5

Simple Sets and Indexing

The next four chapters of this book are a comprehensive presentation of AMPL's facilities for linear programming. The organization is by language features, rather than by model types as in the four preceding tutorial chapters. Since the basic features of AMPL tend to be closely interrelated, we do not attempt to explain any one feature in isolation. Rather, we assume at the outset a basic knowledge of AMPL such as Chapters 1 through 4 provide.

We begin with sets, the most fundamental components of an AMPL model. Almost all of the parameters, variables, and constraints in a typical model are indexed over sets, and many expressions contain operations (usually summations) over sets. Set indexing is the feature that permits a concise model to describe a large mathematical program.

Because sets are so fundamental, AMPL offers a broad variety of set types and operations. A set's members may be strings or numbers, ordered or unordered; they may occur singly, or as ordered pairs, triples or longer ''tuples''. Sets may be defined by listing or computing their members explicitly, by applying operations like union and intersection to other sets, or by specifying arbitrary arithmetic or logical conditions for membership.

Any model component or iterated operation can be indexed over any set, using a standard form of indexing expression. Even sets themselves may be declared in collections indexed over other sets.

This chapter introduces the simpler kinds of sets, as well as set operations and indexing expressions; it concludes with a discussion of ordered sets. Chapter 6 shows how these ideas are extended to compound sets, including sets of pairs and triples, and indexed collections of sets. Chapter 7 is devoted to parameters and expressions, and Chapter 8 to the variables, objectives and constraints that make up a linear program.

5.1 Unordered sets

The most elementary kind of AMPL set is an unordered collection of character strings. Usually all of the strings in a set are intended to represent instances of the same kind of

entity — such as raw materials, products, factories or cities. Often the strings are chosen to have recognizable meanings (`coils`, `FISH`, `New_York`), but they could just as well be codes known only to the modeler (`23RPFG`, `486/33C`). A literal string that appears in an AMPL model must be delimited by quotes, either single (`'A&P'`) or double (`"Bell+Howell"`). In all contexts, upper case and lower case letters are distinct, so that for example `"fish"`, `"Fish"`, and `"FISH"` represent different set members.

The declaration of a set need only contain the keyword `set` and a name. For example, a model may declare

```
set PROD;
```

to indicate that a certain set will be referred to by the name `PROD` in the rest of the model. A name may be any sequence of letters, numerals, and underscore (`_`) characters that is not a legal number. A few names have special meanings in AMPL, and may only be used for specific purposes, while a larger number of names have predefined meanings that can be changed if they are used in some other way. For example, `sum` is reserved for the iterated addition operator; but `prod` is merely pre-defined as the iterated multiplication operator, so you can redefine `prod` as a set of products:

```
set prod;
```

A list of reserved words is given in Section A.1.

A declared set's membership is normally specified as part of the data for the model, in the manner to be described in Chapter 9; this separation of model and data is recommended for most mathematical programming applications. Occasionally, however, it is desirable to refer to a particular set of strings within a model. A literal set of this kind is specified by listing its members within braces:

```
{"bands", "coils", "plate"}
```

This expression may be used anywhere that a set is valid, for example in a model statement that gives the set `PROD` a fixed membership:

```
set PROD = {"bands", "coils", "plate"};
```

This sort of declaration is best limited to cases where a set's membership is small, is a fundamental aspect of the model, or is not expected to change often. Nevertheless we will see that the = phrase is often useful in set declarations, for the purpose of defining a set in terms of other sets and parameters. The operator = may be replaced by `default` to initialize the set while allowing its value to be overridden by a data statement or changed by subsequent assignments. These options are more important for parameters, however, so we discuss them more fully in Section 7.5.

Notice that AMPL makes a distinction between a string such as `"bands"` and a set like `{"bands"}` that has a membership of one string. The set that has no members (the empty set) is denoted `{}`.

5.2 Sets of numbers

Set members may also be numbers. In fact a set's members may be a mixture of numbers and strings, though this is seldom the case. In an AMPL model, a literal number is written in the customary way as a sequence of digits, optionally preceded by a sign, containing an optional decimal point, and optionally followed by an exponent; the exponent consists of a `d`, `D`, `e`, or `E`, optionally a sign, and a sequence of digits. A number (`1`) and the corresponding string (`"1"`) are distinct; by contrast, different representations of the same number, such as `100` and `1E+2`, stand for the same set member.

A set of numbers is often a sequence that corresponds to some progression in the situation being modeled, such as a series of weeks or years. Just as for strings, the numbers in a set can be specified as part of the data, or can be specified within a model as a list between braces, such as `{1,2,3,4,5,6}`. This sort of set can be described more concisely by the notation `1..6`. An additional `by` clause can be used to specify an interval other than 1 between the numbers; for instance,

```
1990 .. 2020 by 5
```

represents the set

```
{1990, 1995, 2000, 2005, 2010, 2015, 2020}
```

This kind of expression can be used anywhere that a set is appropriate, and in particular within the assignment phrase of a set declaration:

```
set YEARS = 1990 .. 2020 by 5;
```

By giving the set a short and meaningful name, this declaration may help to make the rest of the model more readable.

It is not good practice to specify all the numbers within a `..` expression by literals like `2020` and `5`, unless the values of these numbers are fundamental to the model or will rarely change. A better arrangement is seen in the multiperiod production example of Figures 4-4 and 4-5, where a parameter `T` is declared to represent the number of periods, and the expressions `1..T` and `0..T` are used to represent sets of periods over which parameters, variables, constraints and sums are indexed. The value of `T` is specified in the data, and is thus easily changed from one run to the next. As a more elaborate example, we could write

```
param start integer;
param end > start integer;
param interval > 0 integer;

set YEARS = start .. end by interval;
```

If subsequently we were to give the data as

```
param start := 1990;
param end := 2020;
param interval := 5;
```

then YEARS would be the same set as in the previous example (as it would also be if end were 2023.) You may use any arithmetic expression to represent any of the values in a .. expression.

The members of a set of numbers have the same properties as any other numbers, and hence can be used in arithmetic expressions. A simple example is seen in Figure 4-4, where the material balance constraint is declared as

```
subject to Balance {p in PROD, t in 1..T}:
   Make[p,t] + Inv[p,t-1] = Sell[p,t] + Inv[p,t];
```

Because t runs over the set 1..T, we can write Inv[p,t-1] to represent the inventory at the end of the previous week. If t instead ran over a set of strings, the expression t-1 would be rejected as an error.

Set members need not be integers. AMPL attempts to store each numerical set member as the nearest representable floating-point number. You can see how this works out on your computer by trying an experiment like the following:

```
ampl: option display_width 50;
ampl: display -5/3 .. 5/3 by 1/3;
set -5/3 .. 5/3 by 1/3 :=
-1.6666666666666667      0.3333333333333326
-1.3333333333333335      0.6666666666666663
-1                       0.9999999999999998
-0.6666666666666667      1.3333333333333333
-0.3333333333333335      1.6666666666666663
-2.220446049250313e-16;
```

You might expect 0 and 1 to be members of this set, but things do not work out that way due to rounding error in the floating-point computations. It is unwise to use fractional numbers in sets, if your model relies on set members having precise values. There should be no comparable problem with integer members of reasonable size; integers are represented exactly for magnitudes up to 2^{53} (approximately 10^{16}) for IEEE standard arithmetic, and up to 2^{47} (approximately 10^{14}) for almost any computer in current use.

5.3 Set operations

AMPL has four operators that construct new sets from existing ones:

A union B	union: in either A or B
A inter B	intersection: in both A and B
A diff B	difference: in A but not B
A symdiff B	symmetric difference: in A or B but not both

The following excerpt from an AMPL session shows how these work:

```
ampl: set Y1 = 1990 .. 2020 by 5;
ampl: set Y2 = 2000 .. 2025 by 5;
ampl: display Y1 union Y2, Y1 inter Y2;
set Y1 union Y2 := 1990 1995 2000 2005 2010 2015 2020 2025;
set Y1 inter Y2 := 2000 2005 2010 2015 2020;

ampl: display Y1 diff Y2, Y1 symdiff Y2;
set Y1 diff Y2 := 1990 1995;
set Y1 symdiff Y2 := 1990 1995 2025;
```

The operands of set operators may be other set expressions, allowing more complex expressions to be built up:

```
ampl: display Y1 symdiff (Y1 symdiff Y2);
set Y1 symdiff (Y1 symdiff Y2) :=
2000   2005   2010   2015   2020   2025;

ampl: display (Y1 union {2025,2035,2045}) diff Y2;
set Y1 union  {2025, 2035, 2045} diff Y2 :=
1990   1995   2035   2045;

ampl: display 2000..2040 by 5 symdiff (Y1 union Y2);
set 2000 .. 2040 by 5 symdiff (Y1 union Y2) :=
2030   2035   2040   1990   1995;
```

The operands must always represent sets, however, so that for example you must write `Y1 union {2025}`, not `Y1 union 2025`.

Set operators group to the left unless parentheses are used to indicate otherwise. The `union`, `diff`, and `symdiff` operators have the same precedence, just below that of `inter`. Thus, for example,

```
A union B inter C diff D
```

is parsed as

```
(A union (B inter C)) diff D
```

A precedence hierarchy of all AMPL operators is given in Table A-1 of Section A.4.

Set operations are often used in the assignment phrase of a set declaration, to define a new set in terms of already declared sets. A simple example is provided by a variation on the diet model of Figure 2-1. Rather than specifying a lower limit and an upper limit on the amount of every nutrient, suppose that you want to specify a set of nutrients that have a lower limit, and a set of nutrients that have an upper limit. (Every nutrient is in one set or the other; some nutrients might be in both.) You could declare:

```
set MINREQ;    # nutrients with minimum requirements
set MAXREQ;    # nutrients with maximum requirements
set NUTR;      # all nutrients (DUBIOUS)
```

But then you would be relying on the user of the model to make sure that `NUTR` contains exactly all the members of `MINREQ` and `MAXREQ`. At best this is unnecessary work, and at worst it will be done incorrectly. Instead you can define `NUTR` as the union:

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

```
set MINREQ;   # nutrients with minimum requirements
set MAXREQ;   # nutrients with maximum requirements

set NUTR = MINREQ union MAXREQ;     # nutrients
set FOOD;                           # foods

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

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

param amt {NUTR,FOOD} >= 0;

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

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

subject to Diet_Min {i in MINREQ}:
   sum {j in FOOD} amt[i,j] * Buy[j] >= n_min[i];

subject to Diet_Max {i in MAXREQ}:
   sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
```

Figure 5-1: Diet model using `union` operator (`dietu.mod`).

All three of these sets are needed, since the nutrient minima and maxima are indexed over `MINREQ` and `MAXREQ`,

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

while the amounts of nutrients in the foods are indexed over `NUTR`:

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

The modification of the rest of the model is straightforward; the result is shown in Figure 5-1.

As a general principle, it is a bad idea to set up a model so that redundant information has to be provided. Instead a minimal necessary collection of sets should be chosen to be supplied in the data, while other relevant sets are defined by expressions in the model.

5.4 Set membership operations and functions

Two other AMPL operators, `in` and `within`, test the membership of sets. As an example, the expression

```
"B2" in NUTR
```

is true if and only if the string `"B2"` is a member of the set `NUTR`. The expression

```
MINREQ within NUTR
```

is true if all members of the set MINREQ are also members of NUTR — that is, if MINREQ is a subset of (or is the same as) NUTR. The in and within operators are the AMPL counterparts of $\in$ and $\subseteq$ in traditional algebraic notation. The distinction between members and sets is especially important here; the left operand of in must be an expression that evaluates to a string or number, whereas the left operand of within must be an expression that evaluates to a set.

AMPL also provides operators not in and not within, which reverse the truth value of their result.

You may apply within directly to a set you are declaring, to say that it must be a subset of some other set. Returning to the diet example, if all nutrients have a minimum requirement, but only some subset of nutrients has a maximum requirement, it would make sense to declare the sets as:

```
set NUTR;
set MAXREQ within NUTR;
```

AMPL will reject the data for this model if any member specified for MAXREQ is not also a member of NUTR.

The built-in function card computes the number of members in (or cardinality of) a set; for example, card(NUTR) is the number of members in NUTR. The argument of the card function may be any expression that evaluates to a set.

5.5 Indexing expressions

In algebraic notation, the use of sets is indicated informally by phrases such as ''for all $i \in P$'' or ''for $t = 1, \ldots, T$'' or ''for all $j \in R$ such that $c_j > 0$.'' The AMPL counterpart is the *indexing expression* that appears within braces { ... } in nearly all of our examples. An indexing expression is used whenever we specify the set over which a model component is indexed, or the set over which a summation runs. Since an indexing expression defines a set, it can be used in any place where a set is appropriate.

The simplest form of indexing expression is just a set name or expression within braces. We have seen this in parameter declarations such as these from the multiperiod production model of Figure 4-4:

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

Later in the model, references to these parameters are subscripted with a single set member, in expressions such as avail[t] and rate[p]. Variables can be declared and used in exactly the same way, except that the keyword var takes the place of param.

The names such as t and i that appear in subscripts and other expressions in our models are examples of *dummy indices* that have been defined by indexing expressions. In fact, any indexing expression may optionally define a dummy index that runs over the specified set. Dummy indices are convenient in specifying bounds on parameters:

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

and on variables:

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

They are also essential in specifying the sets over which constraints are defined, and the sets over which summations are done. We have often seen these uses together, in declarations such as

```
subject to Time {t in 1..T}:
   sum {p in PROD} (1/rate[p]) * Make[p,t] <= avail[t];
```

and

```
subject to Diet_Min {i in MINREQ}:
   sum {j in FOOD} amt[i,j] * Buy[j] >= n_min[i];
```

An indexing expression consists of an index name, the keyword `in`, and a set expression as before. We have been using single letters for our index names, but this is not a requirement; an index name can be any sequence of letters, digits, and underscores that is not a valid number, just like the name for a model component.

Although a name defined by a model component's declaration is known throughout all subsequent statements in the model, the definition of a dummy index name is effective only within the *scope* of the defining indexing expression. Normally the scope is evident from the context. For instance, in the `Diet_Min` declaration above, the scope of `{i in MINREQ}` runs to the end of the statement, so that `i` can be used anywhere in the description of the constraint. On the other hand, the scope of `{j in FOOD}` covers only the summand `amt[i,j] * Buy[j]`. The scope of indexing expressions for sums and other iterated operators is discussed further in Chapter 7.

Once an indexing expression's scope has ended, its dummy index becomes undefined. Thus the same index name can be defined again and again in a model, and in fact it is good practice to use relatively few different index names. A common convention is to associate certain index names with certain sets, so that for example `i` always runs over `NUTR` and `j` always runs over `FOOD`. This is merely a convention, however, not a restriction imposed by AMPL. Indeed, when we modified the diet model so that there was a subset `MINREQ` of `NUTR`, we used `i` to run over `MINREQ` as well as `NUTR`. The opposite situation occurs, for example, if we want to specify a constraint that the amount of each food `j` in the diet is at least some fraction `min_frac[j]` of the total food in the diet:

```
subject to Food_Ratio {j in FOOD}:
   Buy[j] >= min_frac[j] * sum {jj in FOOD} Buy[jj];
```

Since the scope of `j in FOOD` extends to the end of the declaration, a different index `jj` is defined to run over the set `FOOD` in the summation within the constraint.

As a final option, the set in an indexing expression may be followed by a colon (`:`) and a logical condition. The indexing expression then represents only the subset of members that satisfy the condition. For example,

```
{j in FOOD: f_max[j] - f_min[j] < 1}
```

describes the set of all foods whose minimum and maximum amounts are nearly the same, and

```
{i in NUTR: i in MAXREQ or n_min[i] > 0}
```

describes the set of nutrients that are either in MAXREQ or for which n_min is positive. The use of operators such as or and < to form logical conditions will be fully explained in Chapter 7.

By specifying a condition, an indexing expression defines a new set. You can use the indexing expression to represent this set not only in indexed declarations and summations, but anywhere else that a set expression may appear. For example, you could say either of

```
set NUTREQ = {i in NUTR: i in MAXREQ or n_min[i] > 0};
set NUTREQ = MAXREQ union {i in MINREQ: n_min[i] > 0};
```

to define NUTREQ to represent our preceding example of a set expression, and you could use either of

```
set BOTHREQ = {i in MINREQ: i in MAXREQ};
set BOTHREQ = MINREQ inter MAXREQ;
```

to define BOTHREQ to be the set of all nutrients that have both minimum and maximum requirements. It's not unusual to find that there are several ways of describing some complicated set, depending on how you combine set operations and indexing expression conditions. Of course, some possibilities are easier to read than others, so it's worth taking some trouble to find the most readable. In Chapter 6 we also discuss efficiency considerations that sometimes make one alternative preferable to another in specifying compound sets.

In addition to being valuable within the model, indexing expressions are useful in display statements to summarize characteristics of the data or solution. The following example is based on the model of Figure 5-1 and the data of Figure 5-2:

```
ampl: model dietu.mod;
ampl: data dietu.dat;

ampl: display MAXREQ union {i in MINREQ: n_min[i] > 0};
set MAXREQ union {i in MINREQ: n_min[i] > 0}  := A NA CAL C;

ampl: solve;
CPLEX 8.0.0: optimal solution; objective 74.27382022
2 dual simplex iterations (0 in phase I)

ampl: display {j in FOOD: Buy[j] > f_min[j]};
set {j in FOOD: Buy[j] > f_min[j]}  := CHK MTL SPG;

ampl: display {i in MINREQ: Diet_Min[i].slack = 0};
set {i in MINREQ: (Diet_Min[i].slack) == 0}  := C CAL;
```

AMPL interactive commands are allowed to refer to variables and constraints in the condition phrase of an indexing expression, as illustrated by the last two display state-

```
set MINREQ := A B1 B2 C CAL ;
set MAXREQ := A NA CAL ;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:   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  ;

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

param amt (tr):   A    C   B1   B2    NA   CAL :=
          BEEF   60   20   10   15   938   295
          CHK     8    0   20   20  2180   770
          FISH    8   10   15   10   945   440
          HAM    40   40   35   10   278   430
          MCH    15   35   15   15  1182   315
          MTL    70   30   15   15   896   400
          SPG    25   50   25   15  1329   370
          TUR    60   20   15   10  1397   450 ;
```

Figure 5-2: Data for diet model (`dietu.dat`).

ments above. Within a model, however, only sets, parameters and dummy indices may be mentioned in any indexing expression.

The set `BOTHREQ` above might well be empty, in the case where every nutrient has either a minimum or a maximum requirement in the data, but not both. Indexing over an empty set is not an error. When a model component is declared to be indexed over a set that turns out to be empty, AMPL simply skips generating that component. A `sum` over an empty set is zero, and other iterated operators over empty sets have the obvious interpretations (see A.4).

5.6 Ordered sets

Any set of numbers has a natural ordering, so numbers are often used to represent entities, like time periods, whose ordering is essential to the specification of a model. To

describe the difference between this week's inventory and the previous week's inventory, for example, we need the weeks to be ordered so that the ''previous'' week is always well defined.

An AMPL model can also define its own ordering for any set of numbers or strings, by adding the keyword `ordered` or `circular` to the set's declaration. The order in which you give the set's members, in either the model or the data, is then the order in which AMPL works with them. In a set declared `circular`, the first member is considered to follow the last one, and the last to precede the first; in an `ordered` set, the first member has no predecessor and the last member has no successor.

Ordered sets of strings often provide better documentation for a model's data than sets of numbers. Returning to the multiperiod production model of Figure 4-4, we observe that there is no way to tell from the data which weeks the numbers 1 through `T` refer to, or even that they are weeks instead of days or months. Suppose that instead we let the weeks be represented by an ordered set that contains, say, `27sep`, `04oct`, `11oct` and `18oct`. The declaration of `T` is replaced by

```
set WEEKS ordered;
```

and all subsequent occurrences of `1..T` are replaced by `WEEKS`. In the `Balance` constraint, the expression `t-1` is replaced by `prev(t)`, which selects the member before `t` in the set's ordering:

```
subject to Balance {p in PROD, t in WEEKS}:
   Make[p,t] + Inv[p,prev(t)] = Sell[p,t] + Inv[p,t]; # WRONG
```

This is not quite right, however, because when `t` is the first week in `WEEKS`, the member `prev(t)` is not defined. When you try to solve the problem, you will get an error message like this:

```
error processing constraint Balance['bands','27sep']:
        can't compute prev('27sep', WEEKS) --
           '27sep' is the first member
```

One way to fix this is to give a separate balance constraint for the first period, in which `Inv[p,prev(t)]` is replaced by the initial inventory, `inv0[p]`:

```
subject to Balance0 {p in PROD}:
   Make[p,first(WEEKS)] + inv0[p]
      = Sell[p,first(WEEKS)] + Inv[p,first(WEEKS)];
```

The regular balance constraint is limited to the remaining weeks:

```
subject to Balance {p in PROD, t in WEEKS: ord(t) > 1}:
   Make[p,t] + Inv[p,prev(t)] = Sell[p,t] + Inv[p,t];
```

The complete model and data are shown in Figures 5-3 and 5-4. As a tradeoff for more meaningful week names, we have to write a slightly more complicated model.

As our example demonstrates, AMPL provides a variety of functions that apply specifically to ordered sets. These functions are of three basic types.

```
set PROD;           # products
set WEEKS ordered;  # number of weeks

param rate {PROD} > 0;           # tons per hour produced
param inv0 {PROD} >= 0;          # initial inventory
param avail {WEEKS} >= 0;        # hours available in week
param market {PROD,WEEKS} >= 0;  # limit on tons sold in week

param prodcost {PROD} >= 0;      # cost per ton produced
param invcost {PROD} >= 0;       # carrying cost/ton of inventory
param revenue {PROD,WEEKS} >= 0; # revenue/ton sold

var Make {PROD,WEEKS} >= 0;      # tons produced
var Inv {PROD,WEEKS} >= 0;       # tons inventoried
var Sell {p in PROD, t in WEEKS} >= 0, <= market[p,t]; # tons sold

maximize Total_Profit:
   sum {p in PROD, t in WEEKS} (revenue[p,t]*Sell[p,t] -
      prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t]);

         # Objective: total revenue less costs in all weeks

subject to Time {t in WEEKS}:
   sum {p in PROD} (1/rate[p]) * Make[p,t] <= avail[t];

         # Total of hours used by all products
         # may not exceed hours available, in each week

subject to Balance0 {p in PROD}:
   Make[p,first(WEEKS)] + inv0[p]
      = Sell[p,first(WEEKS)] + Inv[p,first(WEEKS)];

subject to Balance {p in PROD, t in WEEKS: ord(t) > 1}:
   Make[p,t] + Inv[p,prev(t)] = Sell[p,t] + Inv[p,t];

         # Tons produced and taken from inventory
         # must equal tons sold and put into inventory
```

Figure 5-3: Production model with ordered sets (`steelT2.mod`).

First, there are functions that return a member from some absolute position in a set. You can write `first(WEEKS)` and `last(WEEKS)` for the first and last members of the ordered set `WEEKS`. To pick out other members, you can use `member(5,WEEKS)`, say, for the 5th member of `WEEKS`. The arguments of these functions must evaluate to an ordered set, except for the first argument of `member`, which can be any expression that evaluates to a positive integer.

A second kind of function returns a member from a position relative to another member. Thus you can write `prev(t,WEEKS)` for the member immediately before `t` in `WEEKS`, and `next(t,WEEKS)` for the member immediately after. More generally, expressions such as `prev(t,WEEKS,5)` and `next(t,WEEKS,3)` refer to the 5th member before and the 3rd member after `t` in `WEEKS`. There are also ‘‘wraparound’’ versions `prevw` and `nextw` that work the same except that they treat the end of the set as wrapping around to the beginning; in effect, they treat all ordered sets as if their decla-

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

param avail :=  27sep 40  04oct 40  11oct 32  18oct 40 ;

param rate :=  bands 200  coils 140 ;
param inv0 :=  bands  10  coils   0 ;

param prodcost :=  bands 10    coils 11 ;
param invcost  :=  bands  2.5  coils  3 ;

param revenue: 27sep   04oct   11oct   18oct :=
      bands       25      26      27      27
      coils       30      35      37      39 ;

param market:  27sep   04oct   11oct   18oct :=
      bands     6000    6000    4000    6500
      coils     4000    2500    3500    4200 ;
```

Figure 5-4: Data for production model (`steelT2.dat`).

rations were `circular`. In all of these functions, the first argument must evaluate to a number or string, the second argument to an ordered set, and the third to an integer. Normally the integer is positive, but zero and negative values are interpreted in a consistent way; for instance, `next(t,WEEKS,0)` is the same as `t`, and `next(t,WEEKS,-5)` is the same as `prev(t,WEEKS,5)`.

Finally, there are functions that return the position of a member within a set. The expression `ord(t,WEEKS)` returns the numerical position of `t` within the set `WEEKS`, or gives you an error message if `t` is not a member of `WEEKS`. The alternative `ord0(t,WEEKS)` is the same except that it returns 0 if `t` is not a member of `WEEKS`. For these functions the first argument must evaluate to a positive integer, and the second to an ordered set.

If the first argument of `next`, `nextw`, `prev`, `prevw`, or `ord` is a dummy index that runs over an ordered set, its associated indexing set is assumed if a set is not given as the second argument. Thus in the constraint

```
subject to Balance {p in PROD, t in WEEKS: ord(t) > 1}:
   Make[p,t] + Inv[p,prev(t)] = Sell[p,t] + Inv[p,t];
```

the functions `ord(t)` and `prev(t)` are interpreted as if they had been written `ord(t,WEEKS)` and `prev(t,WEEKS)`.

Ordered sets can also be used with any of the AMPL operators and functions that apply to sets generally. The result of a `diff` operation preserves the ordering of the left operand, so the material balance constraint in our example could be written:

```
subject to Balance {p in PROD, t in WEEKS diff {first(WEEKS)}}:
   Make[p,t] + Inv[p,prev(t)] = Sell[p,t] + Inv[p,t];
```

For `union`, `inter` and `symdiff`, however, the ordering of the result is not well defined; AMPL treats the result as an unordered set.

For a set that is contained in an ordered set, AMPL provides a way to say that the ordering should be inherited. Suppose for example that you want to try running the multiperiod production model with horizons of different lengths. In the following declarations, the ordered set ALL_WEEKS and the parameter T are given in the data, while the subset WEEKS is defined by an indexing expression to include only the first T weeks:

```
set ALL_WEEKS ordered;
param T > 0 integer;

set WEEKS = {t in ALL_WEEKS: ord(t) <= T} ordered by ALL_WEEKS;
```

We specify `ordered by ALL_WEEKS` so that WEEKS becomes an ordered set, with its members having the same ordering as they do in ALL_WEEKS. The `ordered by` and `circular by` phrases have the same effect as the `within` phrase of Section 5.4 together with `ordered` or `circular`, except that they also cause the declared set to inherit the ordering from the containing set. There are also `ordered by reversed` and `circular by reversed` phrases, which cause the declared set's ordering to be the opposite of the containing set's ordering. All of these phrases may be used either with a subset supplied in the data, or with a subset defined by an expression as in the example above.

Predefined sets and interval expressions

AMPL provides special names and expressions for certain common intervals and other sets that are either infinite or potentially very large. Indexing expressions may not iterate over these sets, but they can be convenient for specifying the conditional phrases in `set` and `param` declarations.

AMPL intervals are sets containing all numbers between two bounds. There are intervals of real (floating-point) numbers and of integers, introduced by the keywords `interval` and `integer` respectively. They may be specified as closed, open, or half-open, following standard mathematical notation,

$$\texttt{interval [a, b]} \equiv \{x: \texttt{a} \le x \le \texttt{b}\},$$
$$\texttt{interval (a, b]} \equiv \{x: \texttt{a} < x \le \texttt{b}\},$$
$$\texttt{interval [a, b)} \equiv \{x: \texttt{a} \le x < \texttt{b}\},$$
$$\texttt{interval (a, b)} \equiv \{x: \texttt{a} < x < \texttt{b}\},$$
$$\texttt{integer [a, b]} \equiv \{x \in I : \texttt{a} \le x \le \texttt{b}\},$$
$$\texttt{integer (a, b]} \equiv \{x \in I : \texttt{a} < x \le \texttt{b}\},$$
$$\texttt{integer [a, b)} \equiv \{x \in I : \texttt{a} \le x < \texttt{b}\},$$
$$\texttt{integer (a, b)} \equiv \{x \in I : \texttt{a} < x < \texttt{b}\}$$

where `a` and `b` are any arithmetic expressions, and I denotes the set of integers. In the declaration phrases

```
in interval
within interval
ordered by [ reversed ] interval
circular by [ reversed ] interval
```

the keyword `interval` may be omitted.

As an example, in declaring Chapter 1's parameter `rate`, you can declare

```
param rate {PROD} in interval (0,maxrate];
```

to say that the production rates have to be greater than zero and not more than some previously defined parameter `maxrate`; you could write the same thing more concisely as

```
param rate {PROD} in (0,maxrate];
```

or equivalently as

```
param rate {PROD} > 0, <= maxrate;
```

An open-ended interval can be specified by using the predefined AMPL parameter `Infinity` as the right-hand bound, or `-Infinity` as the left-hand bound, so that

```
param rate {PROD} in (0,Infinity];
```

means exactly the same thing as

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

in Figure 1-4a. In general, intervals do not let you say anything new in set or parameter declarations; they just give you alternative ways to say things. (They have a more essential role in defining imported functions, as discussed in Section A.22.)

The predefined infinite sets `Reals` and `Integers` are the sets of all floating-point numbers and integers, respectively, in numeric order. The predefined infinite sets `ASCII`, `EBCDIC`, and `Display` all represent the universal set of strings and numbers from which members of any one-dimensional set are drawn. `ASCII` and `EBCDIC` are ordered by the ASCII and EBCDIC collating sequences, respectively. `Display` has the ordering used in AMPL's `display` command (Section A.16): numbers precede literals and are ordered numerically; literals are sorted by the ASCII collating sequence.

As an example, you can declare

```
set PROD ordered by ASCII;
```

to make AMPL's ordering of the members of `PROD` alphabetical, regardless of their ordering in the data. This reordering of the members of `PROD` has no effect on the solutions of the model in Figure 1-4a, but it causes AMPL listings of most entities indexed over `PROD` to appear in the same order (see A.6.2).

Exercises

5-1. (a) Display the sets

```
-5/3 .. 5/3 by 1/3
0 .. 1 by .1
```

Explain any evidence of rounding error in your computer's arithmetic.

(b) Try the following commands from Sections 5.2 and 5.4 on your computer:

```
ampl: set HUGE = 1..1e7;
ampl: display card(HUGE);
```

When AMPL runs out of memory, how many bytes does it say were available? (If your computer really does have enough memory, try `1..1e8`.) Experiment to see how big a set `HUGE` your computer can hold without running out of memory.

5-2. Revise the model of Exercise 1-6 so that it makes use of two different attribute sets: a set of attributes that have lower limits, and a set of attributes that have upper limits. Use the same approach as in Figure 5-1.

5-3. Use the `display` command, together with indexing expressions as demonstrated in Section 5.5, to determine the following sets relating to the diet model of Figures 5-1 and 5-2:

– Foods that have a unit cost greater than $2.00.

– Foods that have a sodium (`NA`) content of more than 1000.

– Foods that contribute more than $10 to the total cost in the optimal solution.

– Foods that are purchased at more than the minimum level but less than the maximum level in the optimal solution.

– Nutrients that the optimal diet supplies in exactly the minimum allowable amount.

– Nutrients that the optimal diet supplies in exactly the maximum allowable amount.

– Nutrients that the optimal diet supplies in more than the minimum allowable amount but less than the maximum allowable amount.

5-4. This exercise refers to the multiperiod production model of Figure 4-4.

(a) Suppose that we define two additional scalar parameters,

```
param Tbegin integer >= 1;
param Tend integer > Tbegin, <= T;
```

We want to solve the linear program that covers only the weeks from `Tbegin` through `Tend`. We still want the parameters to use the indexing `1..T`, however, so that we don't need to change the data tables every time we try a different value for `Tbegin` or `Tend`.

To start with, we can change every occurrence of `1..T` in the variable, objective and constraint declarations to `Tbegin..Tend`. By making these and other necessary changes, create a model that correctly covers only the desired weeks.

(b) Now suppose that we define a different scalar parameter,

```
param Tagg integer >= 1;
```

We want to ''aggregate'' the model, so that one ''period'' in our LP becomes `Tagg` weeks long, rather than one week. This would be appropriate if we have, say, a year of weekly data, which would yield an LP too large to be convenient for analysis.

To aggregate properly, we must define the availability of hours in each period to be the sum of the availabilities in all weeks of the period:

```
param avail_agg {t in 1..T by Tagg}
   = sum {u in t..t+Tagg-1} avail[u];
```

The parameters `market` and `revenue` must be similarly summed. Make all the necessary changes to the model of Figure 4-4 so that the resulting LP is properly aggregated.

(c) Re-do the models of (a) and (b) to use an ordered set of strings for the periods, as in Figure 5-3.

5-5. Extend the transportation model of Figure 3-1a to a multiperiod version, in which the periods are months represented by an ordered set of character strings such as `"Jan"`, `"Feb"` and so forth. Use inventories at the origins to link the periods.

5-6. Modify the model of Figure 5-3 to merge the `Balance0` and `Balance` constraints, as in Figure 4-4. Hint: `0..T` and `1..T` are analogous to

```
set WEEKS0 ordered;
set WEEKS = {i in WEEKS0: ord(i) > 1} ordered by WEEKS0;
```