination of set, parameter, variable, objective and constraint names.

6.3 The expand command: displaying model instances

In checking a model and its data for correctness, you may want to look at some of the specific constraints that AMPL is generating. The `expand` command displays all constraints in a given indexed collection,

```

ampl: model multmip3.mod;
ampl: data multmip3.dat;

ampl: expand Max_Serve;

s.t. Max_Serve['GARY']:
    Use['GARY','FRA'] + Use['GARY','DET'] +
        Use['GARY','LAN'] +
    Use['GARY','WIN'] + Use['GARY','STL'] +
        Use['GARY','FRE'] +
    Use['GARY','LAF'] <= 5;

s.t. Max_Serve['CLEV']:
    Use['CLEV','FRA'] + Use['CLEV','DET'] +
        Use['CLEV','LAN'] +
    Use['CLEV','WIN'] + Use['CLEV','STL'] +
        Use['CLEV','FRE'] +
    Use['CLEV','LAF'] <= 5;

s.t. Max_Serve['PITT']:
    Use['PITT','FRA'] + Use['PITT','DET'] +
        Use['PITT','LAN'] +
    Use['PITT','WIN'] + Use['PITT','STL'] +
        Use['PITT','FRE'] +
    Use['PITT','LAF'] <= 5;

```

or specific constraints that you identify:

```

ampl: expand {i in ORIG} Min_Ship[i,'DET'];

s.t. Min_Ship['GARY','DET']:
    Trans['GARY','DET','bands'] +
    Trans['GARY','DET','coils'] +
    Trans['GARY','DET','plate'] - 375*Use['GARY','DET']= 0;

s.t. Min_Ship['CLEV','DET']:
    Trans['CLEV','DET','bands'] +
    Trans['CLEV','DET','coils'] +
    Trans['CLEV','DET','plate'] - 375*Use['CLEV','DET']= 0;

s.t. Min_Ship['PITT','DET']:
    Trans['PITT','DET','bands'] +
    Trans['PITT','DET','coils'] +
    Trans['PITT','DET','plate'] - 375*Use['PITT','DET']= 0;

ampl: expand Multi['CLEV','DET'];

s.t. Multi['CLEV','DET']:
    Trans['CLEV','DET','bands'] +
    Trans['CLEV','DET','coils'] +
    Trans['CLEV','DET','plate'] - 625*Use['CLEV','DET']<=0;

```

The ordering of terms in an expanded constraint does not necessarily correspond to the order of the symbolic terms in the constraint's declaration.

Objectives may be expanded in the same way.

When `expand` is applied to a variable, it lists all of the nonzero coefficients of that variable in the linear terms of objectives and constraints:

```

ampl: expand {i in ORIG} Use[i,'DET'];

Coefficients of Use['GARY','DET']:
    Max_Serve['GARY']           1
    Multi['GARY','DET']         -625
    Min_Ship['GARY','DET']      -375
    total_cost                  1200

Coefficients of Use['CLEV','DET']:
    Max_Serve['CLEV']           1
    Multi['CLEV','DET']         -625
    Min_Ship['CLEV','DET']      -375
    total_cost                  1000

Coefficients of Use['PITT','DET']:
    Max_Serve['PITT']           1
    Multi['PITT','DET']         -625
    Min_Ship['PITT','DET']      -375
    total_cost                  1200

```

In a future implementation, `expand` will also display nonlinear terms in which the variable appears, and the values of bounds and initial values that have appeared in the variable's declaration. In the case of defined variables (Section 13.2), the defining expression will be expanded, too.

To produce an expansion of all variables, objectives and constraints in a model, you can simply type `expand` without any arguments. Since a single `expand` command can produce

a very long listing, you may want to redirect its output to a file by placing filename at the end.

The formatting of numbers in the expanded output is governed by the options `expand_precision` and `expand_round`, which work the same as the `display` command's `display_precision` and `display_round` (Section 10.5).

The output of `expand` reflects the "modeler's view" of the problem; it is based on the model and data as they were initially read and translated. AMPL's presolve phase (Section 10.2) may make significant simplifications to the problem before it is sent to the solver, however. To see the expansion of the "solver's view" of the problem following presolve, use the keyword `solexpand` in place of `expand`.

6.4 Generic synonyms for variables, constraints and objectives

Occasionally it is useful to make a listing or a test that applies to all variables, constraints, or objectives. For this purpose, AMPL provides automatically updated parameters that hold the numbers of variables, constraints, and objectives in the currently generated problem instance:

```
_nvars    number of variables in the current problem
_ncons    number of constraints in current problem
_nobjs    number of objectives in current problem
```

Correspondingly indexed parameters contain the AMPL names of all the components:

```
_varname{1.._nvars}  names of variables in the current
                      problem
_conname{1.._ncons}  names of constraints in the current
                      problem
_objname{1.._nobjs}  names of objectives in the current
                      problem
```

Finally, the following synonyms for the components are made available:

```
_var{1.._nvars}      synonyms for variables in the current
                      problem
_con{1.._ncons}      synonyms for constraints in the current
                      problem
_obj{1.._nobjs}      synonyms for objectives in the current
                      problem
```

These synonyms let you refer to components "generically" by number, rather than by the usual indexed names. Using the variables as an example, `_var[5]` refers to the 5th variable in the problem, `_var[5].ub` to its upper bound, `_var[5].rc` to its reduced cost, and so forth, while `_varname[5]` is a string giving the variable's true AMPL name.

Generic names are useful for tabulating properties of all variables, where the variables have been defined in several different `var` declarations:

```
ampl: model net3.mod
ampl: data net3.dat
```

```

ampl: solve;
MINOS 5.4: optimal solution found.
3 iterations, objective 1819

ampl: display {j in 1.._nvars}
ampl?  (_varname[j],_var[j],_var[j].ub,_var[j].rc);

:      _varname[j]      _var[j] _var[j].ub      _var[j].rc  :=
1  "PD_Ship['NE']"      250      250      -0.5
2  "PD_Ship['SE']"      200      250      -1.11022e-16
3  "DW_Ship['NE','BOS']"  90       90       0
4  "DW_Ship['NE','EWR']"  100      100      -1.1
5  "DW_Ship['NE','BWI']"  60       100      0
6  "DW_Ship['SE','EWR']"  20       100      2.22045e-16
7  "DW_Ship['SE','BWI']"  60       100      2.22045e-16
8  "DW_Ship['SE','ATL']"  70       70       0
9  "DW_Ship['SE','MCO']"  50       50       0
;

```

Another use is to list all variables having some property, such as being positive in the optimal solution:

```

ampl: display {j in 1.._nvars:
ampl?  _var[j] < _var[j].ub - 0.00001} _varname[j];

_varname[j] [*] :=
2  "PD_Ship['SE']"
5  "DW_Ship['NE','BWI']"
6  "DW_Ship['SE','EWR']"
7  "DW_Ship['SE','BWI']"
;

```

The same comments apply to constraints and objectives. More precise formatting of this information can be obtained by using the `printf` command (Sections 10.6, A.13.1) instead of `display`.

As in the case of the `expand` command, these parameters and generic synonyms reflect the "modeler's view" of the problem; their values are determined from the model and data as they were initially read and translated. AMPL's presolve phase (Section 10.2) may make significant simplifications to the problem before it is sent to the solver, however. To work with parameters and generic synonyms that reflect the "solver's view" of the problem following presolve, replace `_` by `_s` in the names given above; for example in the case of variables, use `_snvars`, `_svarname` and `_svar`.

6.5 Summary model information

The following sets are automatically updated to contain the names of model components declared so far:

Set:	Contains names of all declared:
<code>_SETS</code>	sets
<code>_PARS</code>	parameters
<code>_VARS</code>	variables
<code>_OBS</code>	objectives
<code>_CONS</code>	constraints
<code>_FUNCS</code>	user-defined functions

<code>_PROBS</code>	problems
<code>_ENVS</code>	environments

Two new functions, `arity(s)` and `indexarity(s)`, provide additional information about the component whose name is given by the character string `s`:

`arity(s)`

the dimension of `s`, or number of components in each member of `s`, provided `s` is a set; otherwise 0

`indexarity(s)`

the dimension of the set over which `s` is indexed, or 0 if `s` is not indexed over a set, or -1 if `s` has not been declared

Used together, these features can provide a summary of model components and their dimensions:

```

ampl: display {i in _SETS} arity(i);
arity(i) [*] :=
DEST 1
ORIG 1
PROD 1
;

ampl: display {i in _VARS} indexarity(i);
indexarity(i) [*] :=
Trans 3
Use 2
;

```


7 Looping and Testing 1: Writing "Scripts" in the AMPL Command Language

New `for`, `repeat`, `if` and related commands let you write small programs, or *scripts*, in the AMPL command language. Another collection of new commands let you step through a script, for purposes of observation or debugging.

This section introduces these commands, using formatted printing and sensitivity analysis as examples. For those who want to get started faster, a syntax summary for all these commands appears at the end of this section.

The following section describes additional commands and features that permit you to express new algorithmic schemes by means of AMPL scripts.

7.1 Running scripts

Just as the `model` command reads files of model declarations, and the `data` command reads files of data statements, the `AMPL commands` command reads files containing AMPL scripts -- lists of commands -- that are to be executed. Simple scripts can save you the trouble of typing the same sequences of commands again and again. More sophisticated scripts can carry out repetitive actions that would be impractical to type one at a time, as we will show shortly.

As an example, consider how we might perform a simple sensitivity analysis on the multi-period production problem of Section 4.2. Only 32 hours of production time are available in week 3, compared to 40 hours in the other weeks. Suppose that we want to see how much extra profit could be gained for each extra hour in week 3. We can accomplish this by repeatedly solving, displaying the solution values, and increasing `avail[3]`:

```
ampl: model steelT.mod;  
ampl: data steelT.dat;  
  
ampl: solve;  
MINOS 5.4: optimal solution found.  
15 iterations, objective 515033
```

```

ampl: display total_profit steelT.sens;
ampl: option display_1col 0;
ampl: option omit_zero_rows 0;
ampl: display Make steelT.sens;
ampl: display Sell steelT.sens;
ampl: option display_1col 20;
ampl: option omit_zero_rows 1;
ampl: display Inv steelT.sens;

```

```

ampl: let avail[3] := avail[3] + 5;

```

```

ampl: solve;
MINOS 5.4: optimal solution found.
6 iterations, objective 532033

```

```

ampl: display total_profit steelT.sens;
ampl: option display_1col 0;
ampl: option omit_zero_rows 0;
ampl: display Make steelT.sens;
ampl: display Sell steelT.sens;
ampl: option display_1col 20;
ampl: option omit_zero_rows 1;
ampl: display Inv steelT.sens;

```

```

ampl: let avail[3] := avail[3] + 5;

```

```

ampl: solve;
MINOS 5.4: optimal solution found.
6 iterations, objective 549033

```

```

ampl:

```

To try all integer values of `avail[3]` from 32 to 77, we would complete another seven cycles in the same way. All of the solution displays are saved in the file `steelT.sens`, although we could just as well have made them appear on the screen.

To avoid having to type the same commands again and again, we can create a new file listing the commands to be repeated:

```

solve;

display total_profit steelT.sens;

option display_1col 0;
option omit_zero_rows 0;
display Make steelT.sens;
display Sell steelT.sens;

option display_1col 20;
option omit_zero_rows 1;
display Inv steelT.sens;

let avail[3] := avail[3] + 5;

```

If we call this file `steelT.sal`, then we can execute all the commands in it by typing just the one line commands `steelT.sal`:


```

ampl: model steelT.mod;
ampl: data steelT.dat;

ampl: commands steelT.sal;
MINOS 5.4: optimal solution found.
15 iterations, objective 515033

ampl: commands steelT.sal;
MINOS 5.4: optimal solution found.
6 iterations, objective 532033

ampl: commands steelT.sal;
MINOS 5.4: optimal solution found.
6 iterations, objective 549033

ampl: commands steelT.sal;
MINOS 5.4: optimal solution found.
7 iterations, objective 565193

ampl:

```

We refer to the contents of a file like `steelT.sal` as a *script*. The looping and testing commands to be described in this section are most valuable when they are used within scripts.

A script can itself contain a `commands` command that refers to another script. Scripts can be nested in this way to a depth of up to 10.

[[Can alternatively use the `include` command -- makes no difference here, but does inside a loop as in the next section]]

7.2 Iterating over a set

The above example still requires that some command be typed over and over again. AMPL provides looping commands that can do this work automatically, with various options to determine how long the looping should continue.

We begin with the `for` command, which executes a statement or collection of statements once for each member of some set. To execute our multi-period production problem sensitivity analysis script four times, for example, we need only type a single `for` command followed by the command that we want to repeat:

```

ampl: model steelT.mod;
ampl: data steelT.dat;

ampl: for {1..4} commands steelT.sal;

MINOS 5.4: optimal solution found.
15 iterations, objective 515033
MINOS 5.4: optimal solution found.
6 iterations, objective 532033
MINOS 5.4: optimal solution found.
6 iterations, objective 549033
MINOS 5.4: optimal solution found.
7 iterations, objective 565193

ampl:

```

The expression between `for` and the command can be any AMPL indexing expression, as we will see in later examples.

As an alternative, we can add the `for` command to the script. We then want it to loop over a whole series of commands; this is accomplished by enclosing the commands in braces:

```
model steelT.mod;
data steelT.dat;

for {1..4} {
    solve;

    display total_profit steelT.sens;

    option display_lcol 0;
    option omit_zero_rows 0;
    display Make steelT.sens;
    display Sell steelT.sens;

    option display_lcol 20;
    option omit_zero_rows 1;
    display Inv steelT.sens;

    let avail[3] := avail[3] + 5;
}
```

If this script is stored in `steelT.sa2`, then the whole iterated sensitivity analysis is carried out by simply typing commands `steelT.sa2`.

This latter approach tends to be clearer and easier to work with, particularly as we make the loop more sophisticated. As a first example, consider how we would go about compiling a table of the objective and the dual value on constraint `time[3]`, for successive values of `avail[3]`. A script for this purpose is shown in Figure 1. After the model and data are read, the script provides additional declarations for the table of values:

```
set AVAIL3;
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};
```

The set `AVAIL3` will contain all the different values for `avail[3]` that we want to try; for each such value `a`, `avail3_obj[a]` and `avail3_dual[a]` will be the associated objective and dual values. Once these are set up, we assign the set value to `AVAIL3`,

```
let AVAIL3 := avail[3] .. avail[3] + 15 by 5;
```

and then use a `for` loop to iterate over this set:

```
for {a in AVAIL3} {

    let avail[3] := a;

    solve;

    let avail3_obj[a] := total_profit;
    let avail3_dual[a] := time[3].dual;
```

```
}
```

We see here that a `for` loop can be over an arbitrary set, and that the index running over the set (`a` in this case) can be used in statements within the loop. After the loop is complete, the desired table is produced by displaying `avail3_obj` and `avail3_dual`, as shown at the end of the script in Figure 1. If this script is stored in `steelT.sa3`, then the desired results are produced with a single command:

```
ampl: include steelT.sa3

:   avail3_obj avail3_dual :=
32   515033      3400
37   532033      3400
42   549033      3400
47   565193      2980
;
```

In this case we have suppressed the messages from the solver, by including the command `option solver_msg 0` in our script.

```
-----
model steelT.mod;
data steelT.dat;

set AVAIL3;
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};

let AVAIL3 := avail[3] .. avail[3] + 15 by 5;

option solver_msg 0;

for {a in AVAIL3} {
    let avail[3] := a;

    solve;

    let avail3_obj[a] := total_profit;
    let avail3_dual[a] := time[3].dual;
}

display avail3_obj, avail3_dual;
-----
```

Figure 1. A script for recording sensitivity to the value of the parameter `avail[3]`, using a `solve` within a `for` loop (`steelT.sa3`).

AMPL's `for` loops are also very convenient for generating formatted tables. Suppose that after solving the multi-period production problem, you want to display sales both in tons and as a percentage of the market limit. You could use a `display` command to produce a table like this:

```
ampl: display {t in 1..T, p in PROD}
ampl?   (Sell[p,t], 100*Sell[p,t]/market[p,t]);

:       Sell[p,t] 100*Sell[p,t]/market[p,t]      :=
```

```

1 bands      6000      100
1 coils      307       7.675
2 bands      6000      100
2 coils      2500      100
3 bands      1400       35
3 coils      3500      100
4 bands      2000      30.7692
4 coils      4200      100
;

```

By writing a script that uses AMPL's `printf` command (Section A.13.1), you can create a much more effective table:

```

ampl: commands steelT.tab1

```

```

SALES      bands      coils
week 1      6000 100.0%    307  7.7%
week 2      6000 100.0%   2500 100.0%
week 3      1399  35.0%   3500 100.0%
week 4      1999  30.8%   4200 100.0%

```

The script to write this table can be as short as two `printf` commands:

```

printf "\n%s%14s%17s\n", "SALES", "bands", "coils";

printf {t in 1..T}: "week %d%9d%7.1f%%%9d%7.1f%%\n", t,
    Sell["bands",t], 100*Sell["bands",t]/market["bands",t],
    Sell["coils",t], 100*Sell["coils",t]/market["coils",t];

```

This approach is undesirably restrictive, however, because it assumes that there will always be two products and that they will always be named `coils` and `bands`. In fact the `printf` statement cannot write a table in which both the number of rows and the number of columns depend on the data, because the number of entries in its format string is always fixed.

A more generally applicable script for generating the above table, using a `for` loop over `1..T`, is shown in Figure 2. Each pass through the loop generates one row of the table. There are more `printf` statements than in the previous example, but they are shorter and simpler. We use several statements to write the contents of each line; `printf` does not begin a new line except where `\n` appears in its format string.

```

-----
printf "\nSALES";
printf {p in PROD}: "%14s    ", p;
printf "\n";

for {t in 1..T} {
    printf "week %d", t;

    for {p in PROD} {
        printf "%9d", Sell[p,t];
        printf "%7.1f%%", 100 * Sell[p,t]/market[p,t];
    }

    printf "\n";
}
-----

```

Figure 2. A script for generating a formatted sales table, using nested for loops (steelT.tab).

Within the "outer" loop over `1..T` we generate each line's product entries by use of an "inner" loop over `PROD`. Loops can in general be nested to any depth, and may be over any set that can be represented by an AMPL set expression. There is one pass through the loop for every member of the set, and if the set is ordered -- any set of numbers like `1..T`, or a set declared `ordered` or `circular` -- then the order of the passes is determined by the ordering of the set. If the set is unordered (like `PROD`) then AMPL chooses the order of the passes, but the choice is the same every time; the Figure 2 script relies on this consistency to insure that all of the entries in one column of the table refer to the same product.

7.3 Iterating subject to a condition

A second kind of AMPL looping construct, the `repeat` statement, continues iterating as long as some logical condition is satisfied.

Returning to the multi-period production example, we observe that the dual value on the constraint `time[3]` provides an upper limit on the additional profit that can be realized from each extra hour added to `avail[3]`. When `avail[3]` is made sufficiently large, so that there is more third-week capacity than can be used, the associated dual value falls to zero and further increases in `avail[3]` have no effect on the optimal solution. Either the expression `time[3]` alone, or `time[3].dual`, represents the dual value in AMPL expressions.

We can specify that looping should stop once the dual value falls to zero, by writing a `repeat` statement that has one of the following forms:

```
repeat while time[3].dual < 0 { . . . };
repeat until time[3].dual = 0 { . . . };

repeat { . . . } while time[3].dual < 0;
repeat { . . . } until time[3].dual = 0;
```

The loop body, here indicated by the placeholder `{ . . . }`, must be enclosed within braces. Passes through the loop continue as long as the condition after `while` is true, or as long as the condition after `until` is false. A condition written before the loop body is tested before every pass; if a `while` condition is false or an `until` condition is true before the first pass, then the loop is never executed. A condition written after the loop body is tested after every pass, so that the loop is executed at least once in every case.

A complete script using `repeat` is shown in Figure 3. The `until` phrase is placed after the loop body, so that `time[3].dual` will not be tested until after a solve has been executed in the first pass. Two other features of this script are worth noting, as they are relevant to many AMPL scripts of this kind.

```
-----
model steelT.mod;
data steelT.dat;

option solution_precision 10;
option solver_msg 0;
```

```

set AVAIL3 default {};
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};

param avail3_step := 5;

repeat {

    solve;

    let AVAIL3 := AVAIL3 union {avail[3]};
    let avail3_obj[avail[3]] := total_profit;
    let avail3_dual[avail[3]] := time[3].dual;

    let avail[3] := avail[3] + avail3_step;

} until time[3].dual = 0;

display avail3_obj, avail3_dual;
-----

```

Figure 3. A script for recording sensitivity to the value of `avail[3]`, using a repeat statement to continue until the associated dual value is zero (`steelT.sa4`).

At the beginning of the script, we don't know how many passes the repeat statement will make through the loop. Thus we cannot determine `AVAIL3` in advance as we did in Figure 1. Instead, we declare it initially to be the empty set,

```

set AVAIL3 default {};
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};

```

and add each new value of `avail[3]` to it after solving:

```

let AVAIL3 := AVAIL3 union {avail[3]};
let avail3_obj[avail[3]] := total_profit;
let avail3_dual[avail[3]] := time[3].dual;

```

By adding a new member to `AVAIL3`, we also create new components of the parameters `avail3_obj` and `avail3_dual` that are indexed over `AVAIL3`, and so we can proceed to assign the appropriate values to these components. Any change to an AMPL set is propagated to all declarations that use the set, in the same way that any change to a parameter is propagated.

Because numbers in the computer are represented with a limited number of bits of precision, a solver may return values that differ very slightly from the solution that would be computed using exact arithmetic. Ordinarily you don't see this, because AMPL's display command is set by default to round values to six significant digits. Compare what is shown when rounding is dropped, by setting `display_precision` to 0:

```

ampl: display Make;

Make [*,*] (tr)
:  bands  coils  :=
1   5990   1407
2   6000   1400
3   1400   3500

```

```

4    2000    4200
;

ampl: option display_precision 0;
ampl: display Make;

Make [*,*] (tr)
:      bands                coils      :=
1    5989.999999999999    1407.0000000000002
2    6000                1399.9999999999998
3    1399.9999999999995    3500
4    1999.9999999999993    4200
;

```

These seemingly tiny differences can have undesirable effects whenever a script makes a comparison that uses values returned by the solver. The rounded table would lead you to believe that `Make["coils",2] = 1400` is true, for example, whereas from the second table you can see that really it is false.

You can avoid this kind of surprise by writing your tests more carefully; instead of `until time[3].dual = 0`, for instance, you might say `until time[3].dual <= 0.0000001`. Alternatively, you can instruct AMPL to round all solution values that are returned by the solver, so that numbers that are supposed to be equal really do come out equal. The statement

```
option solution_precision 10;
```

toward the beginning of Figure 3 has this effect; it states that solution values are to be rounded to 10 significant digits. These and related options are discussed and illustrated in Section 10.5 of the AMPL book.

Note finally that the script declares set `AVAIL3` as default `{ }` rather than `:= { }`. The former allows `AVAIL3` to be changed by `let` commands as the script proceeds, whereas the latter permanently defines `AVAIL3` to be the empty set.

7.4 Testing a condition

AMPL's new `if` command takes a specified action conditionally. In the simplest case, the action is the execution of one AMPL command:

```
if Make["coils",2]
```

The action may also be a series of AMPL commands grouped by braces as in the `for` and `repeat` commands:

```
if Make["coils",2]
```

An optional `else` specifies an alternative action that also may be either a single command,

```
if Make["coils",2]
```

or a group of commands:

```
if Make["coils",2]
```

AMPL executes these commands by first evaluating the logical expression following `if`. If the expression is true then the command or commands following then are executed. If

the expression is false then the command or commands following else, if any, are executed.

The if command is most useful for regulating the flow of control in scripts. In Figure 2, we could suppress the many occurrences of 100% by placing the statement that prints `Sell[p,t]/market[p,t]` inside an if:

```
if Sell[p,t] < market[p,t] then
    printf "%7.1f%%", 100 * Sell[p,t]/market[p,t];
else printf "    --- ";
```

In Figure 3, we can use an if command in the repeat loop to test whether the dual value has changed since the previous pass through the loop:

```
let avail[3] := 1;
param avail3_step := 1;
param previous_dual default Infinity;

repeat while previous_dual = 0 {

    solve;

    if time[3].dual < previous_dual then {
        let AVAIL3 := AVAIL3 union {avail[3]};
        let avail3_obj[avail[3]] := total_profit;
        let avail3_dual[avail[3]] := time[3].dual;
        let previous_dual := time[3].dual;
    }

    let avail[3] := avail[3] + avail3_step;
}
```

When used within the Figure 3 script, this loop creates a table that has exactly one entry for each different dual value discovered.

If you run the Figure 3 script with the above changes (steelT.sa5), you will find that the dual value changes when `avail[3]` is increased from 22 to 23 hours. Figure 4 exhibits a script (steelT.sa6) that carries out a binary search to determine more exactly (to 7 decimal places) the value of `avail[3]` where the dual value changes:

```
ampl: commands steelT.sa6;

Dual value 3620.000000 for avail[3] < 22.807142
Dual value 3500.000000 for avail[3] 22.807144
```

The if in this script,

```
if time[3].dual = dual_lower
then let avail3_lower := avail[3];
else let avail3_upper := avail[3];
```

is key to the binary search algorithm, since it determines which half of the current search interval is removed at each step. The equality at the beginning of this if statement is another example of a comparison involving numbers returned by the solver. Thus as before the option `solution_precision` is set to 10 at the start of the script; in this case the results would be wrong otherwise.

```
-----
model steelT.mod;
```



```

data steelT.dat;

option solution_precision 10;
option solver_msg 0;

param avail3_lower default 22; param dual_lower;
param avail3_upper default 23; param dual_upper;

let avail[3] := avail3_lower;
solve; let dual_lower := time[3].dual;

let avail[3] := avail3_upper;
solve; let dual_upper := time[3].dual;

repeat {

    let avail[3] := (avail3_lower + avail3_upper) / 2;

    solve;

    if time[3].dual = dual_lower
        then let avail3_lower := avail[3];
        else let avail3_upper := avail[3];

} until (avail3_upper - avail3_lower) / avail[3] < 0.0000001;

printf "Dual value %11.6f for avail[3] < %9.6f\n",
    dual_lower, avail3_lower;

printf "Dual value %11.6f for avail[3] %9.6f\n",
    dual_upper, avail3_upper;
-----

```

Figure 4. A script that performs a binary search to determine the level of avail[3] at which the associated dual value changes (steelT.sa6).

The Figure 4 script requires an initial interval, bounded by avail3_lower and avail3_upper, such that the dual value changes once as avail[3] moves through the interval. If the dual value might change more than once, then the following if ensures that the script will find the largest value of avail[3] at which there is a change:

```

if time[3].dual < dual_upper
then {
    let avail3_lower := avail[3];
    let dual_lower := time[3].dual;
}
else let avail3_upper := avail[3];

```

A further refinement counts the number of times that the search detects an additional change in the dual value within the initial interval:

```

if time[3].dual = dual_lower
then let avail3_lower := avail[3];
else if time[3].dual = dual_upper
then let avail3_upper := avail[3];
else {
    let other_bkpts := other_bkpts + 1;
    let avail3_lower := avail[3];
    let dual_lower := time[3].dual;
}

```

```
};
```

Here we see how the statement following else can be another if, giving a chain of tests. [[Each nested else paired with nearest available if.]]

7.5 Terminating a loop

Two other new AMPL commands work with looping statements to make some scripts easier to write. The continue command terminates the current pass through a for or while loop; all further statements in the current pass are skipped, and execution continues with the start of the next pass (if any). The break command completely terminates a for or while loop, sending control immediately to the statement following the end of the loop.

As an example of both these commands, Figure 6 exhibits another way of rewriting the loop from the Figure 3 script, so that a table entry is made only when there is a change in the dual value associated with avail[3]. After solving, we test to see if the new dual value is equal to the previous one:

```
if time[3].dual = previous_dual then continue;
```

If yes, there is nothing to be done for this value of avail[3]. Thus we use the continue command to jump to the end of the current pass; execution resumes with the next pass, starting at the beginning of the loop.

```
-----
model steelT.mod;
data steelT.dat;

option solution_precision 10;
option solver_msg 0;

set AVAIL3 default {};
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};

let avail[3] := 0;
param previous_dual default Infinity;

repeat {
    let avail[3] := avail[3] + 1;

    solve;

    if time[3].dual = previous_dual then continue;

    let AVAIL3 := AVAIL3 union {avail[3]};
    let avail3_obj[avail[3]] := total_profit;
    let avail3_dual[avail[3]] := time[3].dual;

    if time[3].dual = 0 then break;

    let previous_dual := time[3].dual;
}

display avail3_obj, avail3_dual;
```

Figure 5. An alternative to the sensitivity analysis script in Figure 3, using the `continue` and `break` commands (steelT.sa7).

After adding an entry to the table, we test to see if the dual value has fallen to zero:

```
if time[3].dual = 0 then break;
```

If yes, we are done with the loop. We use the `break` command to jump out of the loop; execution passes to the `display` command that follows the loop in the script. Since the `repeat` statement in this example has no `while` or `until` conditions, it relies on the `break` command for termination.

When a `break` or `continue` lies within more than one loop, it applies only to the innermost loop. This convention generally has the effect desired. As an example, consider how we could expand Figure 5 to perform a separate sensitivity analysis on each `avail[t]`. The `repeat` loop would be nested in a `for {t in 1..T}` loop (steelT.sa7a), but the `continue` and `break` commands would apply to the inner `repeat` as before.

There do exist situations in which the logic of a script requires breaking out of a loop other than the inner loop. In the script of Figure 5, for instance, we can imagine that instead of stopping when the `time[3].dual` is zero,

```
if time[3].dual = 0 then break;
```

we want to stop when `time[t].dual` falls below 2700 for any `t`. One way to implement this criterion is:

```
for {t in 1..T}
  if time[t].dual
```

This statement does not have the desired effect, however, because `break` applies only to the inner `for` loop that contains it, rather than to the outer `repeat` loop as we desire. To deal with these kinds of situations, AMPL lets you give a name to any loop, so that any `break` or `continue` may specify by name the loop to which it should apply. Using this feature, the outer loop in our example could be named `sens_loop`:

```
repeat sens_loop {
  . . .
}
```

and the inner loop would be

```
for {t in 1..T}
  if time[t].dual
```

The loop name always comes immediately after `repeat` or `for`, and after `break` or `continue`.

7.6 Stepping through a script

If you think that a script might not be doing what you want it to, you can direct AMPL to step through it one command at a time. This facility can be used to provide an elementary form of "symbolic debugger" for AMPL scripts.

To step through a script that does not execute any other scripts, simply reset the AMPL option `single_step` from its default value of zero to one. For example, here is how you might begin stepping through the script in Figure 4:

```
ampl: option single_step 1;

ampl: commands steelT.sa6;

steelT.sa6:3(19)    data ...

<2ampl:
```

The expression `steelT.sa6:3(19)` gives the file name, line number and character number where AMPL has stopped in its processing of the script. It is followed by the beginning of the next command (`data`) to be executed. On the next line you are returned to the `ampl` prompt; the `<2` in front means [[level of prompt, to be explained]].

At this point you may use the `step` command to execute individual commands of the script. Type `step` by itself to execute one command,

```
<2ampl: step
steelT.sa6:5(37)    option ...

<2ampl: step
steelT.sa6:6(67)    option ...

<2ampl: step
steelT.sa6:11(188)  let ...

<2ampl:
```

or type `step` followed by a number to execute that number of commands:

```
<2ampl: step 3
steelT.sa6:14(258)  let ...

<2ampl:
```

Every command is counted except those having to do with model declarations, such as `model` and `param` in this example.

Each step returns you to an AMPL prompt as before. You may continue in this way, but at some point you will want to display some values to see if the script is working as you intended:

```
<2ampl: display avail[3], dual_lower;
avail[3] = 22
dual_lower = 3620

<2ampl: step 3
steelT.sa6:17(328)  repeat ...

<2ampl: display avail[3], dual_upper;
avail[3] = 23
dual_upper = 3500

<2ampl:
```

Any series of AMPL commands may be typed while single-stepping. After each command, the <2ampl prompt returns to remind you that you are still in this mode and may use step to continue executing the script.

At this point, execution of the script has reached the beginning of a repeat loop. The step command takes you into the first pass through the loop,

```
<2ampl: step
steelT.sa6:19(341)    let ...

<2ampl: step
steelT.sa6:21(396)    solve ...

<2ampl:
```

and into the branch of the enclosed if command that is executed:

```
<2ampl: step
steelT.sa6:23(407)    if ...

<2ampl: step
steelT.sa6:24(448)    let ...

<2ampl:
```

Further use of step will take you to the next pass of the loop, back at line 19,

```
<2ampl: step
steelT.sa6:19(341)    let ...

<2ampl: step
steelT.sa6:21(396)    solve ...

<2ampl:
```

and similarly through all of the subsequent passes, after which control passes to the command following repeat.

To help you step through lengthy compound commands (for, repeat, or if) AMPL provides several alternatives to step. The next command steps past a compound command rather than into it. In our example, next would cause the entire repeat command to be executed, stopping again only at the following printf command on line 29:

```
<2ampl: step
steelT.sa6:17(328)    repeat ...

<2ampl: next
steelT.sa6:29(586)    printf ...

<2ampl:
```

Type next n to step past n commands in this way.

The commands skip and skip n work like step and step n, except that they skip the next 1 or n commands in the script rather than executing them.

8 Looping and Testing 2: Implementing Algorithms through AMPL Scripts

You have seen in the previous section how "scripts" of AMPL commands can be set up to run as programs that perform repetitive actions. In several of the examples, a script solves a series of related model instances, by including a solve statement inside a loop. The result is a simple kind of sensitivity analysis algorithm, programmed in the AMPL command language.

This section introduces a variety of AMPL features that help make algorithmic scripts easier to write, and that make more sophisticated scripts possible.

8.1 Alternating between named problems

Up to this point, our AMPL scripts have been confined to solving one model repeatedly, with varying data. Much more powerful algorithmic procedures can be constructed by use of two models. An optimal solution for either model yields new data for the other, and the two are solved in alternation in such a way that some termination condition must eventually be reached.

To use two models in this manner, a script must have some way of switching between one model and the other. After first indicating how switching can be accomplished using only previously defined AMPL features, we show how the same results can be accomplished more clearly and efficiently by use of new AMPL features for defining separately named problems and environments.

We illustrate these possibilities through a script for an elementary form of the "roll trim" or "cutting stock" problem, using a well-known procedure due to Gilmore and Gomory. In the interest of brevity, we give only a sketchy description of the procedure here; a detailed derivation and discussion can be found in Chapter 13 of Chvatal's text. (Vasek Chvatal, *Linear Programming*. New York: W.H. Freeman and Company, 1983.)

In a simple roll trim problem, we wish to cut up long raw widths of some commodity -- such as rolls of paper -- into a combination of smaller widths that meet given orders with as little waste as possible. This problem can be viewed as deciding, for each raw-width roll, where the cuts should be made so as to produce one or more of the ordered smaller

widths. Expressing such a problem in terms of decision variables is awkward, however, and leads to an integer program very difficult to solve except for very small instances.

To derive a more manageable model, the Gilmore-Gomory procedure defines a cutting *pattern* to be any one feasible way in which a raw roll can be cut up. A pattern thus consists of a certain number of rolls of each ordered width, such that their total width does not exceed the raw width. If (as in Exercise 2-6) the raw width is 110", and there are demands for widths of 20", 45", 50", 55" and 75", then two rolls of 45" and one of 20" make an acceptable pattern, as do one of 50" and one of 55" (with 5" of waste). Given a list of these patterns, we can recast the roll-trim problem so that the decision variables are indexed over patterns. For each particular pattern, the associated variable represents the numbers of raw rolls to be cut using that pattern. The objective, naturally, is to cut as few raw rolls as possible while meeting all demands.

Figure 1 gives two simple linear programs that can work together to find an efficient roll-cutting plan. The cutting optimization problem (Figure 1a) finds the minimum number of raw rolls that need be cut, given a collection of cutting patterns that may be used. If we relax the integrality requirement on this problem -- so that the solution may involve fractions of a roll -- then the solver will return "dual values" or "prices" on the constraints Fill[i]. The concept of a price on a constraint is introduced briefly by Section 10.7 of the AMPL book, and at much greater length in any LP textbook. For present purposes, however, it is enough to know that, given these prices, there exists a pattern generation problem (Figure 1b) that does one of two things: either it finds a new pattern that can be used (in the cutting optimization) to reduce the number of raw rolls needed, or it determines that no such new pattern exists. The pattern generation problem is a particularly simple one-constraint integer linear program, known as a knapsack problem.

```

-----
param roll_width 0;          # width of raw rolls

set WIDTHS;                # set of widths to be cut
param orders {WIDTHS} 0;    # number of each width to be cut

param nPAT integer = 0;     # number of patterns
set PATTERNS := 1..nPAT;    # set of patterns

param nbr {WIDTHS,PATTERNS} integer = 0;

    check {j in PATTERNS}:
        sum {i in WIDTHS} i * nbr[i,j] <= roll_width;

                                # defn of patterns: nbr[i,j] = number
                                # of rolls of width i in pattern j

var Cut {PATTERNS} integer = 0;    # rolls cut using each pattern

minimize Number:                # minimize total raw rolls cut
    sum {j in PATTERNS} Cut[j];

subj to Fill {i in WIDTHS}:
    sum {j in PATTERNS} nbr[i,j] * Cut[j] = orders[i];

                                # for each width, total
                                # rolls cut meets total orders
-----

```


Figure 1a. A pattern-based model for the cutting optimization problem (`cut.mod`). Given a collection of patterns, it determines how many raw rolls to cut using each pattern, so as to minimize total raw rolls used.

```
-----
param price {WIDTHS} default 0.0; # prices from cutting opt

var Use {WIDTHS} integer = 0;      # numbers of each width in
pattern

minimize Reduced_Cost:
    1 - sum {i in WIDTHS} price[i] * Use[i];

subj to Width_Limit:
    sum {i in WIDTHS} i * Use[i] <= roll_width;
-----
```

Figure 1b. A knapsack model for the pattern generation problem (`cut.mod`, continued). Given values `price[i]` generated by the cutting optimization problem (with integrality relaxed), it looks for a pattern that may be added to reduce the number of raw rolls required.

We can search for a good cutting plan by solving these two problems repeatedly in alternation. First the cutting optimization problem generates some prices, then the pattern generation problem uses the prices to generate a new pattern, and then the procedure repeats with the collection of patterns extended by one. We stop repeating when the pattern generation problem indicates that no new pattern can lead to an improvement. We then have the best possible solution in terms of (possibly) fractional numbers of raw rolls cut. We may make one last run with the integrality restriction restored, to get the best integral solution using the patterns generated, or we may simply round the fractional numbers of rolls to the nearest integers if that gives an acceptable result.

This is the Gilmore-Gomory procedure. In outline, its steps may be described as follows:

Pick initial patterns sufficient to meet demand.

Repeat

 Solve the (fractional) cutting optimality problem.

 Let `price[i]` equal `Fill[i].dual` for each pattern `i`.

 Solve the pattern generation problem.

 If the optimal value is

 then add a new pattern that cuts `Use[i]` rolls of each width `i`;

 else find a final integer solution and stop.

An easy way to initialize is to generate one pattern for each width, containing as many copies of the width as will fit inside the raw roll. These patterns easily cover any demands.

An implementation as an AMPL script is shown in Figure 2. The statement model `cut.mod` at the very beginning reads a file that contains both the cutting optimization and pattern generation models from in Figure 1. Since these models have no variables or constraints in common, it would be possible to write the script with simple solve statements using alternating objective functions; in outline,

```
repeat {
    objective Number;
```

```

        solve;
        ...
        objective Reduced_Cost;
        solve;
        ...
    }

```

If we took this approach, however, every `solve` would send the solver all of the variables and constraints generated by both models. Especially for larger and more complex iterative procedures, such as arrangement is inefficient and prone to error.

We could instead insure that only the immediately relevant variables and constraints are sent to the solver, by using AMPL's `fix` and `drop` commands to suppress the others. Then the outline of our loop would look like this:

```

repeat {
    unfix Cut; restore Fill; objective Number;
    fix Use; drop Width_Limit;
    solve;
    ...
    unfix Use; restore Width_Limit; objective Reduced_Cost;
    fix Cut; drop Fill;
    solve;
    ...
}

```

Before each `solve`, the previously fixed variables and dropped constraints must also be brought back, by use of `unfix` and `restore`. This approach is efficient, but it remains highly error-prone, and makes scripts difficult to read.

As an alternative, therefore, AMPL allows alternative models to be distinguished by use of the new problem statement seen in Figure 2:

```

problem Cutting_Opt: Cut, Fill, Number;
option relax_integrality 1;

problem Pattern_Gen: Use, Width_Limit, Reduced_Cost;
option relax_integrality 0;

```

The first statement defines a problem named `Cutting_Opt` that consists of the `Cut` variables, the `Fill` constraints, and the objective `Number`. This statement also makes `Cutting_Opt` the current problem; uses of the AMPL `var`, `minimize`, `maximize`, `subject to`, and `option` statements now apply to this problem only. Thus by setting `option relax_integrality` to 1 above, for example, we assure that the integrality condition on the `Cut` variables will be relaxed whenever `Cutting_Opt` is current. In a similar way, we define a problem `Pattern_Gen` that consists of the `Use` variables, the `Width_Limit` constraint, and the objective `Reduced_Cost`; this in turn becomes the current problem, and this time we set `relax_integrality` to 0 because only integer solutions to this problem are meaningful. (The next subsection gives more detailed rules for how a problem and its environment can be defined and made current.)

The `for` loop in Figure 2 creates the initial cutting patterns, after which the main `repeat` loop carries out the Gilmore-Gomory procedure as described previously. The statement

```

solve Cutting_Opt;

```

acts both to restore `Cutting_Opt` as the current problem, along with its environment, and to solve the associated linear program. Then we say

```
let {i in WIDTHS} price[i] := Fill[i].dual;
```

to assign the optimal dual prices from `Cutting_Opt` to the parameters `price[i]` will be used by `Pattern_Gen`. All set and parameter data is global in AMPL, so that it can be referenced or changed whatever the current problem.

The second half of the main loop makes problem `Pattern_Gen` and its environment current, and sends the associated integer program to the solver. If the resulting objective is sufficiently negative, the pattern returned by the `Use[i]` variables is added to the data that will be used by `Cutting_Opt` in the next pass through the loop. Otherwise no further progress can be made, and the loop is terminated.

```
-----
model cut.mod;
data cut.dat;

option solver cplex;
option solution_round 6;

problem Cutting_Opt: Cut, Number, Fill;
option relax_integrality 1;

problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
option relax_integrality 0;

let nPAT := 0;

for {i in WIDTHS} {
  let nPAT := nPAT + 1;
  let nbr[i,nPAT] := floor (roll_width/i);
  let {i2 in WIDTHS: i2 <> i} nbr[i2,nPAT] := 0;
};

repeat {
  solve Cutting_Opt;

  let {i in WIDTHS} price[i] := Fill[i].dual;

  solve Pattern_Gen;

  if Reduced_Cost < -0.00001 then {
    let nPAT := nPAT + 1;
    let {i in WIDTHS} nbr[i,nPAT] := Use[i];
  }
  else break;
};

option Cutting_Opt.relax_integrality 0;
solve Cutting_Opt;

display nbr;
display Cut;
-----
```

Figure 2. A script to implement the Gilmore-Gomory procedure for the cutting-stock problem, using the problem statement and other new AMPL features (`cut.run`).

The script concludes with the following statements, to solve for the best integer solution using all patterns generated:

```
option Cutting_Opt.relax_integrality 0;
solve Cutting_Opt;
```

The expression `Cutting_Opt.relax_integrality` stands for the value of the `relax_integrality` environment in the `Cutting_Opt` environment. We discuss these kinds of names and their uses at greater length in the next section.

As an example of how this works, Figure 3 shows data for cutting 100" raw rolls, to meet demands of 48, 35, 24, 10 and 8 for finished rolls of widths 20, 45, 50, 55 and 75, respectively. Figure 4 shows the output that occurs when Figure 2's script is run with the model and data as shown in Figures 1 and 3. The best fractional solution cuts 46.25 raw rolls in five different patterns, using actually 48 rolls if the fractional values are rounded up to the next integer. The final solve shows that six of the patterns can be combined to meet demand using only 47 raw rolls.

```
-----
param roll_width := 110 ;

param: WIDTHS: orders :=
      20      48
      45      35
      50      24
      55      10
      75       8 ;
-----
```

Figure 3. Data for the cutting-stock model (`cut.dat`).

```
-----
ampl: include cut.run

CPLEX: All rows and columns eliminated.
CPLEX: optimal solution; objective 52.1
0 iterations

CPLEX: optimal integer solution; objective -0.2
17 simplex iterations
18 branch-and-bound nodes

CPLEX: optimal solution; objective 48.6
3 iterations (2 in phase I)

CPLEX: optimal integer solution; objective -0.2
20 simplex iterations
19 branch-and-bound nodes

CPLEX: optimal solution; objective 47
5 iterations (3 in phase I)

CPLEX: optimal integer solution; objective -0.1
23 simplex iterations
19 branch-and-bound nodes

CPLEX: optimal solution; objective 46.25
```

```

7 iterations (4 in phase I)

CPLEX: optimal integer solution; objective -1e-06
28 simplex iterations
44 branch-and-bound nodes

nbr [*,*] (tr)
: 20 45 50 55 75 :=
1 5 0 0 0 0
2 0 2 0 0 0
3 0 0 2 0 0
4 0 0 0 2 0
5 0 0 0 0 1
6 1 2 0 0 0
7 1 0 0 0 1
8 3 0 1 0 0
;

Cut [*] :=
1 0
2 0
3 8.25
4 5
5 0
6 17.5
7 8
8 7.5
;

CPLEX: optimal integer solution; objective 47
11 simplex iterations
4 branch-and-bound nodes

Cut [*] :=
1 2
2 0
3 10
4 5
5 0
6 18
7 8
8 4
;
-----

```

Figure 4. Output generated from execution of Figure 2's cutting-stock script.

We have developed other examples, listed in Figure 5, that show how AMPL can iterate between models to implement well-known iterative schemes including Dantzig-Wolfe decomposition, Benders decomposition, and Lagrangian relaxation. The Dantzig-Wolfe examples in particular show how more than two named problems can be involved.

Script	Uses	Implements
cut1.run	cut1.mod cut.dat	Gilmore-Gomory column generation procedure for the cutting-stock (roll trim) problem

cut2.run	cut2.mod cut.dat	Same as cut1.run, but using an alternative arrangement wherein problems are defined immediately before their members are declared
cut3.run	cut1.mod cut.dat	Same as cut1.run, but with better formatting of output
multi1.run	multi1.mod multi1.dat	Dantzig-Wolfe decomposition for a multi-commodity transportation problem, using a single subproblem
multi2.run	multi2.mod multi2.dat	Same as multi1.run, but using a separate subproblem for each product; subproblems represented in AMPL by an indexed collection of named problems
multi3.run	multi3.mod multi3.dat	Same as multi2.run, except that the separate subproblems are realized by changing the data to a single AMPL named problem
stoch.run	stoch.mod stoch.dat	Benders decomposition for a stochastic programming variant of a multi-period production problem (see Exercise 4-5)
trnloc1d.run	trnloc1d.mod trnloc1.dat	Benders decomposition for a location-transportation problem
trnloc1p.run	trnloc1p.mod trnloc1.dat	Same as trnloc1d.run, but using a primal rather than dual formulation of the subproblem
trnloc2a.run	trnloc2a.mod trnloc2.dat	Lagrangian relaxation for a location-transportation problem
trnloc2b.run	trnloc2b.mod trnloc2.dat	Same as trnloc2a.run, but model has upper limits on the Ship variables to give better lower bounds on the objective
trnloc2c.run	trnloc2c.mod trnloc2.dat	Same as trnloc2b.run, but model has 0-1 constraints disaggregated to give better LP relaxation

Figure 5. Other examples of the use of named problems in AMPL.

8.2 Defining named problems

At any point during an AMPL session, there is a *current* problem consisting of a list of variables, objectives and constraints. By default, the current problem is named *Initial*, and comprises all variables, objectives and constraints defined so far. You can define other "named" problems, however, and can make them current. When a named problem is made current, as seen in the preceding example, all of the model components in the problem's list are made active, while all other variables, objectives and constraints are made inactive. Thus named problems help to simplify and clarify AMPL scripts that work with two or more related models.

AMPL offers a variety of commands for working with named problems, based in many cases on the forms and commands already used for other purposes. We consider first the several ways to define named problems, and then survey the commands for using and displaying them.

You can define a problem most straightforwardly through a `problem` command that gives the problem's name and its list of components. Thus in Figure 2 we have:

```
problem Cutting_Opt: Cut, Number, Fill;
```

A new problem named `Cutting_Opt` is defined, and is specified to contain all of the `Cut` variables, the objective `Number`, and all of the `Fill` constraints from the model in Figure 1. At the same time, `Cutting_Opt` becomes the current problem. Any fixed `Cut` variables are unfixed, while all other declared variables are fixed (at their current values). The objective `Number` is restored if it had been previously dropped, while all other declared objectives are dropped; and similarly any dropped `Fill` constraints are restored, while all other declared constraints are dropped.

For more complex models, the list of a problem's components typically includes several collections of variables and constraints, as in this example from `stoch.run`:

```
problem Sub: Make, Inv, Sell,  
            Stage2_Profit, Time, Balance2, Balance;
```

By specifying an indexing-expression after the problem name, you can define an indexed collection of problems, such as these in `multi2.run`:

```
problem SubII {p in PROD}: Reduced_Cost[p],  
                    {i in ORIG, j in DEST} Trans[i,j,p],  
                    {i in ORIG} Supply[i,p], {j in DEST} Demand[j,p];
```

For each `p` in the set `PROD`, a problem `SubII[p]` is defined. Its components include the objective `Reduced_Cost[p]`, the variables `Trans[i,j,p]` for each combination of `i in ORIG` and `j in DEST`, and the constraints `Supply[i,p]` and `Demand[j,p]` for each `i in ORIG` and each `j in DEST`, respectively.

In general, a problem declaration's form and interpretation somewhat resemble those of the `display` command (Section 10.3). The declaration begins with the keyword `problem`, a problem-name not previously used for any other model component, an optional indexing-expression (to define an indexed collection of problems), and a colon. Following the colon is the comma-separated list of variables, objectives and constraints to be included in the problem. This list may contain items of any of the following forms, where "component" refers to any variable, objective or constraint:

- A component-name, such as `Cut` or `Fill`, which refers to all components having that name.

- A subscripted component-name, such as `Reduced_Cost[p]`, which refers to that component alone.

- An indexing-expression followed by a subscripted component-name, such as `{i in ORIG} Supply[i,p]`, which refers to one component for each member of the indexing set (as in the `display` command).

To save the trouble of repeating an indexing expression when several components are indexed in the same way, the `problem` statement also allows an indexing-expression followed by a parenthesized list of components. Thus for example all of the following would be equivalent:

```

{i in ORIG} Supply1[i,p], {i in ORIG} Supply2[i,p],
{i in ORIG, j in DEST} Trans[i,j,p],
{i in ORIG, j in DEST} Use[i,j,p]

{i in ORIG} (Supply1[i,p], Supply2[i,p]),
{i in ORIG, j in DEST} (Trans[i,j,p], Use[i,j,p])

{i in ORIG} (Supply1[i,p], Supply2[i,p],
  {j in DEST} Trans[i,j,p], {j in DEST} Use[i,j,p])

{i in ORIG} (Supply1[i,p], Supply2[i,p],
  {j in DEST} (Trans[i,j,p], Use[i,j,p]))

```

As these examples show, the list inside the parentheses may contain any item that is valid in a component-list, even an indexing-expression followed by another parenthesized list. This sort of recursion is also found in AMPL's `function` and `print` commands, but is more general than the list format allowed in `display` commands.

Any variables, objectives or constraints that you declare are automatically added to the current problem. Thus in our earlier example, all of Figure 1's model components are first placed by default into the problem `Initial`; then, when Figure 2's script is run, the components are divided into the problems `Cutting_Opt` and `Pattern_Gen` by use of problem statements. As an alternative, you can declare empty problems and then fill in their members through AMPL declarations. Figure 6 (`cut2.mod`) shows how this would be done for the Figure 1 models. The associated script (`cut2.run`) is similar to the previous one. This approach is sometimes clearer or easier for the simpler applications.

```

-----

  problem Cutting_Opt;
  # =====

  param nPAT integer = 0, default 0;
  param roll_width;

  set PATTERNS := 1..nPAT;
  set WIDTHS;

  param orders {WIDTHS} 0;
  param nbr {WIDTHS,PATTERNS} integer = 0;

  check {j in PATTERNS}:
    sum {i in WIDTHS} i * nbr[i,j] <= roll_width;

  var Cut {PATTERNS} = 0;

  minimize Number: sum {j in PATTERNS} Cut[j];

  subj to Fill {i in WIDTHS}:
    sum {j in PATTERNS} nbr[i,j] * Cut[j] = orders[i];

  problem Pattern_Gen;
  # =====

  param price {WIDTHS};

```



```

var Use {WIDTHS} integer = 0;

minimize Reduced_Cost:
    1 - sum {i in WIDTHS} price[i] * Use[i];

subj to Width_Limit:
    sum {i in WIDTHS} i * Use[i] <= roll_width;
-----

```

Figure 6. An alternative way of specifying the two named problems for Figure 1's the cutting-stock problem (`cut2.mod`). The associated script (`cut2.run`) is similar to the one previously shown.

Any use of `drop/restore` or `fix/unfix` (Section 10.8) also modifies the current problem. The `drop` command has the effect of removing constraints or objectives from the current problem, while the `restore` command has the effect of adding constraints or objectives. Similarly, the `fix` command removes variables from the current problem and the `unfix` command adds variables. As an example, `multil.run` uses the following problem statements,

```

problem MasterI: Artificial, Weight, Excess, Multi, Convex;
problem MasterII: Total_Cost, Weight, Multi, Convex;

```

to define master problems for phases I and II of its decomposition procedure. Instead it could specify

```

problem Master: Artificial, Weight, Excess, Multi, Convex;

```

to define the master problem initially, and then

```

drop Artificial; restore Total_Cost;
fix Excess;

```

when the time comes to convert the master problem to a form appropriate for the second phase. With a similar change to the definition of the subproblem, one loop suffices to implement both phases as seen in `multila.run`.

Alternatively, the `redeclare problem` command can be employed to specify a new definition for a problem. The `drop`, `restore`, and `fix` commands above could be replaced, for instance, by

```

redeclare problem Master: Total_Cost, Weight, Multi, Convex;

```

Like other declarations, however, this cannot be used within a compound statment (`if`, `for` or `repeat`) and so cannot be used in the `multila.run` example. (Other uses of the new `redeclare` keyword are discussed further in ...).

A form of the `reset` command lets you undo any changes made to the definition of a problem. For example,

```

reset problem Cutting_Opt;

```

resets the definition of `Cutting_Opt` to the list of components in the problem statement that most recently defined it.

8.3 Using named problems

We next describe alternatives for changing the identity of the current problem. Any change will in general cause different objectives and constraints to be dropped, and different variables to be fixed, with the result that a different optimization problem is generated for the solver. The values associated with model components are not affected simply by a change in the current problem, however. All previously declared components are accessible regardless of the current problem, and keep the same values unless they are explicitly changed by `let` or `data` statements, or by a `solve` in the case of variable and objective values and related quantities (such as dual values, slacks, and reduced costs).

Any problem statement that refers to just one problem will make that problem current. As an example, at the beginning of the cutting-stock script we want to make first one and then the other named problem current, so that we can adjust certain options in the problems' environments. The problem statements in `cut1.run` (Figure 2),

```
problem Cutting_Opt: Cut, Number, Fill;
    option relax_integrality 1;

problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
    option relax_integrality 0;
```

serve both to define the new problems and to make those problems current. The analogous statements in `cut2.run` are simpler:

```
problem Cutting_Opt;
    option relax_integrality 1;

problem Pattern_Gen;
    option relax_integrality 0;
```

These statements serve only to make the named problems current, because the problems have already been defined by problem statements in `cut2.mod` (Figure 6).

A problem statement may also refer to an indexed collection of problems, as in the `stoch.run` example cited previously:

```
problem SubII {p in PROD}: Reduced_Cost[p],
    {i in ORIG, j in DEST} Trans[i,j,p],
    {i in ORIG} Supply[i,p], {j in DEST} Demand[j,p];
```

This form defines potentially many problems. Subsequent declarations go into all of these problems, as long as the unsubscripted problem name is current. Subsequent problem statements can make members of a collection current one at a time, as in a loop having the form

```
for {p in PROD} {
    problem SubII[p];
    ...
}
```

or in a statement such as `problem SubII["coils"]` that refers to a particular member.

As seen in previous examples, the `solve` statement can also include a problem-name, in which case the named problem is made current and then sent to the solver. The effect of a

statement such as `solve Pattern_Gen` is thus exactly the same as the effect of `problem Pattern_Gen` followed by `solve`.

8.4 Displaying named problems

The command consisting of `problem` alone tells you which problem is current:

```
ampl: model cut1.mod;  
ampl: data cut.dat;  
  
ampl: problem;  
problem Initial;  
  
ampl: problem Cutting_Opt: Cut, Number, Fill;  
ampl: problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;  
  
ampl: problem;  
problem Pattern_Gen;
```

The current problem is always `Initial` until other named problems have been defined.

Use the `show` command to get a list of the named problems that you have defined:

```
ampl: show problems;  
problems:   Cutting_Opt   Pattern_Gen
```

The predefined set `_PROBS` also contains these names:

```
ampl: display _PROBS;  
set _PROBS := Cutting_Opt Pattern_Gen;
```

Also use `show` to see the variables, objectives and constraints that make up a particular problem,

```
ampl: show Cutting_Opt, Pattern_Gen;  
problem Cutting_Opt: Fill, Number, Cut;  
problem Pattern_Gen: Width_Limit, Reduced_Cost, Use;
```

or indexed collection of problems.

Use `expand` to see the explicit objectives and constraints of the current problem, after all data values have been substituted:

```
ampl: expand Pattern_Gen;  
  
minimize Reduced_Cost:  
  -0.166667*Use[20] - 0.416667*Use[45] - 0.5*Use[50] -  
    0.5*Use[55] - 0.833333*Use[75] + 1;  
  
s.t. Width_Limit:  
  20*Use[20] + 45*Use[45] + 50*Use[50] + 55*Use[55] +  
  75*Use[75] <= 110;
```

See the Examining Models and Data section of this supplement for further discussion of `expand`. [[At present, `expand` can be applied only to the current problem.]]

8.5 Defining and using named environments

In the same way that there is a current problem at any point in an AMPL session, there is also a current "environment". Whereas a problem is a list of unfixed variables and undropped objectives and constraints, an environment records the values of all AMPL options. By naming different environments, a script can clearly and easily switch between different collections of option settings.

Named environments are automatically defined and changed when you define and change named problems. We begin by describing this mode of operation, which is sufficient for many purposes. We then describe several alternatives that afford a greater degree of control over environments, by permitting them to be managed independently of problems.

When you use the problem statement as shown above, the current environment always has the same name as the current problem. Specifically, at the start of an AMPL session the current environment is named `Initial`, and each subsequent problem statement that defines a new named problem also defines a new environment having the same name. An environment initially inherits all the option settings that existed when it was created, but it may be changed to new option settings. Any subsequent problem or solve statement that changes the current problem also switches to the correspondingly named environment.

As an example, our script for the cutting stock problem (Figure 2) sets up the model and data and then proceeds as follows:

```
option solver cplex;
option solution_round 6;

problem Cutting_Opt: Cut, Number, Fill;
option relax_integrality 1;

problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
option relax_integrality 0;
```

Options `solver` and `solution_round` are reset (by the first two option statements) before any of the problem statements; hence their new settings are inherited by subsequently defined environments and are the same throughout the rest of the script. Next a problem statement defines a new problem and a new environment named `Cutting_Opt`, and makes them current. The ensuing option statement changes `relax_integrality` to 1. Thereafter, when `Cutting_Opt` is the current problem (and environment) in the script, `relax_integrality` will have the value 1. Finally, another problem and option statement do much the same for problem (and environment) `Pattern_Gen`, except that `relax_integrality` is set back to 0 in that environment.

The result of these initial statements is to guarantee a proper setup for each of the subsequent solve statements in the repeat loop. The result of `solve Cutting_Opt` is to set the current environment to `Cutting_Opt`, thereby setting `relax_integrality` to 1 and causing the linear relaxation of the cutting optimization problem to be solved. Similarly the result of `solve Pattern_Gen` is to cause the pattern generation problem to be solved as an integer program. We could instead have used option statements within the loop to switch the setting of `relax_integrality`, but with this approach we have kept the loop -- the key part of

the script -- as simple as possible. The advantages of this approach can be much more pronounced in scripts for more complex algorithms.

9 Presolve Tolerances

Because computers work in finite-precision arithmetic, AMPL must employ a variety of tolerances to cope with small inaccuracies. For example, rounding to finite precision often leads solvers to return numbers that differ from zero by an insignificant amount:

```
ampl: model mod/diet.mod;
ampl: data dat/diet.dat;

ampl: solve;
MINOS 5.4: optimal solution found.
6 iterations, objective 88.2

ampl: option omit_zero_rows 1;
ampl: display Buy;

Buy [*] :=
  MCH  46.6667
  MTL  -1.07823e-16
  SPG  -1.32893e-16
;
```

To help clean up displays in these cases, AMPL's `display_eps` option (Section 10.5 of the AMPL book) specifies the smallest magnitude that is treated as being different from zero in the output of the `display` command:

```
ampl: option display_eps .000001;
ampl: display Buy;

Buy [*] :=
  MCH  46.6667
;
```

See also the discussion in Section 10.5 concerning the care that must be taken in rounding solution values.

Most of AMPL's tolerances relate to the simplifications that are applied by the presolve phase (Section 10.2), after you type `solve` but before the solver is invoked. Presolve makes a series of passes through the linear part of an optimization problem, applying a number of tests that can tighten various bounds without changing the optimum. (Conventions for viewing these bounds are explained in Section A.13.3.) The tightened bounds may imply that some variables can be fixed and some constraints can be dropped, while constraints that have only one unfixed variable may be folded into the variable's

bounds. As a consequence of these changes, presolving can significantly reduce the size of the problem sent to the solver.

Because the decisions made by the presolve phase rely on numerical computations, small differences between numbers can have a large effect on the results. The following tolerance options have been introduced to deal with a variety of situations in which this can occur. Fortunately, the default values of these options produce acceptable results in a large majority of cases. Warnings are displayed if a small change in tolerance settings could affect the optimal values in any significant way.

The `presolve` option specifies the maximum number of passes to be made by the presolve phase. Thus `option presolve 0` turns off all presolving, and `option presolve 1` applies only the more elementary simplifications. The default value is 10.

9.1 Determining that no feasible solution exists

If presolve determines that any variable's lower bound is greater than its upper bound, then there can be no solution satisfying all the bounds and other constraints, and an error message is printed accordingly. For example, here's what would happen if you changed `market["bands"]` to 500 when you meant 5000:

```
ampl: model steel3.mod;
ampl: data steel3.dat;

ampl: let market["bands"] := 500;
ampl: solve;

inconsistent bounds for var Make[bands]:
    lower bound = 1000  upper bound = 500;
    difference = 500
Ignoring solve command because presolve finds no feasible
solution possible.
```

This is a simple case, because the upper bound on variable `Make["bands"]` has clearly been reduced below the lower bound. Presolve's more sophisticated tests can also find infeasibilities that are not due to any one variable. As an example, consider the constraint in this model:

```
subject to Time: sum{p in PROD} 1/rate[p]*Make[p] <= avail;
```

If you reduce the value of `avail` to 13 hours, presolve deduces that this constraint can't possibly be satisfied:

```
ampl: let market["bands"] := 5000;
ampl: let avail := 13;

ampl: solve;

presolve: constraint Time cannot hold:
    body <= 13 cannot be = 13.2589; difference = -0.258929
Ignoring solve command because presolve finds no feasible
solution possible.
```


The "body" of constraint Time is $\sum \{p \text{ in PROD} \} 1/\text{rate}[p] * \text{Make}[p]$, the part that contains the variables (see Section 10.7). Presolve reports that, given the bounds on the variables that it has found, this expression must have a value of at least 13.2589 (to six digits); on the other hand, the value of avail that we gave requires that this expression have a value of at most 13.

Presolve reports the difference between its two bounds for constraint Time as (again to six digits) -0.258929. Thus in this case we can guess that 13.258929 is, approximately, the smallest value of avail that allows for a feasible solution. Indeed, we get:

```
ampl: let avail := 13.258929;
ampl: solve;

MINOS 5.4: optimal solution found.
0 iterations, objective 61750.00214
```

If we make avail just slightly lower, however, we again get the infeasibility message:

```
ampl: let avail := 13.258928;
ampl: solve;

presolve: constraint Time cannot hold:
      body <= 13.2589 cannot be = 13.2589;
      difference = -5.71429e-07
Setting $presolve_eps = 5.714285720159751e-07 might help.

Ignoring solve command because presolve finds no feasible
solution possible.
```

Although the lower bound is here the same as the upper bound to six digits, it is greater than the upper bound in full precision, as the negative value of the difference indicates.

Typing solve a second time in this situation tells AMPL to override presolve and send the seemingly inconsistent deduced bounds to the solver:

```
ampl: solve;

MINOS 5.4: optimal solution found.
0 iterations, objective 61749.99714

ampl: option display_precision 10;
ampl: display commit,Make;

:      commit      Make      :=
bands   1000      999.9998857
coils    500       500
plate    750       750
;
```

The MINOS solver declares that it has found an optimal solution, though with `Make["bands"]` being slightly less than its lower bound `commit["bands"]`! Here MINOS is applying an internal tolerance that allows small infeasibilities to be ignored; the AMPL/MINOS documentation explains how this tolerance works and how it can be changed. Each solver applies feasibility tolerances in its own way, so it's not surprising that a different solver gives different results:

```
ampl: option solver cplex;
ampl: solve;
```

```
CPLEX: All rows and columns eliminated.  
CPLEX: infeasible problem  
0 iterations
```

Here CPLEX has applied its own presolve and has detected the same infeasibility that AMPL did.

AMPL's presolve has its own tolerances that you can set. They are mainly useful for problems that, due to some property of the formulation, cause computed lower bounds to be equal to their corresponding upper bounds. Due to imprecision in the computations, some lower bounds in this case will come out slightly higher than their upper bounds, resulting in a "no feasible solution possible" message such as those above. You can tell that you are in this situation because the reported "difference" between the bounds is very small compared to the bounds themselves.

9.2 Rounding bounds on integer variables

When AMPL's presolve phase determines that a variable declared integer has a fractional lower (or upper) bound, it rounds the bound up to the next highest integer (or down to the next lowest integer). Due to inaccuracies of finite-precision computation, however, a bound may be calculated to have a value that is just slightly different from an integer. An upper bound that should be 7, for example, might be calculated as 6.99999999998, in which case you would not want the bound to be rounded down to 6!

10 Reporting and Display

A variety of miscellaneous new options provide additional information, or regulate the information that AMPL displays.

10.1 Solver return messages

AMPL's solver interfaces are set up to display a brief summary of their status when they are done:

```
ampl: model diet.mod;  
ampl: data diet2.dat;  
ampl: solve;  
  
MINOS 5.5: infeasible problem.  
9 iterations
```

If you are running a script that executes `solve` frequently, these solver return messages can add up to a lot of output. You can turn them off by changing the new AMPL option `solver_msg` from its default value of 1 to 0.

A new built-in symbolic parameter, `solve_message`, always contains the most recent solver return message -- even when display of the message has been turned off. You can display this parameter to verify its value:

```
ampl: display solve_message;  
  
solve_message = 'MINOS 5.5: infeasible problem.\n9 iterations'
```

Because `solve_message` is a symbolic parameter, its value has the form of a character string. It is most useful in AMPL scripts, where you can use AMPL's character-string functions to test the message for indications of optimality and other outcomes.

As an example, the following script for the diet problem builds a table of objective and dual values at steadily decreasing settings of parameter `n_max["NA"]`. The `match` function is used to test whether "infeasible" appears in the solver return message. When it does, a warning is displayed, the `for` loop is aborted, and the completed table is shown:

```
model diet.mod;  
data diet2a.dat;
```

```

set NA;
param NA_obj {NA};
param NA_dual {NA};

let NA := n_max["NA"] .. n_max["NA"] - 10000 by -500;

option solver_msg 0;

for {a in NA} {
    let n_max["NA"] := a;
    solve;

    if match (solve_message, "infeasible") 0 then {
        printf "--- infeasible at %d ---\n\n", a;
        break;
    }

    let NA_obj[a] := total_cost;
    let NA_dual[a] := diet["NA"].dual;
}

display NA_obj, NA_dual;

```

Here are the results of running the script, from file `diet.run`:

```

ampl: commands diet.run;

--- infeasible at 48000 ---

:      NA_obj      NA_dual      :=
48500  122.663    -0.00306905
49000  121.128    -0.00306905
49500  119.594    -0.00306905
50000  118.059    -0.00306905
;

```

Since return messages vary somewhat from one solver to the next, the appropriate test in such a script is somewhat solver-dependent. More sophisticated and solver-independent ways of checking the return condition are planned for future versions of AMPL, but this simple approach should be adequate in many cases.

10.2 Time and memory listings

By changing the `show_stats` option from its default of 0 to any nonzero value, you request summary statistics on the size of the optimization problem that AMPL generates:

```

ampl: option show_stats 1;

ampl: model prod.mod;
ampl: data prod13.dat;
ampl: solve;

```

```

Presolve eliminates 82 constraints and 14 variables.
Adjusted problem:
871 variables, all linear
647 constraints, all linear; 2736 nonzeros

```

```
1 linear objective; 702 nonzeros.
```

```
MINOS 5.4: optimal solution found.  
532 iterations, objective 2359313.555  
ampl:
```

Several other options can give you more detailed information about AMPL's resource requirements.

By changing the `times` option from its default of 0 to any nonzero value, you request a summary of the AMPL translator's time and memory requirements:

```
ampl: option times 1;  
  
ampl: model prod.mod;  
ampl: data prod13.dat;  
  
#  
#phase          seconds      incremental      total  
#phase          seconds      memory         memory  
#parse          0.116667     213000        213000  
#data read      0           20480         233480  
#parse          0           0             233480  
  
ampl: solve;  
  
#data read      0.05         36864         270344  
#compile        0.0333333    0             270344  
#genmod         0.35         159744        430088  
#collect        0.05         126976        557064  
#presolve       0.266667     118784        675848  
#output         0.316667     8192          684040  
  
MINOS 5.4: optimal solution found.  
532 iterations, objective 2359313.555  
  
#Total          1.18333  
  
ampl:
```

The major phases of translation are listed at the left. The "seconds" column gives each phase's execution time in seconds. The "incremental memory" and "total memory" columns give the additional memory that each phase allocated, and the cumulative memory allocated by each phase and all previous ones. This information is mainly useful for benchmarking AMPL's performance.

By similarly changing the `gentimes` option from its default of 0 to any nonzero value, you can get a more detailed summary of the resources that AMPL's `genmod` phase consumes in generating a model instance:

```
ampl: option gentimes 1;  
  
ampl: model prod.mod;  
ampl: data prod13.dat;  
ampl: solve;  
  
##genmod times:  
##seq      seconds      cum. sec.      mem. inc.      name  
## 3        0           0             0      prd  
## 4        0           0             0      pt
```

##	5	0	0	0	pc
##	6	0	0	0	first
##	7	0	0	0	last
##	8	0	0	0	time
##	9	0	0	0	cs
##	10	0	0	0	sl
##	11	0.0166667	0.0166667	0	rtr
##	12	0	0.0166667	0	otr
##	13	0	0.0166667	0	iw
##	14	0	0.0166667	0	dpp
##	15	0	0.0166667	0	ol
##	16	0	0.0166667	0	cmin
##	17	0	0.0166667	0	cmax
##	18	0	0.0166667	0	hc
##	19	0	0.0166667	0	lc
##	20	0	0.0166667	0	dem
##	21	0	0.0166667	0	pro
##	22	0	0.0166667	0	rir
##	23	0	0.0166667	0	pir
##	24	0	0.0166667	0	life
##	25	0	0.0166667	0	cri
##	26	0	0.0166667	0	crs
##	27	0.0166667	0.0333333	0	iinv
##	28	0.05	0.0833333	4096	iil
##	29	0.0333333	0.116667	0	minv
##	30	0	0.116667	0	Crews
##	32	0	0.116667	0	Hire
##	34	0	0.116667	0	Layoff
##	36	0	0.116667	8192	Rprd
##	38	0	0.116667	4096	Oprd
##	40	0.0166667	0.133333	16384	Inv
##	42	0	0.133333	0	Short
##	44	0.0666667	0.2	65536	cost
##	46	0.0166667	0.216667	0	rlim
##	48	0	0.216667	0	olim
##	50	0	0.216667	0	empl0
##	52	0	0.216667	0	empl
##	54	0	0.216667	0	emplbnd
##	56	0.0166667	0.233333	0	dreq1
##	58	0.05	0.283333	28672	dreq
##	60	0.0333333	0.316667	0	ireq
##	62	0	0.316667	0	izero
##	64	0	0.316667	28672	ilim1
##	66	0.0333333	0.35	4096	ilim

MINOS 5.4: optimal solution found.
532 iterations, objective 2359313.555
ampl:

The right-hand column gives the names of model components processed by the genmod phase. The "seconds" column gives AMPL's processing times for the components, and the "cum. sec." column keeps a running total. The "mem. inc." column indicates the additional memory, if any, that AMPL allocated while processing individual model components.

When AMPL appears to "hang" or takes much more time than expected, the gentimes display can help you to associate the difficulty with a particular model component. Typically, some parameter, variable or constraint has been indexed over a set far larger

than what was intended or anticipated, with the result that an excessive amount of time and memory is required.

If you change the model or data and solve again, additional lines of the `gentimes` display will appear:

```
ampl: let {p in prd, t in first..last+1}
ampl?   dem[p,t] := dem[p,t] * 1.05;
ampl: solve;

##genmod times:
##seq      seconds      cum. sec.      mem. inc.      name
## 20          0          0          0          dem
## 28    0.0666667    0.0666667          0          iil
## 29    0.0166667    0.0833333          0          minv
## 56          0          0.0833333          0          dreql
## 58    0.0666667          0.15        4096          dreq
## 60          0.05          0.2          0          ireq

MINOS 5.4: optimal solution found.
366 iterations, objective 2483123.471
ampl:
```

Here we see that a change to the parameter `dem` necessitated changes to only five other model components: the parameters `iil` and `minv` that are defined in terms of `dem`, and the constraints `dreql`, `dreq` and `ireq` that use these parameters. AMPL keeps a representation of your model in memory throughout a session, and regenerates only the affected components when you solve after a change.

The timings shown above apply only to the AMPL translator, not to the solver. Many solvers have a directive for requesting a breakdown of the solve time, as in this example:

```
ampl: option solver cplex;
ampl: option cplex_options 'timing 1';

ampl: model prod.mod;
ampl: data prod13.dat;

ampl: solve;
CPLEX: timing 1

Times (seconds):
Input = 0.283333
Solve = 4.95
Output = 0.0166667

CPLEX: optimal solution; objective 2359313.555
473 iterations (230 in phase I)
ampl:
```

Information on requesting and interpreting these timings is provided in the documentation of AMPL's links to individual solvers.

10.3 Output logs

The `log_file` option instructs AMPL to save subsequent commands and responses to a file. The option's value is a string that is interpreted as a filename; thus you might say

```
ampl: option log_file 'MULTI.TMP';
```

when running under MS-DOS, or

```
ampl: option log_file '/tmp/multi';
```

under Unix. The log file receives all AMPL statements and the output that they produce, with a few exceptions described below. Setting `log_file` to the empty string,

```
ampl: option log_file '';
```

turns off writing to the file; the empty string is the default value for this option.

When AMPL reads from an input file by means of a `model`, `data` or `include` command, the statements from that file are not copied to the log file -- although any output resulting from those statements does get logged. In typical situations, this arrangement prevents your model and data files from being echoed to your log file. To request that AMPL echo the contents of input files, change the option `log_model` (for input in model mode) or `log_data` (for input in data mode) from its default value of 0 to some nonzero value.

When you invoke a solver, AMPL logs at least a few lines summarizing the objective value, solution status and work required. Through certain solver-specific directives, you can typically request additional solver output such as logs of iterations or branch-and-bound nodes. Many solvers automatically send all of their output to AMPL's log file, but this compatibility is not universal. If a solver's output does not appear in your log file, you should consult the supplementary documentation for that solver's AMPL link; possibly that solver accepts nonstandard directives for diverting its output to files.

10.4 Limits on messages

By specifying option `eexit n`, where n is some integer, you determine how AMPL handles error messages. If n is not zero, any AMPL statement is terminated after it has produced `abs(n)` error messages; a negative value causes only the one statement to be terminated, while a positive value results in termination of the entire AMPL session. The effect of this option is most often seen in the use of model statements,

```
ampl: option eexit -3;  
ampl: model multempl.mod;
```

```
multempl.mod, line 2 (offset 22):  
    syntax error  
context:    set <<< DEST;    # destinations
```

```
multempl.mod, line 6 (offset 149):  
    DEST is not defined  
context:    param demand {DEST, <<< PROD} = 0;
```

```
multempl.mod, line 9 (offset 272):  
    DEST is not defined  
context:    sum {i in ORIG} supply[i,p]  
            = sum {j in DEST} <<< demand[j,p];
```

```
Bailing out after 3 warnings.  
ampl:
```


and data statements:

```
stretto% ampl
ampl: option eexit 3;

ampl: model multemp2.mod;
ampl: data multemp2.dat;

multemp2.dat, line 9 (offset 262):
    expected :=
context:          plate      200      300      300  >>> ; <<<

multemp2.dat, line 11 (offset 271):
    demad is not a param (or var or constraint)
context:  param  >>> demad <<< (tr):

multemp2.dat, line 17 (offset 508):
    1 item(s) missing in last line of table,
    which starts with "default"
context:  param limit default 625  >>> ; <<<

Bailing out after 3 warnings.
stretto%
```

The default value for `eexit` is -10. Setting it to 0 causes all error messages to be displayed.

The `eexit` setting also applies to infeasibility warnings produced by AMPL's presolve phase after you type `solve`. The number of these warnings is simultaneously limited by the value of option `presolve_warnings`, which is typically set to a smaller value:

```
ampl: option presolve_warnings 3;

ampl: model multemp3.mod;
ampl: data multemp3.dat;

ampl: solve;
presolve: constraint Demand[LAF,plate] cannot hold:
    body >= 250 cannot be <= 75; difference = 175
presolve: constraint Demand[LAF,coils] cannot hold:
    body >= 500 cannot be <= 75; difference = 425
presolve: constraint Demand[LAF,bands] cannot hold:
    body >= 250 cannot be <= 75; difference = 175

52 presolve messages suppressed.
Infeasible solution determined by presolve.

ampl:
```

The default value for `presolve_warnings` is 5.

An AMPL data statement may specify values that correspond to illegal combinations of indices, due to any of a number of mistakes such as incorrect index sets in the model, indices in the wrong order, misuse of `(tr)`, and typing errors. Similar errors may be caused by `let` statements that change the membership of index sets. In the following example, there are three set members typed incorrectly in the data table for parameter `supply`:

```
set ORIG := GARY CLEV PITT ;
set PROD := bands coils plate ;
```

```

param supply (tr):  GARY    Clev    PIT :=
                    bands    400    700    800
                    coils    800    1600   1800
                    plates   200    300    300 ;

```

As a result, the table specifies invalid combinations of indices for seven data values. AMPL catches these errors after `solve` is typed. The number of invalid combinations actually displayed is determined from the value of the option `bad_subscripts`:

```

ampl: option bad_subscripts 2;

ampl: model multemp4.mod;
ampl: data multemp4.dat;
ampl: solve;

Error executing "solve" command:
error processing param supply:
    7 invalid subscripts discarded:
    supply[Clev,bands]
    supply[PIT,bands]
    and 5 more.

```

The default value for `bad_subscripts` is 3.

10.5 Prompts

AMPL has three pairs of prompts whose appearance you can change through option settings. The default settings are:

```

option cmdprompt1 '%s ampl: ';
option cmdprompt2 '%s ampl? ';
option dataprompt1 'ampl data: ';
option dataprompt2 'ampl data? ';
option prompt1 'ampl: ';
option prompt2 'ampl? ';

```

The values of `prompt1` and `prompt2` are what you usually see: `prompt1` appears when a new statement is expected, and `prompt2` when you begin a new line in the middle of a statement.

In data mode, the values of `dataprompt1` and `dataprompt2` are used instead. When a new line is begun in the middle of an `if`, `for` or `repeat` statement, the values of `cmdprompt1` and `cmdprompt2` are used, with `%s` replaced by the appropriate command name; for example:

```

ampl: for {t in time} {

    for{...} { ? ampl: if t <= 6
    for{...} { ? ampl?   then let cmin[t] := 3;
    if ... then {...} ? ampl: else let cmin[t] := 4;

    for{...} { ? ampl: };
ampl:

```

Since data tables and the `if`, `for` and `repeat` statements are most conveniently read from files rather than typed at the command line, the alternative prompts are rare in normal use.

11 Statuses

In addition to the solution and related numerical values, it can be useful to have certain symbolic information about the results of the most recent `solve`. For example,

In a script of AMPL commands, you may want to test whether the most recent `solve` encountered an unbounded or infeasible problem.

After you have solved a linear program by the simplex method, you may want to use the optimal basis partition to provide a good start for solving a related problem.

A new feature of the AMPL-solver interface permits solvers to return these and related kinds of *status* information so that you can examine and use it in a standard way.

We begin by describing a convenient and precise new way of testing the overall result of a solver run, and the result of the most recent solver run for a specified objective or named problem. Then we describe the *solver status* for variables and constraints and its use to specify an advanced start. We conclude by describing a complementary *AMPL status* that characterizes model components that have not been sent to the solver for various reasons. For most purposes, a variable or constraint's status of interest can be denoted by adding the suffix `.status` to its name.

11.1 Solve results

A solver finishes its work because it has identified an optimal solution or has encountered some other terminating condition. In addition to the values of the variables, the solver may set two built-in AMPL parameters and an AMPL option that provide information about the outcome of the optimization process:

```
ampl: model diet.mod;
ampl: data diet2.dat;

ampl: display solve_result_num, solve_result;
solve_result_num = -1
solve_result = '?'

ampl: solve;
MINOS 5.5: infeasible problem.
9 iterations

ampl: display solve_result_num, solve_result;
```

```

solve_result_num = 200
solve_result = infeasible

ampl: option solve_result_table;
option solve_result_table '\
0      solved\
100    solved?\
200    infeasible\
300    unbounded\
400    limit\
500    failure\
';

```

At the beginning of an AMPL session, `solve_result_num` is -1 and `solve_result` is '?'. Any `solve` command resets these parameters, however, so that they describe the solver's status at the end of its run, `solve_result_num` by a number and `solve_result` by a character string. The `solve_result_table` lists the possible combinations, which may be interpreted as follows:

number	string	interpretation
0 - 99	solved	optimal solution found
100 - 199	solved?	optimal solution indicated, but error likely
200 - 299	infeasible	constraints cannot be satisfied
300 - 399	unbounded	objective can be improved without limit
400 - 499	limit	stopped by a limit that you set (such as on iterations)
500 - 599	failure	stopped by an error condition in the solver routines

Normally this status information is used in scripts, where it can be tested to distinguish among cases that must be handled in different ways. As an example, the following script for the diet problem progressively decrements the sodium limit, `n_max["NA"]`, until it finds a value that makes the problem infeasible:

```

model diet.mod;
data diet2a.dat;

repeat {
  solve;
  if solve_result = "infeasible" then break;
  let n_max[ "NA" ] := n_max[ "NA" ] - 1000;
};

```

The `if` condition `solve_result = "infeasible"` could equivalently be written `200 <= solve_result_num < 300`. Normally you will want to avoid the latter alternative, since it only makes the script more cryptic. It can occasionally be useful, however, in making fine distinctions among different solver termination conditions. For example, here are some of the values that the CPLEX solver sets for different optimality conditions:

<code>solve_result_num</code>	message at termination
0	optimal solution
1	primal has unbounded optimal face
2	optimal integer solution
3	optimal integer solution within mipgap or absmipgap

The value of `solve_result` is "solved" in all of these cases, but you can test `solve_result_num` if you need to distinguish among them. The interpretations of `solve_result_num` are entirely solver-specific; you'll have to look at a particular solver's instructions to see which values it returns and what they mean.

The brief result message that the solver returns to AMPL is also accessible via a built-in symbolic parameter, `solve_message`:

```
ampl: display solve_message;
solve_message = 'MINOS 5.5: infeasible problem.\
9 iterations'
```

You can use AMPL's string functions to test the contents of `solve_message`, but for most situations a test of `solve_result` will be simpler and less solver-dependent.

Solve results can be returned as described above only if AMPL's invocation of the solver has been successful. Invocation can fail because the operating system is unable to find or execute the specified solver, or because some low-level error prevents the solver from attempting or completing the optimization. Typical causes include a misspelled solver name, improper installation of the solver, missing or corrupt input files, insufficient memory or disk space, and termination of the solver process by an execution fault ("core dump") or a "break" from the keyboard.

The built-in parameter `solve_exitcode` records the success or failure of the most recent solver invocation. Initially -1, it is reset to 0 whenever there has been a successful invocation, and to some nonzero value otherwise:

```
ampl: reset;
ampl: display solve_exitcode;
solve_exitcode = -1

ampl: model diet.mod;
ampl: data diet2.dat;
ampl: option solver xplex;
ampl: solve;

Cannot invoke xplex: No such file or directory
exit code 4
<BREAK>

ampl: display solve_exitcode;
solve_exitcode = 1024

ampl: display solve_result, solve_result_num;
solve_result = '?'
solve_result_num = -1
```

Here the failed invocation, due to the misspelled solver name `xplex`, is reflected by a positive `solve_exitcode` value. The solve status parameters `solve_result` and `solve_result_num` are also reset to their initial values '?' and -1.

If `solve_exitcode` exceeds the value in option `solve_exitcode_max`, then AMPL aborts any currently executing compound commands (`include`, `commands`, `repeat`, `for`, `if`). The default value of `solve_exitcode_max` is 0, so that AMPL normally aborts compound commands whenever a solver's invocation fails. A script that

sets `solve_exitcode_max` to a higher value may test the value of `solve_exitcode`, but in general its interpretation is not consistent across operating systems or solvers.

11.2 Solver statuses of objectives and problems

Sometimes it is convenient to be able to refer to the solve result obtained when a particular objective was most recently optimized, or when a particular named problem was most recently solved. For this purpose, AMPL associates with each built-in solve result parameter a "status" suffix:

built-in parameter	suffix
<code>solve_result</code>	<code>.result</code>
<code>solve_result_num</code>	<code>.result_num</code>
<code>solve_message</code>	<code>.message</code>
<code>solve_exitcode</code>	<code>.exitcode</code>

Appended to an objective or problem name, this suffix indicates the value of the corresponding built-in parameter at the most recent solve in which the objective or problem was current. Thus our sample script `trnloc1p.run` for Benders decomposition suffixes `SubPoint` with `.result` to refer to the most recent subproblem solution:

```
if SubPoint.result = "infeasible" then { ...
```

As another example, consider the small McDonald's diet model. Its objective functions are defined as:

```
minimize Total_Cost: sum {j in FOOD} cost[j] * Buy[j];
minimize Nutr_Amt {i in NUTR}: sum {j in FOOD} amt[i,j]*Buy[j];
```

After minimizing three of these objectives, we can view the solve status values for all of them:

```
ampl: model diet1.mod;
ampl: data diet1.dat;
ampl: solve;

CPLEX: optimal integer solution; objective 15.05
74 simplex iterations
69 branch-and-bound nodes
Objective = Total_Cost

ampl: objective Nutr_Amt['Cal'];
ampl: solve;

CPLEX: optimal integer solution; objective 2500
11 simplex iterations
3 branch-and-bound nodes

ampl: objective Nutr_Amt['VitC'];
ampl: solve;

CPLEX: optimal integer solution; objective 100
34 simplex iterations
```

26 branch-and-bound nodes

```
ampl: display Total_Cost.result, Nutr_Amt.result;
```

```
Total_Cost.result = solved
```

```
Nutr_Amt.result [*] :=
```

```
    Cal    solved
    Carbo   '?'
Protein    '?'
    VitA    '?'
    VitC    solved
    Calc    '?'
    Iron    '?'
;
```

For the objectives that have not yet been used, the `.result` suffix is unchanged (at its initial value of `'?'` in this case).

11.3 Solver statuses of variables

In addition to providing for return of the overall status of the optimization process as described above, AMPL lets a solver return an individual status for each variable. This feature is intended mainly for reporting the basis status of variables after a linear program is solved either by the simplex method, or by an interior-point (barrier) method followed by a "crossover" routine. The basis status is also relevant to solutions returned by certain nonlinear solvers, notably MINOS, that employ an extension of the concept of a basic solution.

In addition to the variables declared by `var` statements in an AMPL model, solvers also define "slack" or "artificial" variables that are associated with constraints. Solver statuses for these latter variables are defined in a similar way, as explained in the next section. Both variables and constraints also have an "AMPL status" that distinguishes those in the current problem from those that have been removed from the problem by presolve or by commands such as `drop`. The interpretation of AMPL statuses and their relationship to solver statuses are considered in a subsequent section.

The major use of solver status values from an optimal basic solution is to provide a good starting point for the next optimization run. The new option `send_statuses`, when left at its default value of 1, instructs AMPL to include statuses with the information about variables sent to the solver at each `solve`. You can see the effect of this feature in almost any sensitivity analysis that re-solves after making some small change to the problem. As an example, consider what happens when the multi-period production example from the AMPL book is solved repeatedly after increases of 5% in the availability of labor. With the `send_statuses` option set to 0, the solver reports about 30 iterations of the simplex method each time it is run:

```
ampl: model steelT3.mod;
ampl: data steelT3.dat;
ampl: option send_statuses 0;

ampl: solve;
CPLEX: optimal solution; objective 514521.7143
30 iterations (1 in phase I)
```

```

ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX: optimal solution; objective 537104
31 iterations (1 in phase I)

```

```

ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX: optimal solution; objective 560800.4
31 iterations (1 in phase I)

```

```

ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX: optimal solution; objective 585116.22
30 iterations (1 in phase I)

```

With `send_statuses` left at its default value of 1, however, only the first solve takes 30 iterations. Subsequent runs take a few iterations at most:

```

ampl: model steelT3.mod;
ampl: data steelT3.dat;

ampl: solve;
CPLEX: optimal solution; objective 514521.7143
30 iterations (1 in phase I)

ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX: optimal solution; objective 537104
3 iterations (2 in phase I)

ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX: optimal solution; objective 560800.4
0 iterations

ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX: optimal solution; objective 585116.22
2 iterations (1 in phase I)

```

Each solve after the first automatically uses the variables' basis statuses from the previous solve to construct a starting point that turns out to be only a few iterations from the optimum. In the case of the third solve, the previous basis remains optimal; the solver thus confirms optimality immediately and reports taking 0 iterations.

The following discussion explains how you can view, interpret, and change variables' status values in the AMPL environment. You don't need to know any of this to use optimal bases as starting points as shown above, but these features can be useful in certain advanced circumstances.

AMPL refers to a variable's solver status by appending `.sstatus` to its name. Thus you can exhibit the statuses of variables by use of the `display` command. At the beginning of a session (or after a `reset`), when no problem has yet been solved, all variables have the status `none`:

```

ampl: model diet.mod;

```



```

ampl: data diet2a.dat;

ampl: display Buy.sstatus;

Buy.sstatus [*] :=
BEEF  none
CHK   none
FISH  none
HAM   none
MCH   none
MTL   none
SPG   none
TUR   none
;

```

After an invocation of a simplex method solver, the same `display` lists the statuses of the variables at the optimal basic solution:

```

ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032

ampl: display Buy.sstatus;

Buy.sstatus [*] :=
BEEF  bas
CHK   low
FISH  low
HAM   upp
MCH   upp
MTL   upp
SPG   bas
TUR   low
;

```

Two of the variables -- `Buy['BEEF']` and `Buy['SPG']` -- have status `bas`, which means they are in the optimal basis. Three have status `low` and three `upp`, indicating that they are nonbasic at lower and upper bound, respectively. A table of the recognized solver status values is stored in the AMPL option `sstatus_table`:

```

ampl: option sstatus_table;

option sstatus_table '\
0      none    no status assigned\
1      bas     basic\
2      sup     superbasic\
3      low     nonbasic <= (normally =) lower bound\
4      upp     nonbasic = (normally =) upper bound\
5      equ     nonbasic at equal lower and upper bounds\
6      btw     nonbasic between bounds\
';

```

Numbers and short strings representing status values are given in the first two columns. (The numbers are mainly for communication between AMPL and solvers, though you can access them by using the suffix `.sstatus_num` in place of `.sstatus`.) The entries in the third column are comments. For nonbasic variables as defined in many textbook simplex methods, only the `low` status is applicable; other nonbasic statuses are required for the more general bounded-variable simplex methods in large-scale

implementations. The `sup` status is used by solvers like MINOS to accommodate nonlinear problems. This is AMPL's standard `sstatus_table`; a solver may substitute its own table, in which case its instructions will indicate the interpretation of the table entries.

You can change a variable's status by use of AMPL's `let` command. This facility is sometimes useful when you want to re-solve a problem after a small, well-defined change. For example, consider the variables defined in our cutting-stock model, `cut1.mod`:

```
param nPAT integer = 0;    # number of patterns
set PATTERNS := 1..nPAT;  # set of patterns

var Cut {PATTERNS} integer = 0; # rolls cut using each
pattern
```

At each pass through the Gilmore-Gomory procedure, as implemented in `cut1.run`, `nPAT` is stepped by one, causing a new variable `Cut[nPAT]` to be created. It has an initial solver status of "none", like all new variables, but it is guaranteed -- by the way that the pattern generation procedure is constructed -- to enter the basis as soon as the expanded cutting problem is re-solved. Thus we give it a status of "bas" instead:

```
let Cut[nPAT].sstatus := "bas";
```

It turns out that this change tends to reduce the number of iterations in each re-optimization of the cutting problem, at least with some simplex solvers. Setting a few statuses in this way is not guaranteed to reduce the number of iterations, however. Its success depends on the particular problem and solver, and on their interaction with a number of complicating factors:

After the problem and statuses have been modified, the statuses conveyed to the solver at the next `solve` may not properly define a basic solution.

After the problem has been modified, AMPL's presolve phase may send a different subset of variables and constraints to the solver (unless option `presolve` is set to zero). As a result, the statuses conveyed to the solver may not correspond to a useful starting point for the next `solve`, and may not properly define a basic solution.

Some solvers, notably MINOS, use the current values as well as the statuses of the variables in constructing the starting point at the next `solve` (unless option `reset_initial_guesses` is set to one).

Each solver has its own way of adjusting the statuses that it receives from AMPL, when necessary, to produce an initial basic solution that it can use. Thus some experimentation is usually necessary to determine whether any particular strategy for modifying the statuses is useful.

For models that have several `var` declarations, AMPL's generic synonyms for variables provide a convenient way of getting information about the statuses of all variables. Using expressions such as `_var`, `_varname` and `_var.sstatus` in a `display` statement, you can produce a quick summary table of the results. You can also easily list the names of all variables that satisfy some condition, by using the statuses to define an appropriate subset of the variables. Here, for example, are the names of all the basic variables:

```
ampl: display {j in 1.._nvars: _var[j].sstatus = "bas"}
```

```

_varname[j];

_varname[j] [*] :=
1  "Make[ 'bands' ,1]"
2  "Make[ 'bands' ,2]"
3  "Make[ 'bands' ,3]"
4  "Make[ 'bands' ,4]"
5  "Make[ 'coils' ,1]"
6  "Make[ 'coils' ,2]"
7  "Make[ 'coils' ,3]"
8  "Make[ 'coils' ,4]"
15 "Inv[ 'coils' ,1]"
22 "Sell[ 'bands' , 'east' ,4]"
32 "Sell[ 'coils' , 'west' ,2]"
33 "Sell[ 'coils' , 'west' ,3]"
;

```

To display the values of the variables along with their names, you could use the same statement with `_varname[j]` replaced by `(_varname[j],_var[j])`.

11.4 Solver statuses of constraints

Implementations of the simplex method typically add one variable for each constraint that they receive from AMPL. Each added variable has a coefficient of 1 or -1 in its associated constraint, and coefficients of 0 in all other constraints. If the associated constraint is an inequality, then the addition is used as a "slack" or "surplus" variable; its bounds are chosen so that it has the effect of turning the inequality into an equivalent equation. If the associated constraint is an equality to begin with, then the added variable is an "artificial" one whose lower and upper bounds are both zero.

A simplex solver gains two advantages from having these "logical" variables added to the "structural" ones it gets from AMPL:

The linear program is converted to a simpler form, in which the only inequalities are the bounds on the variables.

The solver's initialization (or "crash") routines can be designed so that they always find a starting basis quickly.

The artificial variables can also play a special role in helping the solver to find a solution that satisfies all the constraints (a *feasible* solution), but they are not needed for such a purpose in efficient versions of the simplex method. Large-scale implementations thus commonly treat all logical variables in much the same way as the structural ones, with only very minor adjustments to handle the case in which a variable's lower and upper bounds are equal. A basic optimal solution is defined by the collection of basis statuses of all variables.

To accommodate statuses of logical variables, AMPL permits a solver to return status values corresponding to the constraints as well as the variables. The solver status of a constraint -- written as the constraint name suffixed by `.sstatus` -- is interpreted as the status of the logical variable associated with that constraint. For example, in our diet model, where the constraints are all inequalities,

```
subject to diet {i in NUTR}:
```

```
n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
```

the logical variables are slacks that have the same variety of statuses as the structural variables:

```
ampl: model diet.mod;
ampl: data diet2a.dat;
ampl: option show_stats 1;

ampl: solve;

8 variables, all linear
6 constraints, all linear; 47 nonzeros
1 linear objective; 8 nonzeros.

MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032

ampl: display Buy.sstatus;
Buy.sstatus [*] :=
BEEF  bas
  CHK  low
FISH  low
  HAM  upp
  MCH  upp
  MTL  upp
  SPG  bas
  TUR  low
;

ampl: display diet.sstatus;
diet.sstatus [*] :=
  A  bas
  B1 bas
  B2 low
  C  bas
  CAL bas
  NA upp
;
```

There are a total of 6 basic variables, equal in number to the 6 constraints (one for each member of set NUTR) as is always the case at a basic solution. In our transportation model, where the constraints are equalities,

```
subject to Supply {i in ORIG}:
    sum {j in DEST} Trans[i,j] = supply[i];

subject to Demand {j in DEST}:
    sum {i in ORIG} Trans[i,j] = demand[j];
```

the logical variables are artificials that receive the status "equ" when nonbasic. Here's how the statuses for all constraints might be displayed using AMPL's constraint generic synonyms (analogous to the variable synonyms previously shown):

```
ampl: model transp.mod;
ampl: data transp.dat;
```

```

ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 196200

ampl: display _conname, _con.sstatus;
:      _conname      _con.sstatus      :=
1      "Supply[ 'GARY' ]"      equ
2      "Supply[ 'CLEV' ]"      equ
3      "Supply[ 'PITT' ]"      equ
4      "Demand[ 'FRA' ]"      bas
5      "Demand[ 'DET' ]"      equ
6      "Demand[ 'LAN' ]"      equ
7      "Demand[ 'WIN' ]"      equ
8      "Demand[ 'STL' ]"      equ
9      "Demand[ 'FRE' ]"      equ
10     "Demand[ 'LAF' ]"      equ
;

```

One artificial variable, on the constraint Demand['FRA'], is in the optimal basis, though at a value of zero like all artificial variables in any feasible solution. (In fact there must be some artificial variable in every basis for this problem, due to a linear dependency among the equations in the model.)

11.5 AMPL statuses

Only those variables, objectives and constraints that AMPL actually sends to a solver can receive solver statuses on return. So that you can distinguish these from components that are removed prior to a solve, a separate "AMPL status" is also maintained. You can work with AMPL statuses much like solver statuses, by using the suffix `.astatus` in place of `.sstatus`, and referring to option `astatus_table` for a summary of the recognized values:

```

ampl: option astatus_table;
option astatus_table '\
0      in      normal state (in problem)\
1      drop    removed by drop command\
2      pre     eliminated by presolve\
3      fix     fixed by fix command\
4      sub     defined variable, substituted out\
5      unused  not used in current problem\
';

```

Here's an example of the most common cases, using one of our diet models:

```

ampl: model dietu.mod;
ampl: data dietu.dat;

ampl: drop diet_min[ 'CAL' ];
ampl: fix Buy[ 'SPG' ] := 5;
ampl: fix Buy[ 'CHK' ] := 3;

ampl: solve;
MINOS 5.5: optimal solution found.

```

3 iterations, objective 54.76

```
ampl: display Buy.astatus;
Buy.astatus [*] :=
BEEF  in
  CHK  fix
FISH  in
  HAM  in
  MCH  in
  MTL  in
  SPG  fix
  TUR  in
;

ampl: display diet_min.astatus;
diet_min.astatus [*] :=
  A  in
  B1 pre
  B2 pre
  C  in
  CAL drop
;
```

An AMPL status of "in" indicates components that are in the problem sent to the solver, such as variable `Buy['BEEF']` and constraint `diet_min['A']`. Three other statuses indicate components left out of the problem:

Variables `Buy['CHK']` and `Buy['SPG']` have AMPL status "fix" because the `fix` command was used to specify their values in the solution.

Constraint `diet_min['CAL']` has AMPL status "drop" because it was removed by use of the `drop` command.

Constraints `diet_min['B1']` and `diet_min['B2']` have AMPL status "pre" because they were removed from the problem by simplifications performed in AMPL's presolve phase.

The objective and problem commands also have the effect of dropping certain components, such as alternative objectives in this example:

```
ampl: model dietobj.mod;
ampl: data dietobj.dat;
ampl: objective total_cost[ 'JEWEL' ];

ampl: solve;
MINOS 5.5: optimal solution found.
8 iterations, objective 74.3532967

ampl: display total_cost.astatus;
total_cost.astatus [*] :=
'A&P'  drop
JEWEL  in
VONS   drop
;
```

Not shown here are the AMPL status "unused" for a variable that does not appear in any objective or constraint, and "sub" for variables and constraints eliminated by substitution (as explained on pages 247-248 of the AMPL book).

For a variable or constraint, you will normally be interested in only one of the statuses at a time: the solver status if the variable or constraint was included in the problem sent most recently to the solver, or the AMPL status otherwise. Thus AMPL provides the suffix `.status` to denote the one status of interest:

```

ampl: display Buy.status, Buy.astatus, Buy.sstatus;
:      Buy.status Buy.astatus Buy.sstatus      :=
BEEF   low      in      low
CHK    fix      fix      none
FISH   low      in      low
HAM    low      in      low
MCH    bas      in      bas
MTL    low      in      low
SPG    fix      fix      none
TUR    low      in      low
;

ampl: display diet_min.status, diet_min.astatus,
diet_min.sstatus;
:      diet_min.status diet_min.astatus diet_min.sstatus      :=
A      bas      in      bas
B1     pre      pre     none
B2     pre      pre     none
C      low      in      low
CAL    drop     drop     none
;

```

In general, `name.status` is equal to `name.sstatus` if `name.astatus` is "in", and is equal to `name.astatus` otherwise.

12 Suffixes

To represent values associated with a model component, AMPL employs various qualifiers or *suffixes* appended to component names. A suffix consists of a period or "dot" (.) followed by a (usually short) identifier, so that for example the reduced cost associated with a variable `Buy[j]` is written `Buy[j].rc`, and the reduced costs of all such variables can be viewed by giving the command `display Buy.rc`. There are numerous *built-in* suffixes of this kind, available for use in any AMPL session; see our table of built-in suffixes for details.

AMPL cannot anticipate all of the values that a solver might associate with model components, however. The values recognized (as input) or computed (as output) depend on the design of each solver and its algorithms. To provide for the representation of these values, the concept of a suffix has been extended to permit the definition of new suffixes for the duration of an AMPL session.

We first consider suffixes that are defined by the user, generally for the purpose of sending values to a solver. Then we describe how suffixes may be defined by a solver for the purpose of returning additional values.

12.1 User-defined suffixes

Most solvers recognize a variety of algorithmic choices or settings, each governed by a single value that applies to the entire problem being solved. Thus you can alter selected settings by setting up a single string of directives, as in this example applying the CPLEX solver to an integer program:

```
ampl: model multmip3.mod;  
ampl: data multmip3.dat;  
  
ampl: option solver cplex;  
ampl: option cplex_options 'nodesel 3 varsel 2 backtrack  
1.5';  
ampl: solve;  
  
CPLEX: nodesel 3  
varsel 2  
backtrack 1.5
```

```

CPLEX: optimal integer solution; objective 235625
611 simplex iterations
104 branch-and-bound nodes

```

A few kinds of solver settings are more complex, however, in that they require separate values to be set for individual model components. These settings are far too numerous to be accommodated in a directive string. Instead you can now specify them by means of *user-defined* suffixes that the solver interface has been set up to recognize.

As an example, for each variable in an integer program, CPLEX recognizes a separate branching priority and a separate preferred branching direction, represented by an integer in [0,9999] and in [-1,1] respectively. A new version of AMPL's CPLEX driver recognizes the suffixes `.priority` and `.direction` as giving these settings. To make use of these suffixes, you begin by entering a `suffix` command to define each one for the current AMPL session:

```

ampl: reset;
ampl: model multmip3.mod;
ampl: data multmip3.dat;

ampl: suffix priority IN, integer, = 0, <= 9999;
ampl: suffix direction IN, integer, = -1, <= 1;

```

The effect of these statements is to define expressions of the form `name.priority` and `name.direction`, where `name` denotes any variable, objective or constraint of the current model. The argument `IN` specifies that values corresponding to these suffixes are to be read in by the solver, and the subsequent phrases place restrictions on the values that will be accepted (much as in a `param` declaration).

The newly defined suffixed expressions may be assigned values by use of AMPL's `let` command. In particular, for our current example we want to use these suffixes to assign CPLEX priority and direction values corresponding to the binary variables `Use[i,j]`. Normally you will choose such values based on your knowledge of the problem and your past experience with similar problems. Here is one possibility:

```

ampl: let {i in ORIG, j in DEST}
ampl?   Use[i,j].priority := sum {p in PROD} demand[j,p];

ampl: let Use["GARY","FRE"].direction := -1;

```

Variables not assigned a `.priority` or `.direction` value get a default value of zero (as do all constraints and objectives in this example), as you can check:

```

ampl: display Use.direction;
Use.direction [*,*] (tr)

:      CLEV GARY PITT      :=
DET    0        0    0
FRA    0        0    0
FRE    0       -1    0
LAF    0        0    0
LAN    0        0    0
STL    0        0    0
WIN    0        0    0

```

;

With the suffix values assigned as shown, CPLEX's search for a solution turns out to require fewer simplex iterations and fewer branch-and-bound nodes:

```
ampl: solve;

CPLEX: nodesel 3
varsel 2
backtrack 1.5

CPLEX: optimal integer solution; objective 235625
413 simplex iterations
64 branch-and-bound nodes
```

Further information about the suffixes recognized by CPLEX, and how to determine the corresponding settings, can be found in the AMPL User Guide from ILOG. Other solver interfaces may recognize different suffixes for different purposes; you'll need to check separately for each solver you want to use.

12.2 Solver-defined suffixes

The most common use of AMPL suffixes is to represent solver result values that correspond to variables, constraints, and other model components. Yet only the most standard kinds of results, such as reduced costs and slacks, are covered by the built-in suffixes. To allow for other, more solver-specific results of optimization, AMPL permits solver drivers to define new suffixes and to associate solution result information with them.

To exhibit the possibilities, we give examples of *solver-defined* suffixes that are created by the revised CPLEX driver. Sensitivity analysis provides an example of numerical solver-defined suffixes, and infeasibility diagnosis shows how a symbolic (string-valued) suffix works.

Before a solver-defined suffix can be used in an AMPL script, it must be declared. The reporting of a direction of unboundedness gives an example of this situation.

For further information on all of these features, see the CPLEX information in the AMPL User Guide from ILOG.

Sensitivity analysis. When the keyword `sensitivity` is included in CPLEX's list of directives, classical sensitivity ranges are computed and are returned in three new suffixes:

```
ampl: model steelt.mod;
ampl: data steelt.dat;

ampl: option solver cplex;
ampl: option cplex_options 'sensitivity';

ampl: solve;
CPLEX: sensitivity
CPLEX: optimal solution; objective 515033
18 iterations (1 in phase I)
```

```

suffix up OUT;
suffix down OUT;
suffix current OUT;

```

The three lines at the end show the `suffix` commands that are executed by AMPL in response to the results from the solver. These commands are invoked automatically; you do not need to type them. The argument `OUT` in each command says that these are suffixes whose values will be written out by the solver (in contrast to the previous example, where the argument `IN` indicated suffix values that the solver was to read in).

The sensitivity suffixes are interpreted as follows. For variables, suffix `.current` indicates the objective function coefficient in the current problem, while `.down` and `.up` give the smallest and largest values of the objective coefficient for which the current LP basis remains optimal:

```

ampl: display Sell.down, Sell.current, Sell.up;
:      Sell.down Sell.current      Sell.up      :=
bands 1      23.3          25          1e+20
bands 2      25.4          26          1e+20
bands 3      24.9          27          27.5
bands 4      10            27          29.1
coils 1      29.2857       30          30.8571
coils 2      33            35          1e+20
coils 3      35.2857       37          1e+20
coils 4      35.2857       39          1e+20
;

```

For constraints, the interpretation is similar except that it applies to a constraint's constant term (the so-called right-hand side value):

```

ampl: display time.down, time.current, time.up;
:      time.down time.current      time.up      :=
1      37.8071          40          66.3786
2      37.8071          40          47.8571
3      25              32          45
4      30              40          62.5
;

```

You can use AMPL's generic synonyms to display a table of ranges for *all* variables or constraints, similar to the tables produced by the standalone version of CPLEX. (Values of $-1e+20$ in the `.down` column and $1e+20$ in the `.up` column correspond to what CPLEX calls `-infinity` and `+infinity` in its tables.)

Infeasibility diagnosis. For a linear program that has no feasible solution, you can ask CPLEX to find an *irreducible infeasible subset* (or *IIS*) of the constraints and variable bounds, which may provide useful information on the source of the infeasibility. You turn on the IIS finder by changing the `iisfind` directive from its default value of 0 to either 1 (for a relatively fast version) or 2 (for a slower version that tends to find a smaller constraint subset).

The following example shows how IIS finding might be applied to the infeasible diet problem from chapter 2 of the AMPL book. After `solve` detects that there is no feasible solution, it is repeated with the `'iisfind 1'` directive:

```

ampl: model diet.mod;
ampl: data diet2.dat;

ampl: option solver cplex;
ampl: solve;
CPLEX: infeasible problem
7 iterations (7 in phase I)

ampl: option cplex_options 'iisfind 1';
ampl: solve;
CPLEX: iisfind 1
CPLEX: infeasible problem
0 iterations

Returning iis of 7 variables and 2 constraints.

suffix iis symbolic OUT;

option iis_table '\
0      non      not in the iis\
1      low      at lower bound\
2      fix      fixed\
3      upp      at upper bound\
';

```

Again, AMPL shows the suffix statement that has been executed automatically. You can see that the new suffix is named `.iis` and that it is symbolic, or string-valued. An associated option `iis_table`, also set up by the solver driver and displayed automatically by `solve`, shows the strings that may be associated with `.iis` and gives brief descriptions of what they mean.

You can use `display` to look at the `.iis` values that have been returned:

```

ampl: display _varname, _var.iis, _conname, _con.iis;

:      _varname      _var.iis      _conname      _con.iis      :=
1  "Buy[ 'BEEF' ]"    upp          "diet[ 'A' ]"      non
2  "Buy[ 'CHK' ]"     low          "diet[ 'B1' ]"     non
3  "Buy[ 'FISH' ]"    low          "diet[ 'B2' ]"     low
4  "Buy[ 'HAM' ]"     upp          "diet[ 'C' ]"      non
5  "Buy[ 'MCH' ]"     non          "diet[ 'NA' ]"     upp
6  "Buy[ 'MTL' ]"     upp          "diet[ 'CAL' ]"    non
7  "Buy[ 'SPG' ]"     low          .                  .
8  "Buy[ 'TUR' ]"     low          .                  .
;

```

This information indicates that the IIS consists of four lower and three upper bounds on the variables, plus the constraints providing the lower bound on B2 and the upper bound on NA in the diet. Together these restrictions have no feasible solution, but dropping any one of them will permit a solution to be found to the remaining ones.

If dropping the bounds is not of interest, then you may want to list only the constraints that are in the IIS:

```

ampl: display {i in 1.._ncons: _con[i].iis <> "non"}
ampl?      (_conname[i], _con[i].iis);

```

```

:   _conname[i]   _con[i].iis      :=
3   "diet['B2']"   low
5   "diet['NA']"   upp
;

```

AMPL's `expand` command lets you look at these constraints:

```

AMPL: expand {i in 1.._ncons: _con[i].iis <> "non"} _con[i];

s.t. diet['B2']:
      700 <= 15*Buy['BEEF'] + 20*Buy['CHK'] +
10*Buy['FISH'] + 10*Buy['HAM']
      + 15*Buy['MCH'] + 15*Buy['MTL'] + 15*Buy['SPG'] +
10*Buy['TUR'] <=
      20000;

s.t. diet['NA']:
      0 <= 938*Buy['BEEF'] + 2180*Buy['CHK'] +
945*Buy['FISH'] +
      278*Buy['HAM'] + 1182*Buy['MCH'] + 896*Buy['MTL'] +
1329*Buy['SPG'] +
      1397*Buy['TUR'] <= 40000;

```

You might conclude in this case that, if the bounds on purchases are to be met, then you need to allow either less vitamin B2 or more sodium in the diet. Further experimentation would be necessary to determine the amount less or more, however. (After you make adjustments that remove this IIS, the entire problem might become feasible, or CPLEX might find another IIS. The finder's algorithm detects only one IIS at a time.)

Direction of unboundedness. For an unbounded linear program -- one that has in effect a minimum of `-Infinity` or a maximum of `+Infinity` -- the "solution" is characterized by a feasible point together with a direction of unboundedness from that point. On return from CPLEX, the values of the variables define the feasible point. The direction of unboundedness is given by an additional value associated with each variable through the solver-defined suffix `.unbdd`.

An application of the direction of unboundedness can be found in our example of Benders decomposition applied to a transportation-location problem. One part of the decomposition scheme is a subproblem obtained by fixing the variables `Build[i]`, which indicate the warehouses that are to be built, to trial values `build[i]`. In its dual form, this subproblem is:

```

var Supply_Price {ORIG} <= 0;
var Demand_Price {DEST};

maximize Dual_Ship_Cost:
  sum {i in ORIG} Supply_Price[i] * supply[i] * build[i] +
  sum {j in DEST} Demand_Price[j] * demand[j];

subj to Dual_Ship {i in ORIG, j in DEST}:
  Supply_Price[i] + Demand_Price[j] <= var_cost[i,j];

```

When all values `build[i]` are set to zero, no warehouses are built, and the primal subproblem is infeasible. As a result, the dual formulation of the subproblem -- which

always has a feasible solution -- must be unbounded. When this dual problem is solved from the AMPL command line, CPLEX returns the direction of unboundedness in the expected way:

```

ampl: model trnloc1d.mod;
ampl: data trnloc1.dat;

ampl: problem Sub: Supply_Price, Demand_Price,
Dual_Ship_Cost, Dual_Ship;
ampl: let {i in ORIG} build[i] := 0;

ampl: option solver cplex, cplex_options 'presolve 0';
ampl: solve;
CPLEX: presolve 0
CPLEX: unbounded problem
30 iterations (0 in phase I)
variable.unbdd returned

suffix unbdd OUT;

```

The suffix message indicates that .unbdd has been created automatically. You can use this suffix to display the direction of unboundedness, which is quite simple in this case:

```

ampl: display Supply_Price.unbdd;
Supply_Price.unbdd [*] :=
  1 -1      4 -1      7 -1      10 -1      13 -1      16 -1      19 -1      22 -1
25 -1
  2 -1      5 -1      8 -1      11 -1      14 -1      17 -1      20 -1      23 -1
  3 -1      6 -1      9 -1      12 -1      15 -1      18 -1      21 -1      24 -1
;

ampl: display Demand_Price.unbdd;
Demand_Price.unbdd [*] :=
A3 1   A6 1   A8 1   A9 1   B2 1   B4 1
;

```

Our script for Benders decomposition (trnloc1d.mod) solves the subproblem repeatedly, with differing build[i] values generated from the master problem. After each solve, the result is tested for unboundedness and an extension of the master problem is constructed accordingly. The essentials of the main loop are as follows:

```

repeat {
  solve Sub;
  if Dual_Ship_Cost <= Max_Ship_Cost + 0.00001 then break;

  if Sub.result = "unbounded" then {
    let nCUT := nCUT + 1;
    let cut_type[nCUT] := "ray";
    let {i in ORIG}
      supply_price[i,nCUT] := Supply_Price[i].unbdd;
    let {j in DEST}
      demand_price[j,nCUT] := Demand_Price[j].unbdd;
  }
  else {
    let nCUT := nCUT + 1;
  }
}

```

```

        let cut_type[nCUT] := "point";
        let {i in ORIG}
            supply_price[i,nCUT] := Supply_Price[i];
        let {j in DEST}
            demand_price[j,nCUT] := Demand_Price[j];
        }

        solve Master;
        let {i in ORIG} build[i] := Build[i];
    };

```

An attempt to use `.unbdd` in this context fails, however:

```

ampl: include /tmp/trnlocld.run;

/tmp/trnlocld.run, line 41 (offset 991):
    Bad suffix .unbdd for Supply_Price

context: let {i in ORIG} supply_price[i,nCUT] := >>>
Supply_Price[i].unbdd; <<<

```

The difficulty here is that AMPL scans all commands in the repeat loop before beginning to execute any of them. As a result it encounters the use of `.unbdd` before any infeasible subproblem has had a chance to cause this suffix to be defined. To make this script run as intended, it is necessary to place the statement

```
suffix unbdd OUT;
```

in the script before the loop, so that `.unbdd` is already defined at the time the loop is scanned.

12.3 The suffix statement

A new AMPL *suffix* is defined by a statement of the following general form:

```
suffix suffix-name typeopt restriction-phraseopt inoutopt ... ;
```

This statement causes AMPL to recognize *suffixed expressions* of the form *component-name* . *suffix-name*, where *component-name* refers to any currently declared variable, constraint, objective, or problem. The definition of a suffix remains in effect until the next `reset` statement or the end of the current AMPL session.

The *suffix-name* is subject to the same rules as other names in AMPL. Suffixes have a separate name space, however, so that a suffix may have the same name as a parameter, variable, or other model component. The optional phrases of the `suffix` statement may appear in any order; their effects are described below.

A suffixed expression may be assigned (or reassigned) a value by use of AMPL's `let` statement. The optional *type* in the `suffix` statement indicates what values may be assigned, with all numerical values being the default:

<i>type</i>	values allowed
<i>none specified</i>	any numerical value
<i>integer</i>	integer numerical values
<i>binary</i>	0 or 1
<i>symbolic</i>	character strings listed in option <i>suffix-name_table</i>

All numerical-valued suffixed expressions have an initial value of 0. Their permissible values may be further limited by a *restriction-phrase* having either of the forms

```
>= arith-expr
<= arith-expr
```

where *arith-expr* is any valid AMPL arithmetic expression not involving variables.

For each `symbolic` suffix, AMPL automatically defines an associated integer-valued suffix, *suffix-name_num*. An AMPL option *suffix-name_table* must then be created to define a relation between the *.suffix-name* and *.suffix-name_num* values, as in the following example:

```
suffix iis symbolic OUT;

option iis_table '\
0      non      not in the iis\
1      low      at lower bound\
2      fix      fixed\
3      upp      at upper bound\
';
```

Each line of the table consists of an integer value, a string value, and an optional comment. Every string value is associated with its adjacent integer value, and with any higher integer values that are less than the integer on the next line. Assigning a string value to a *.suffix-name* expression is equivalent to assigning the associated numerical value (or one of the associated numerical values) to a *.suffix-name_num* expression. The latter expressions are initially assigned the value 0, and are subject to any *restriction-phrases* as described above. (Normally the string values of `symbolic` suffixes are used in AMPL commands and scripts, while the numerical values are used in communication with solvers.)

The optional *inout* keyword determines how suffix values interact with the solver:

***inout* handling of suffix values**

IN written by AMPL before invoking the solver, then ***read in by solver***
OUT ***written out by solver***, then read by AMPL after the solver is finished
INOUT both read and written, as for IN and OUT above
LOCAL neither read nor written
INOUT is the default if no *inout* keyword is specified.