9 Specifying Data

As we emphasize throughout this book, there is a distinction between an AMPL model for an optimization problem, and the data values that define a particular instance of the problem. Chapters 5 through 8 focused on the declarations of sets, parameters, variables, objectives and constraints that are necessary to describe models. In this chapter and the next, we take a closer look at the statements that specify the data.

Examples of AMPL *data statements* appear in almost every chapter. These statements offer several formats for lists and tables of set and parameter values. Some formats are most naturally created and maintained in a text editing or word processing environment, while others are easy to generate from programs like database systems and spreadsheets. The `display` command (Chapter 12) also produces output in these formats. Wherever possible, similar syntax and concepts are used for both sets and parameters.

This chapter first explains how AMPL's `data` command is used, in conjunction with data statements, to read data values from files such as those whose names end in `.dat` throughout our examples. Options to the `data` command also allow or force selected sets and parameters to be read again.

Subsequent sections describe data statements, first for lists and then for tables of set and parameter data, followed by brief sections on initial values for variables, values for indexed collections of sets, and default values. A summary of data statement formats appears in Section A.12.

A final section describes the `read` command, which reads *unformatted* lists of values into sets and parameters. Chapter 10 is devoted to AMPL's features for data stored in relational database tables.

9.1 Formatted data: the `data` command

Declarations like `param` and `var`, and commands like `solve` and `display`, are executed in *model mode*, the standard mode for most modeling activity. But model mode is inconvenient for reading long lists of set and parameter values. Instead AMPL reads its data statements in a *data mode* that is initiated by the `data` command. In its most common use, this command consists of the keyword `data` followed by the name of a file. For example,

```
ampl: data diet.dat;
```

reads data from a file named `diet.dat`. Filenames containing spaces, semicolons, or nonprinting characters must be enclosed in quotes.

While reading in data mode, AMPL treats white space, that is, any sequence of space, tab, and ‘‘newline’’ characters, as a single space. Commas separating strings or numbers are also ignored. Judicious use of these separators can help to arrange data into easy-to-read lists and tables; our examples use a combination of spaces and newlines. If data statements are produced as output from other data management software and sent directly to AMPL, however, then you may ignore visual appearance and use whatever format is convenient.

Data files often contain numerous character strings, representing set members or the values of symbolic parameters. Thus in data mode AMPL does not, in general, require strings to be enclosed in quotes. Strings that include any character other than letters, digits, underscores, period, + and - must be quoted, however, as in the case of `A&P`. You may use a pair of either single quotes (`'A&P'`) or double quotes (`"A&P"`), unless the string contains a quote, in which case the other kind of quote must surround it (`"DOMINICK'S"`) or the surrounding quote must be doubled within it (`'DOMINICK''S'`).

A string that looks like a number (for example `"+1"` or `"3e4"`) must also be quoted, to distinguish it from a set member or parameter value that is actually a number. Numbers that have the same internal representation are considered to be the same, so that for example `2`, `2.00`, `2.e0` and `0.02E+2` all denote the same set member.

When AMPL finishes reading a file in data mode, it normally reverts to whatever mode it was in before the `data` command was executed. Hence a data file can itself contain `data` commands that read data from other files. If the last data statement in a data file lacks its terminating semicolon, however, then data mode persists regardless of the previous mode.

A `data` command with no filename puts AMPL into data mode, so subsequent input is taken as data statements:

```
ampl: model dietu.mod;
ampl: data;
ampl data: set MINREQ := A B1 B2 C CAL;
ampl data: set MAXREQ := A NA CAL;
ampl data: display NUTR;
set NUTR := A B1 B2 C CAL NA;

ampl:
```

AMPL leaves data mode when it sees any statement (like `display`) that does not begin with a keyword (like `set` or `param`) that begins a `data` statement. The `model` command, with or without filename, also causes a return to model mode.

Model components may be assigned values from any number of data files, by using multiple `data` commands. Regardless of the number of files, AMPL checks that no component is assigned a value more than once, and 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 data values in the new file would normally be treated as erroneous duplicates, but you can tell AMPL to accept them by first giving a `reset data` or `update data` command. These alternatives are described in Section 11.3, along with the use of `reset data` to resample randomly-computed parameters, and of `let` to directly assign new set or parameter values.

9.2 Data in lists

For an unindexed (scalar) parameter, a data statement assigns one value:

```
param avail := 40;
```

Most of a typical model's parameters are indexed over sets, however, and their values are specified in a variety of lists and tables that are introduced in this section and the next, respectively.

We start with sets of simple one-dimensional objects, and the one-dimensional collections of parameters indexed over them. We then turn to two-dimensional sets and parameters, for which we have the additional option of organizing the data into ‘‘slices’’. The options for two dimensions are then shown to generalize readily to higher dimensions, for which we present some three-dimensional examples. Finally, we show how data statements for a set and the parameters indexed over it can be combined to provide a more concise and convenient representation.

Lists of one-dimensional sets and parameters

For a parameter indexed over a one-dimensional set like

```
set PROD;
param rate {PROD} > 0;
```

the specification of the set can be simply a listing of its members:

```
set PROD := bands coils plate ;
```

and the parameter's specification may be virtually the same except for the addition of a value after each set member:

```
param rate := bands 200  coils 140  plate 160 ;
```

The parameter specification could equally well be written

```
param rate :=
        bands  200
        coils  140
        plate  160 ;
```

since extra spaces and line breaks are ignored.

If a one-dimensional set has been declared with the attribute `ordered` or `circular` (Section 5.6), then the ordering of its members is taken from the data statement that defines it. For example, we specified

```
set WEEKS := 27sep 04oct 11oct 18oct ;
```

as the membership of the ordered set `WEEKS` in Figure 5-4.

Members of a set must all be different; AMPL will warn of duplicates:

```
duplicate member coils for set PROD
context:  set PROD := bands coils plate coils  >>> ; <<<
```

Also a parameter may not be given more than one value for each member of the set over which it is indexed. A violation of this rule provokes a similar message:

```
rate['bands'] already defined
context:  param rate := bands 200 bands 160  >>> ; <<<
```

The context bracketed by `>>>` and `<<<` isn't the exact point of the error, but the message makes the situation clear.

A set may be specified as empty by giving an empty list of members; simply put the semicolon right after the `:=` operator. A parameter indexed over an empty set has no data associated with it.

Lists of two-dimensional sets and parameters

The extension of data lists to the two-dimensional case is largely straightforward, but with each set member denoted by a pair of objects. As an example, consider the following sets from Figure 6-2a:

```
set ORIG;   # origins
set DEST;   # destinations

set LINKS within {ORIG,DEST};   # transportation links
```

The members of `ORIG` and `DEST` can be given as for any one-dimensional sets:

```
set ORIG := GARY CLEV PITT ;
set DEST := FRA DET LAN WIN STL FRE LAF ;
```

Then the membership of `LINKS` may be specified as a list of tuples such as you would find in a model's indexing expressions,

```
set LINKS :=
   (GARY,DET) (GARY,LAN) (GARY,STL) (GARY,LAF) (CLEV,FRA)
   (CLEV,DET) (CLEV,LAN) (CLEV,WIN) (CLEV,STL) (CLEV,LAF)
   (PITT,FRA) (PITT,WIN) (PITT,STL) (PITT,FRE) ;
```

or as a list of pairs, without the parentheses and commas:

```
set LINKS :=
   GARY DET   GARY LAN   GARY STL   GARY LAF
   CLEV FRA   CLEV DET   CLEV LAN   CLEV WIN
   CLEV STL   CLEV LAF   PITT FRA   PITT WIN
   PITT STL   PITT FRE ;
```

The order of members within each pair is significant — the first must be from `ORIG`, and the second from `DEST` — but the pairs themselves may appear in any order.

An alternative, more concise way to describe this set of pairs is to list all second components that go with each first component:

```
set LINKS :=
   (GARY,*) DET LAN STL LAF
   (CLEV,*) FRA DET LAN WIN STL LAF
   (PITT,*) FRA WIN STL FRE ;
```

It is also easy to list all first components that go with each second component:

```
set LINKS :=
   (*,FRA) CLEV PITT   (*,DET) GARY CLEV   (*,LAN) GARY CLEV
   (*,WIN) CLEV PITT   (*,LAF) GARY CLEV   (*,FRE) PITT
   (*,STL) GARY CLEV PITT ;
```

An expression such as `(GARY,*)` or `(*,FRA)`, resembling a pair but with a component replaced by a `*`, is a data *template*. Each template is followed by a list, whose entries are substituted for the `*` to generate pairs; these pairs together make up a *slice* through the dimension of the set where the `*` appears. A tuple without any `*`'s, like `(GARY,DET)`, is in effect a template that specifies only itself, so it is not followed by any values. At the other extreme, in the table that consists of pairs alone,

```
set LINKS :=
   GARY DET   GARY LAN   GARY STL   GARY LAF
   CLEV FRA   CLEV DET   CLEV LAN   CLEV WIN
   CLEV STL   CLEV LAF   PITT FRA   PITT WIN
   PITT STL   PITT FRE ;
```

a default template `(*,*)` applies to all entries.

For a parameter indexed over a two-dimensional set, the AMPL list formats are again derived from those for sets by placing parameter values after the set members. Thus if we have the parameter `cost` indexed over the set `LINKS`:

```
param cost {LINKS} >= 0;
```

then the `set` data statement for `LINKS` is extended to become the following `param` data statement for `cost`:

```
param cost :=
   GARY DET 14  GARY LAN 11  GARY STL 16  GARY LAF  8
   CLEV FRA 27  CLEV DET  9  CLEV LAN 12  CLEV WIN  9
   CLEV STL 26  CLEV LAF 17  PITT FRA 24  PITT WIN 13
   PITT STL 28  PITT FRE 99 ;
```

Lists of slices through a set extend similarly, by placing a parameter value after each implied set member. Thus, corresponding to our concise data statement for LINKS:

```
set LINKS :=
   (GARY,*) DET LAN STL LAF
   (CLEV,*) FRA DET LAN WIN STL LAF
   (PITT,*) FRA WIN STL FRE ;
```

there is the following statement for the values of cost:

```
param cost :=
   [GARY,*] DET 14  LAN 11  STL 16  LAF  8
   [CLEV,*] FRA 27  DET  9  LAN 12  WIN  9  STL 26  LAF 17
   [PITT,*] FRA 24  WIN 13  STL 28  FRE 99 ;
```

The templates are given in brackets to distinguish them from the set templates in parentheses, but they work in the same way. Thus a template such as [GARY,*] indicates that the ensuing entries will be for values of cost that have a first index of GARY, and an entry such as DET 14 gives cost["GARY","DET"] a value of 14.

All of the above applies just as well to the use of templates that slice on the first dimension, so that for instance you could also specify parameter cost by:

```
param cost :=
   [*,FRA] CLEV 27  PITT 24
   [*,DET] GARY 14  CLEV  9
   [*,LAN] GARY 11  CLEV 12
   [*,WIN] CLEV  9  PITT 13
   [*,STL] GARY 16  CLEV 26  PITT 28
   [*,FRE] PITT 99
   [*,LAF] GARY  8  CLEV 17
```

You can even think of the list-of-pairs example,

```
param cost :=
   GARY DET 14  GARY LAN 11  GARY STL 16  GARY LAF  8
   ...
```

as also being a case of this form, corresponding to the default template [*,*].

Lists of higher-dimensional sets and parameters

The concepts underlying data lists for two-dimensional sets and parameters extend straightforwardly to higher-dimensional cases. The only difference of any note is that nontrivial slices may be made through more than one dimension. Hence we confine the presentation here to some illustrative examples in three dimensions, followed by a sketch of the general rules for the AMPL data list format that are given in Section A.12.

We take our example from Section 6.3, where we suggest a version of the multicommodity transportation model that defines a set of triples and costs indexed over them:

```
set ROUTES within {ORIG,DEST,PROD};
param cost {ROUTES} >= 0;
```

Suppose that ORIG and DEST are as above, that PROD only has members bands and coils, and that ROUTES has as members certain triples from {ORIG,DEST,PROD}. Then the membership of ROUTES can be given most simply by a list of triples, either

```
set ROUTES :=
   (GARY,LAN,coils) (GARY,STL,coils) (GARY,LAF,coils)
   (CLEV,FRA,bands) (CLEV,FRA,coils) (CLEV,DET,bands)
   (CLEV,DET,coils) (CLEV,LAN,bands) (CLEV,LAN,coils)
   (CLEV,WIN,coils) (CLEV,STL,bands) (CLEV,STL,coils)
   (CLEV,LAF,bands) (PITT,FRA,bands) (PITT,WIN,bands)
   (PITT,STL,bands) (PITT,FRE,bands) (PITT,FRE,coils) ;
```

or

```
set ROUTES :=
   GARY LAN coils   GARY STL coils   GARY LAF coils
   CLEV FRA bands   CLEV FRA coils   CLEV DET bands
   CLEV DET coils   CLEV LAN bands   CLEV LAN coils
   CLEV WIN coils   CLEV STL bands   CLEV STL coils
   CLEV LAF bands   PITT FRA bands   PITT WIN bands
   PITT STL bands   PITT FRE bands   PITT FRE coils ;
```

Using templates as before, but with three items in each template, we can break the specification into slices through one dimension by placing one * in each template. In the following example, we slice through the second dimension:

```
set ROUTES :=
   (CLEV,*,bands) FRA DET LAN STL LAF
   (PITT,*,bands) FRA WIN STL FRE

   (GARY,*,coils) LAN STL LAF
   (CLEV,*,coils) FRA DET LAN WIN STL
   (PITT,*,coils) FRE ;
```

Because the set contains no members with origin GARY and product bands, the template (GARY,*,bands) is omitted.

When the set's dimension is more than two, the slices can also be through more than one dimension. A slice through two dimensions, in particular, naturally involves placing two *'s in each template. Here we slice through both the first and third dimensions:

```
set ROUTES :=
   (*,FRA,*)  CLEV bands  CLEV coils  PITT bands
   (*,DET,*)  CLEV bands  CLEV coils
   (*,LAN,*)  GARY coils  CLEV bands  CLEV coils
   (*,WIN,*)  CLEV coils  PITT bands
   (*,STL,*)  GARY coils  CLEV bands  CLEV coils  PITT bands
   (*,FRE,*)  PITT bands  PITT coils
   (*,LAF,*)  GARY coils  CLEV bands ;
```

Since these templates have two *'s, they must be followed by pairs of components, which are substituted from left to right to generate the set members. For instance the template (*,FRA,*) followed by CLEV bands specifies that (CLEV,FRA,bands) is a member of the set.

Any of the above forms suffices for giving the values of parameter `cost` as well. We could write

```
param cost :=
   [CLEV,*,bands] FRA 27  DET  9  LAN 12  STL 26  LAF 17
   [PITT,*,bands] FRA 24  WIN 13  STL 28  FRE 99

   [GARY,*,coils] LAN 11  STL 16  LAF  8
   [CLEV,*,coils] FRA 23  DET  8  LAN 10  WIN  9  STL 21
   [PITT,*,coils] FRE 81 ;
```

or

```
param cost :=
   [*,*,bands]  CLEV FRA 27  CLEV DET  9  CLEV LAN 12
                CLEV STL 26  CLEV LAF 17  PITT FRA 24
                PITT WIN 13  PITT STL 28  PITT FRE 99

   [*,*,coils]  GARY LAN 11  GARY STL 16  GARY LAF  8
                CLEV FRA 23  CLEV DET  8  CLEV LAN 10
                CLEV WIN  9  CLEV STL 21  PITT FRE 81
```

or

```
param cost :=
   CLEV DET bands  9  CLEV DET coils  8  CLEV FRA bands 27
   CLEV FRA coils 23  CLEV LAF bands 17  CLEV LAN bands 12
   CLEV LAN coils 10  CLEV STL bands 26  CLEV STL coils 21
   CLEV WIN coils  9  GARY LAF coils  8  GARY LAN coils 11
   GARY STL coils 16  PITT FRA bands 24  PITT FRE bands 99
   PITT FRE coils 81  PITT STL bands 28  PITT WIN bands 13 ;
```

By placing the `*`'s in different positions within the templates, we can slice one-dimensionally in any of three different ways, or two-dimensionally in any of three different ways. (The template `[*,*,*]` would specify a three-dimensional list like

```
param cost :=
   CLEV DET bands  9  CLEV DET coils  8  CLEV FRA bands 27
   ...
```

as already shown above.)

More generally, a template for an *n*-dimensional set or parameter in list form must have *n* entries. Each entry is either a legal set member or a `*`. Templates for sets are enclosed in parentheses (like the tuples in set-expressions) and templates for parameters are enclosed in brackets (like the subscripts of parameters). Following a template is a series of items, each item consisting of one set member for each `*`, and additionally one parameter value in the case of a parameter template. Each item defines an *n*-tuple, by substituting its set members for the `*`s in the template; either this tuple is added to the set being specified, or the parameter indexed by this tuple is assigned the value in the item.

A template applies to all items between it and the next template (or the end of the data statement). Templates having different numbers of `*`s may even be used together in the same data statement, so long as each parameter is assigned a value only once. Where no template appears, a template of all `*`s is assumed.

Combined lists of sets and parameters

When we give data statements for a set and a parameter indexed over it, like

```
set PROD := bands coils plate ;
param rate := bands 200  coils 140  plate 160 ;
```

we are specifying the set's members twice. AMPL lets us avoid this duplication by including the set's name in the `param` data statement:

```
param: PROD: rate := bands 200  coils 140  plate 160 ;
```

AMPL uses this statement to determine both the membership of `PROD` and the values of `rate`.

Another common redundancy occurs when we need to supply data for several parameters indexed over the same set, such as `rate`, `profit` and `market` all indexed over `PROD` in Figure 1-4a. Rather than write a separate data statement for each parameter,

```
param rate   := bands  200  coils  140  plate  160 ;
param profit := bands   25  coils   30  plate   29 ;
param market := bands 6000  coils 4000  plate 3500 ;
```

we can combine these statements into one by listing all three parameter names after the keyword `param`:

```
param: rate profit market :=
  bands 200 25 6000  coils 140 30 4000  plate 160 29 3500 ;
```

Since AMPL ignores extra spaces and line breaks, we have the option of rearranging this information into an easier-to-read table:

```
param:    rate  profit  market :=
  bands    200    25      6000
  coils    140    30      4000
  plate    160    29      3500 ;
```

Either way, we still have the option of adding the indexing set's name to the statement,

```
param: PROD:   rate  profit  market :=
       bands    200    25      6000
       coils    140    30      4000
       plate    160    29      3500 ;
```

so that the specifications of the set and all three parameters are combined.

The same rules apply to lists of any higher-dimensional sets and the parameters indexed over them. Thus for our two-dimensional example `LINKS` we could write

```
param: LINKS: cost :=
   GARY DET 14  GARY LAN 11  GARY STL 16  GARY LAF  8
   CLEV FRA 27  CLEV DET  9  CLEV LAN 12  CLEV WIN  9
   CLEV STL 26  CLEV LAF 17  PITT FRA 24  PITT WIN 13
   PITT STL 28  PITT FRE 99 ;
```

to specify the membership of LINKS and the values of the parameter cost indexed over it, or

```
param: LINKS: cost  limit :=
  GARY DET     14    1000
  GARY LAN     11     800
  GARY STL     16    1200
  GARY LAF      8    1100
  CLEV FRA     27    1200
  CLEV DET      9     600
  CLEV LAN     12     900
  CLEV WIN      9     950
  CLEV STL     26    1000
  CLEV LAF     17     800
  PITT FRA     24    1500
  PITT WIN     13    1400
  PITT STL     28    1500
  PITT FRE     99    1200 ;
```

to specify the values of cost and limit together. The same options apply when templates are used, making possible further alternatives such as

```
param: LINKS: cost :=
   [GARY,*] DET 14  LAN 11  STL 16  LAF  8
   [CLEV,*] FRA 27  DET  9  LAN 12  WIN  9  STL 26  LAF 17
   [PITT,*] FRA 24  WIN 13  STL 28  FRE 99 ;
```

and

```
param:   LINKS:  cost  limit :=
  [GARY,*] DET    14    1000
           LAN    11     800
           STL    16    1200
           LAF     8    1100
  [CLEV,*] FRA    27    1200
           DET     9     600
           LAN    12     900
           WIN     9     950
           STL    26    1000
           LAF    17     800
  [PITT,*] FRA    24    1500
           WIN    13    1400
           STL    28    1500
           FRE    99    1200 ;
```

Here the membership of the indexing set is specified along with the two parameters; for example, the template [GARY,*] followed by the set member DET and the values 14 and 1000 indicates that (GARY,DET) is to be added to the set LINKS, that cost[GARY,DET] has the value 14, and that limit[GARY,DET] has the value 1000.

As our illustrations suggest, the key to the interpretation of a param statement that provides values for several parameters or for a set and parameters is in the first line, which consists of param followed by a colon, then optionally the name of an indexing set followed by a colon, then by a list of parameter names terminated by the := assignment operator. Each subsequent item in the list consists of a number of set members equal to the number of *s in the most recent template and then a number of parameter values equal to the number of parameters listed in the first line.

Normally the parameters listed in the first line of a param statement are all indexed over the same set. This need not be the case, however, as seen in the case of Figure 5-1. For this variation on the diet model, the nutrient restrictions are given by

```
set MINREQ;
set MAXREQ;

param n_min {MINREQ} >= 0;
param n_max {MAXREQ} >= 0;
```

so that n_min and n_max are indexed over sets of nutrients that may overlap but that are not likely to be the same.

Our sample data for this model specifies:

```
set MINREQ := A B1 B2 C CAL ;
set MAXREQ := A NA CAL ;

param:   n_min  n_max :=
   A      700   20000
   C      700       .
   B1       0       .
   B2       0       .
   NA       .   50000
   CAL  16000   24000 ;
```

Each period or dot (.) indicates to AMPL that no value is being given for the corresponding parameter and index. For example, since MINREQ does not contain a member NA, the parameter n_min[NA] is not defined; consequently a . is given as the entry for NA and n_min in the data statement. We cannot simply leave a space for this entry, because AMPL will take it to be 50000: data mode processing ignores all extra spaces. Nor should we put a zero in this entry; in that case we will get a message like

```
error processing param n_min:
        invalid subscript n_min['NA'] discarded.
```

when AMPL first tries to access n_min, usually at the first solve.

When we name a set in the first line of a param statement, the set must not yet have a value. If the specification of parameter data in Figure 5-1 had been given as

```
param: NUTR:  n_min  n_max :=
       A       700   20000
       C       700       .
       B1        0       .
       B2        0       .
       NA        .   50000
       CAL   16000   24000 ;
```

AMPL would have generated the error message

```
dietu.dat, line 16 (offset 366):
        NUTR was defined in the model
context:  param: NUTR >>> : <<<   n_min  n_max :=
```

because the declaration of `NUTR` in the model,

```
set NUTR = MINREQ union MAXREQ;
```

defines it already as the union of `MINREQ` and `MAXREQ`.

9.3 Data in tables

The table format of data, with indices running along the left and top edges and values corresponding to pairs of indices, can be more concise or easier to read than the list format described in the previous section. Here we describe tables first for two-dimensional parameters and then for slices from higher-dimensional ones. We also show how the corresponding multidimensional sets can be specified in tables that have entries of + or − rather than parameter value entries.

AMPL also supports a convenient extension of the table format, in which more than two indices may appear along the left and top edge. The rules for specifying such tables are provided near the end of this section.

Two-dimensional tables

Data values for a parameter indexed over two sets, such as the shipping cost data from the transportation model of Figure 3-1a:

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

are very naturally specified in a table (Figure 3-1b):

```
param cost:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
       GARY   39   14   11   14   16   82    8
       CLEV   27    9   12    9   26   95   17
       PITT   24   14   17   13   28   99   20 ;
```

The row labels give the first index and the column labels the second index, so that for example `cost["GARY","FRA"]` is set to 39. To enable AMPL to recognize this as a table, a colon must follow the parameter name, while the `:=` operator follows the list of column labels.

For larger index sets, the columns of tables become impossible to view within the width of a single screen or page. To deal with this situation, AMPL offers several alternatives, which we illustrate on the small table above.

When only one of the index sets is uncomfortably large, the table may be transposed so that the column labels correspond to the smaller set:

```
param cost (tr):
        GARY CLEV PITT :=
   FRA   39   27   24
   DET   14    9   14
   LAN   11   12   17
   WIN   14    9   13
   STL   16   26   28
   FRE   82   95   99
   LAF    8   17   20 ;
```

The notation `(tr)` after the parameter name indicates a transposed table, in which the column labels give the first index and the row labels the second index. When both of the index sets are large, either the table or its transpose may be divided up in some way. Since line breaks are ignored, each row may be divided across several lines:

```
param cost:  FRA  DET  LAN  WIN
             STL  FRE  LAF      :=
       GARY   39   14   11   14
              16   82    8
       CLEV   27    9   12    9
              26   95   17
       PITT   24   14   17   13
              28   99   20      ;
```

Or the table may be divided columnwise into several smaller ones:

```
param cost:  FRA  DET  LAN  WIN :=
       GARY   39   14   11   14
       CLEV   27    9   12    9
       PITT   24   14   17   13

          :  STL  FRE  LAF :=
       GARY   16   82    8
       CLEV   26   95   17
       PITT   28   99   20 ;
```

A colon indicates the start of each new sub-table; in this example, each has the same row labels, but a different subset of the column labels.

In the alternative formulation of this model presented in Figure 6-2a, `cost` is not indexed over all combinations of members of `ORIG` and `DEST`, but over a subset of pairs from these sets:

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

As we have seen in Section 9.2, the membership of `LINKS` can be given concisely by a list of pairs:

```
set LINKS :=
   (GARY,*) DET LAN STL LAF
   (CLEV,*) FRA DET LAN WIN STL LAF
   (PITT,*) FRA WIN STL FRE ;
```

Rather than being given in a similar list, the values of `cost` can be given in a table like this:

```
param cost:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
      GARY     .   14   11    .   16    .    8
      CLEV    27    9   12    9   26    .   17
      PITT    24    .    .   13   28   99    . ;
```

A `cost` value is given for all pairs that exist in `LINKS`, while a dot (`.`) serves as a place-holder for pairs that are not in `LINKS`. The dot can appear in any AMPL table to indicate ‘‘no value specified here’’.

The set `LINKS` may itself be given by a table that is analogous to the one for `cost`:

```
set LINKS:   FRA  DET  LAN  WIN  STL  FRE  LAF :=
      GARY    -    +    +    -    +    -    +
      CLEV    +    +    +    +    +    -    +
      PITT    +    -    -    +    +    +    -  ;
```

A + indicates a pair that is a member of the set, and a − indicates a pair that is not a member. Any of AMPL’s table formats for specifying parameters can be used for sets in this way.

Two-dimensional slices of higher-dimensional data

To provide data for parameters of more than two dimensions, we can specify the values in two-dimensional slices that are represented as tables. The rules for using slices are much the same as for lists. As an example, consider again the three-dimensional parameter `cost` defined by

```
set ROUTES within {ORIG,DEST,PROD};
param cost {ROUTES} >= 0;
```

The values for this parameter that we specified in list format in the previous section as

```
param cost :=
   [*,*,bands]  CLEV FRA 27  CLEV DET  9  CLEV LAN 12
                CLEV STL 26  CLEV LAF 17  PITT FRA 24
                PITT WIN 13  PITT STL 28  PITT FRE 99

   [*,*,coils]  GARY LAN 11  GARY STL 16  GARY LAF  8
                CLEV FRA 23  CLEV DET  8  CLEV LAN 10
                CLEV WIN  9  CLEV STL 21  PITT FRE 81
```

can instead be written in table format as

```
param cost :=

 [*,*,bands]: FRA  DET  LAN  WIN  STL  FRE  LAF :=
        CLEV   27    9   12    .   26    .   17
        PITT   24    .    .   13   28   99    .

 [*,*,coils]: FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY    .    .   11    .   16    .    8
        CLEV   23    8   10    9   21    .    .
        PITT    .    .    .    .    .   81    . ;
```

Since we are working with two-dimensional tables, there must be two *'s in the templates. A table value's row label is substituted for the first *, and its column label for the second, unless the opposite is specified by (tr) right after the template. You can omit any rows or columns that would have no significant entries, such as the row for GARY in the [*,*,bands] table above.

As before, a dot in the table for any slice indicates a tuple that is not a member of the table.

An analogous table to specify the set ROUTES can be constructed by putting a + where each number appears:

```
set ROUTES :=

 (*,*,bands): FRA DET LAN WIN STL FRE LAF :=
        CLEV   +   +   +   -   +   -   +
        PITT   +   -   -   +   +   +   -

 (*,*,coils): FRA DET LAN WIN STL FRE LAF :=
        GARY   -   -   +   -   +   -   +
        CLEV   +   +   +   +   +   -   -
        PITT   -   -   -   -   -   +   -  ;
```

Since the templates are now set templates rather than parameter templates, they are enclosed in parentheses rather than brackets.

Higher-dimensional tables

By putting more than one index to the left of each row or at the top of each column, you can describe multidimensional data in a single table rather than a series of slices. We'll continue with the three-dimensional cost data to illustrate some of the wide variety of possibilities.

By putting the first two indices, from sets ORIG and DEST, to the left, with the third index from set PROD at the top, we produce the following three-dimensional table of the costs:

```
param cost: bands coils :=
  CLEV FRA    27    23
  CLEV DET     8     8
  CLEV LAN    12    10
  CLEV WIN     .     9
  CLEV STL    26    21
  CLEV LAF    17     .
  PITT FRA    24     .
  PITT WIN    13     .
  PITT STL    28     .
  PITT FRE    99    81
  GARY LAN     .    11
  GARY STL     .    16
  GARY LAF     .     8 ;
```

Putting only the first index to the left, and the second and third at the top, we arrive instead at the following table, which for convenience we break into two pieces:

```
param cost:  FRA   DET   LAN   WIN   STL   FRE   LAF
          : bands bands bands bands bands bands bands :=
       CLEV   27     9    12     .    26     .    17
       PITT   24     .     .    13    28    99     .

          :  FRA   DET   LAN   WIN   STL   FRE   LAF
          : coils coils coils coils coils coils coils :=
       GARY    .     .    11     .    16     .     8
       CLEV   23     8    10     9    21     .     .
       PITT    .     .     .     .     .    81     . ;
```

In general a colon must precede each of the table heading lines, while a := is placed only after the last heading line.

The indices are taken in the order that they appear, first at the left and then at the top, if no indication is given to the contrary. As with other tables, you can add the indicator (tr) to transpose the table, so that the indices are still taken in order but first from the top and then from the left:

```
param cost (tr): CLEV CLEV CLEV CLEV CLEV CLEV
               :  FRA  DET  LAN  WIN  STL  LAF :=
         bands     27    8   12    .   26   17
         coils     23    8   10    9   21    .

               : PITT PITT PITT PITT GARY GARY GARY
               :  FRA  WIN  STL  FRE  LAN  STL  LAF :=
         bands     24   13   28   99    .    .    .
         coils      .    .    .   81   11   16    8 ;
```

Templates can also be used to specify more precisely what goes where. For multidimensional tables the template has two symbols in it, * to indicate those indices that appear at the left and : to indicate those that appear at the top. For example the template [*,:,*] gives a representation in which the first and third indices are at the left and the second is at the top:

```
param cost :=
   [*,:,*] :  FRA  DET  LAN  WIN  STL  FRE  LAF :=
  CLEV bands   27    9   12    .   26    .   17
  CLEV coils   23    8   10    9   21    .    .
  PITT bands   24    .    .   13   28   99    .
  PITT coils    .    .    .    .    .   81    .
  GARY coils    .    .   11    .   16    .    8 ;
```

The ordering of the indices is always preserved in tables of this kind. The third index is never correctly placed before the first, for example, no matter what transposition or templates are employed.

For parameters of four or more dimensions, the ideas of slicing and multidimensional tables can be applied together provide an especially broad choice of table formats. If `cost` were indexed over `ORIG`, `DEST`, `PROD`, and `1..T`, for instance, then the templates `[*,:,bands,*]` and `[*,:,coils,*]` could be used to specify two slices through the third index, each specified by a multidimensional table with two indices at the left and one at the top.

Choice of format

The arrangement of slices to represent multidimensional data has no effect on how the data values are used in the model, so you can choose the most convenient format. For the `cost` parameter above, it may be appealing to slice along the third dimension, so that the data values are organized into one shipping-cost table for each product. Alternatively, placing all of the origin-product pairs at the left gives a particularly concise representation. As another example, consider the `revenue` parameter from Figure 6-3:

```
set PROD;          # products
set AREA {PROD};   # market areas for each product
param T > 0;       # number of weeks

param revenue {p in PROD, AREA[p], 1..T} >= 0;
```

Because the index set `AREA[p]` is potentially different for each product `p`, slices through the first (`PROD`) dimension are most attractive. In the sample data from Figure 6-4, they look like this:

```
param T := 4 ;
set PROD := bands coils ;
set AREA[bands] := east north ;
set AREA[coils] := east west export ;

param revenue :=
  [bands,*,*]:   1      2      3      4   :=
      east     25.0   26.0   27.0   27.0
      north    26.5   27.5   28.0   28.5
  [coils,*,*]:   1      2      3      4   :=
      east      30     35     37     39
      west      29     32     33     35
      export    25     25     25     28 ;
```

We have a separate revenue table for each product p, with market areas from AREA[p] labeling the rows, and weeks from 1..T labeling the columns.

9.4 Other features of data statements

Additional features of the AMPL data format are provided to handle special situations. We describe here the data statements that specify default values for parameters, that define the membership of individual sets within an indexed collection of sets, and that assign initial values to variables.

Default values

Data statements must provide values for exactly the parameters in your model. You will receive an error message if you give a value for a nonexistent parameter:

```
error processing param cost:
        invalid subscript cost['PITT','DET','coils'] discarded.
```

or if you fail to give a value for a parameter that does exist:

```
error processing objective Total_Cost:
        no value for cost['CLEV','LAN','coils']
```

The error message appears the first time that AMPL tries to use the offending parameter, usually after you type solve.

If the same value would appear many times in a data statement, you can avoid specifying it repeatedly by including a default phrase that provides the value to be used when no explicit value is given. For example, suppose that the parameter cost above is indexed over all possible triples:

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

param cost {ORIG,DEST,PROD} >= 0;
```

but that a very high cost is assigned to routes that should not be used. This can be expressed as

```
param cost  default 9999  :=
 [*,*,bands]: FRA  DET  LAN  WIN  STL  FRE  LAF :=
        CLEV   27    9   12    .   26    .   17
        PITT   24    .    .   13   28   99    .
 [*,*,coils]: FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY    .    .   11    .   16    .    8
        CLEV   23    8   10    9   21    .    .
        PITT    .    .    .    .    .   81    . ;
```

Missing parameters like `cost["GARY","FRA","bands"]`, as well as those explicitly marked ‘‘omitted’’ by use of a dot (like `cost["GARY","FRA","coils"]`), are given the value 9999. In total, 24 values of 9999 are assigned.

The `default` feature is especially useful when you want all parameters of an indexed collection to be assigned the same value. For instance, in Figure 3-2, we apply a transportation model to an assignment problem by setting all supplies and demands to 1. The model declares

```
param supply {ORIG} >= 0;
param demand {DEST} >= 0;
```

but in the data we give only a default value:

```
param supply default 1 ;
param demand default 1 ;
```

Since no other values are specified, the default of 1 is automatically assigned to every element of `supply` and `demand`.

As explained in Chapter 7, a parameter declaration in the model may include a `default` expression. This offers an alternative way to specify a single default value:

```
param cost {ORIG,DEST,PROD} >= 0, default 9999;
```

If you just want to avoid storing a lot of 9999’s in a data file, however, it is better to put the `default` phrase in the data statement. The `default` phrase should go in the model when you want the default value to depend in some way on other data. For instance, a different arbitrarily large cost could be given for each product by specifying:

```
param huge_cost {PROD} > 0;
param cost {ORIG, DEST, p in PROD} >= 0, default huge_cost[p];
```

A discussion of `default`’s relation to the `=` phrase in `param` statements is given in Section 7.5.

Indexed collections of sets

For an indexed collection of sets, separate data statements specify the members of each set in the collection. In the example of Figure 6-3, for example, the sets named `AREA` are indexed by the set `PROD`:

```
set PROD;          # products
set AREA {PROD};   # market areas for each product
```

The membership of these sets is given in Figure 6-4 by:

```
set PROD := bands coils ;
set AREA[bands] := east north ;
set AREA[coils] := east west export ;
```

Any of the data statement formats for a set may be used with indexed collections of sets. The only difference is that the set name following the keyword `set` is subscripted.

As for other sets, you may specify one or more members of an indexed collection to be empty, by giving an empty list of elements. If you want to provide a data statement only for those members of an indexed collection that are not empty, define the empty set as the default value in the model:

```
set AREA {PROD} default {};
```

Otherwise you will be warned about any set whose data statement is not provided.

Initial values for variables

You may optionally assign initial values to the variables of a model, using any of the options for assigning values to parameters. A variable's name stands for its value, and a constraint's name stands for the associated dual variable's value. (See Section 12.5 for a short explanation of dual variables.)

Any `param` data statement may specify initial values for variables. The variable or constraint name is simply used in place of a parameter name, in any of the formats described by the previous sections of this chapter. To help clarify the intent, the keyword `var` may be substituted for `param` at the start of a data statement. For example, the following data table gives initial values for the variable `Trans` of Figure 3-1a:

```
var Trans:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
      GARY  100  100  800  100  100  500  200
      CLEV  900  100  100  500  500  200  200
      PITT  100  900  100  500  100  900  200 ;
```

As another example, in the model of Figure 1-4, a single table can give values for the parameters `rate`, `profit` and `market`, and initial values for the variables `Make`:

```
param:    rate  profit  market  Make :=
  bands    200    25     6000   3000
  coils    140    30     4000   2500
  plate    160    29     3500   1500 ;
```

All of the previously described features for default values also apply to variables.

Initial values of variables (as well as the values of expressions involving these initial values) may be viewed before you type `solve`, using the `display`, `print` or `printf` commands described in Sections 12.1 through 12.4. Initial values are also optionally passed to the solver, as explained in Section 14.1 and A.18.1. After a solution is returned, the variables no longer have their initial values, but even then you can refer to the initial values by placing an appropriate suffix after the variable's name, as shown in Section A.11.

The most common use of initial values is to give a good starting guess to a solver for nonlinear optimization, which is discussed in Chapter 18.

9.5 Reading unformatted data: the `read` command

The `read` command provides a particularly simple way of getting values into AMPL, given that the values you need are listed in a regular order in a file. The file must be *unformatted* in the sense that it contains nothing except the values to be read — no set or parameter names, no colons or `:=` operators.

In its simplest form, `read` specifies a list of parameters and a file from which their values are to be read. The values in the file are assigned to the entries in the list in the order that they appear. For example, if you want to read the number of weeks and the hours available each week for our simple production model (Figure 4-4),

```
param T > 0;
param avail {1..T} >= 0;
```

from a file `week_data.txt` containing

```
4
40 40 32 40
```

then you can give the command

```
read T, avail[1], avail[2], avail[3], avail[4] <week_data.txt;
```

Or you can use an indexing expression to say the same thing more concisely and generally:

```
read T, {t in 1..T} avail[t] <week_data.txt;
```

The notation < *filename* specifies the name of a file for reading. (Analogously, > indicates writing to a file; see A.15.)

In general, the `read` command has the form

`read` *item-list* < *filename* `;`

with the *item-list* being a comma-separated list of items that may each be any of the following:

parameter
`{` *indexing* `}` *parameter*
`{` *indexing* `}` `(` *item-list* `)`

The first two are used in our example above, while the third allows for the same indexing to be applied to several items. Using the same production example, to read in values for

```
param prodcost {PROD} >= 0;
param invcost {PROD} >= 0;
param revenue {PROD,1..T} >= 0;
```

from a file organized by parameters, you could read each parameter separately:

```
read {p in PROD} prodcost[p] < cost_data;
read {p in PROD} invcost[p] < cost_data;
read {p in PROD, t in 1..T} revenue[p,t] < cost_data;
```

reading from file `cost_data` first all the production costs, then all the inventory costs, and then all the revenues.

If the data were organized by product instead, you could say

```
read {p in PROD}
   (prodcost[p], invcost[p], {t in 1..T} revenue[p,t])
      <cost_data;
```

to read the production and inventory costs and the revenues for the first product, then for the second product, and so forth.

A parenthesized *item-list* may itself contain parenthesized *item-list*s, so that if you also want to read

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

from the same file at the same time, you could say

```
read {p in PROD} (prodcost[p], invcost[p],
   {t in 1..T} (revenue[p,t], market[p,t])) <cost_data;
```

in which case for each product you would read the two costs as before, and then for each week the product's revenue and market demand.

As our descriptions suggest, the form of a `read` statement's *item-list* depends on how the data values are ordered in the file. When you are reading data indexed over sets of strings that, like `PROD`, are not inherently ordered, then the order in which values are read is the order in which AMPL is internally representing them. If the members of the set came directly from a `set` data statement, then the ordering will be the same as in the data statement. Otherwise, it is best to put an `ordered` or `ordered by` phrase in the model's `set` declaration to ensure that the ordering is always what you expect; see Section 5.6 for more about ordered sets.

An alternative that avoids knowing the order of the members in a set is to specify them explicitly in the file that is read. As an example, consider how you might use a `read` statement rather than a data statement to get the values from the `cost` parameter of Section 9.4 that was defined as

```
param cost {ORIG,DEST,PROD} >= 0, default 9999;
```

You could set up the `read` statement as follows:

```
param ntriples integer;
param ic symbolic in ORIG;
param jc symbolic in DEST;
param kc symbolic in PROD;

read ntriples, {1..ntriples}
   (ic, jc, kc, cost[ic,jc,kc]) <cost_data;
```

The corresponding file `cost_data` must begin something like this:

```
18
CLEV FRA bands 27
PITT FRA bands 24
CLEV FRA coils 23
...
```

with 15 more entries needed to give all 18 data values shown in the Section 9.4 example.

Strings in a file for the `read` command that include any character other than letters, digits, underscores, period, + and - must be quoted, just as for data mode. However, the `read` statement itself is interpreted in model mode, so if the statement refers to any particular string, as in, say,

```
read {t in 1..T} revenue ["bands",t];
```

that string must be quoted. The filename following < need not be quoted unless it contains spaces, semicolons, or nonprinting characters.

If a `read` statement contains no < *filename*, values are read from the current input stream. Thus if you have typed the `read` command at an AMPL prompt, you can type the values at subsequent prompts until all of the listed items have been assigned values. For example:

```
ampl: read T, {t in 1..T} avail[t];
ampl? 4
ampl? 40 40 32 40
ampl: display avail;
avail [*] :=
1 40   2 40   3 32   4 40
;
```

The prompt changes from `ampl?` back to `ampl:` when all the needed input has been read.

The filename ‘‘-’’ (a literal minus sign) is taken as the standard input of the AMPL process; this is useful for providing input interactively.

Further uses of `read` within AMPL scripts, to read values directly from script files or to prompt users for values at the command line, are described in Chapter 13.

All of our examples assume that underlying sets such as `ORIG` and `PROD` have already been assigned values, through data statements as described earlier in this chapter, or through other means such as database access or assignment to be described in later chapters. Thus the `read` statement would normally supplement rather than replace other input commands. It is particularly useful in handling long files of data that are generated for certain parameters by programs outside of AMPL.

Exercises

9-1. Section 9.2 gave a variety of data statements for a three-dimensional set, `ROUTES`. Construct some other alternatives for this set as follows:

(a) Use templates that look like `(CLEV,FRA,*)`.

(b) Use templates that look like `(*,*,bands)`, with the list format.

(c) Use templates that look like `(CLEV,*,*)`, with the table format.

(d) Specify some of the set's members using templates with one `*`, and some using templates with two `*`'s.

9-2. Rewrite the production model data of Figure 5-4 so that it consists of just three data statements arranged as follows:

The set `PROD` and parameters `rate`, `inv0`, `prodcost` and `invcost` are given in one table.

The set `WEEKS` and parameter `avail` are given in one table.

The parameters `revenue` and `market` are given in one table.

9-3. For the assignment problem whose data is depicted in Figure 3-2, suppose that the only information you receive about people's preferences for offices is as follows:

Coullard	M239	M233	D241	D237	D239		
Daskin	D237	M233	M239	D241	D239	C246	C140
Hazen	C246	D237	M233	M239	C250	C251	D239
Hopp	D237	M233	M239	D241	C251	C250	
Iravani	D237	C138	C118	D241	D239		
Linetsky	M233	M239	C250	C251	C246	D237	
Mehrotra	D237	D239	M239	M233	D241	C118	C251
Nelson	D237	M233	M239				
Smilowitz	M233	M239	D239	D241	C251	C250	D237
Tamhane	M239	M233	C251	C250	C118	C138	D237
White	M239	M233	D237	C246			

This means that, for example, Coullard's first choice is M239, her second choice is M233, and so on through her fifth choice, D239, but she hasn't given any preference for the other offices.

To use this information with the transportation model of Figure 3-1a as explained in Chapter 3, you must set `cost["Coullard","M239"]` to 1, `cost["Coullard","M233"]` to 2, and so forth. For an office not ranked, such as C246, you can set `cost["Coullard","C246"]` to 99, to indicate that it is a highly undesirable assignment.

(a) Using the list format and a `default` phrase, convert the information above to an appropriate AMPL data statement for the parameter `cost`.

(b) Do the same, but with a table format.

9-4. Sections 9.2 and 9.3 gave a variety of data statements for a three-dimensional parameter, `cost`, indexed over the set `ROUTES` of triples. Construct some other alternatives for this parameter as follows:

(a) Use templates that look like `[CLEV,FRA,*]`.

(b) Use templates that look like `[*,*,bands]`, employing the list format.

(c) Use templates that look like `[CLEV,*,*]`, employing the table format.

(d) Specify some of the parameter values using templates with one `*`, and some using templates with two `*`'s.

9-5. For the three-dimensional parameter `revenue` of Figure 6-4, construct alternative data statements as follows:

(a) Use templates that look like `[*,east,*]`, employing the table format.

(b) Use templates that look like `[*,*,1]`, employing the table format.

(c) Use templates that look like `[bands,*,1]`.

9-6. Given the following declarations,

```
set ORIG;
set DEST;
var Trans {ORIG, DEST} >= 0;
```

how could you use a data statement to assign an initial value of 300 to all of the `Trans` variables?