Copyright © 2003 by Robert Fourer, David M. Gay and Brian W. Kernighan

7

Parameters and Expressions

A large optimization model invariably uses many numerical values. As we have explained before, only a concise symbolic description of these values need appear in an AMPL model, while the explicit data values are given in separate data statements, to be described in Chapter 9.

In AMPL a single named numerical value is called a *parameter*. Although some parameters are defined as individual scalar values, most occur in vectors or matrices or other collections of numerical values indexed over sets. We will thus loosely refer to an indexed collection of parameters as ''a parameter'' when the meaning is clear. To begin this chapter, Section 7.1 describes the rules for declaring parameters and for referring to them in an AMPL model.

Parameters and other numerical values are the building blocks of the expressions that make up a model's objective and constraints. Sections 7.2 and 7.3 describe arithmetic expressions, which have a numerical value, and logical expressions, which evaluate to true or false. Along with the standard unary and binary operators of conventional algebraic notation, AMPL provides iterated operators like `sum` and `prod`, and a conditional (`if-then-else`) operator that chooses between two expressions, depending on the truth of a third expression.

The expressions in objectives and constraints necessarily involve variables, whose declaration and use will be discussed in Chapter 8. There are several common uses for expressions that involve only sets and parameters, however. Section 7.4 describes how logical expressions are used to test the validity of data, either directly in a parameter declaration, or separately in a `check` statement. Section 7.5 introduces features for defining new parameters through arithmetic expressions in previously declared parameters and sets, and 7.6 describes randomly-generated parameters.

Although the key purpose of parameters is to represent numerical values, they can also represent logical values or arbitrary strings. These possibilities are covered in Sections 7.7 and 7.8, respectively. AMPL provides a range of operators for strings, but as they are most often used in AMPL commands and programming rather than in models, we defer their introduction to Section 13.7.

7.1 Parameter declarations

A parameter declaration describes certain data required by a model, and indicates how the model will refer to data values in subsequent expressions.

The simplest parameter declaration consists of the keyword `param` and a name:

```
param T;
```

At any point after this declaration, `T` can be used to refer to a numerical value.

More often, the name in a parameter declaration is followed by an indexing expression:

```
param avail {1..T};
param demand {DEST,PROD};
param revenue {p in PROD, AREA[p], 1..T};
```

One parameter is defined for each member of the set specified by the indexing expression. Thus a parameter is uniquely determined by its name and its associated set member; throughout the rest of the model, you would refer to this parameter by writing the name and bracketed ''subscripts'':

```
avail[i]
demand[j,p]
revenue[p,a,t]
```

If the indexing is over a simple set of objects as described in Chapter 5, there is one subscript. If the indexing is over a set of pairs, triples, or longer tuples as described in Chapter 6, there must be a corresponding pair, triple, or longer list of subscripts separated by commas. The subscripts can be any expressions, so long as they evaluate to members of the underlying index set.

An unindexed parameter is a scalar value, but a parameter indexed over a simple set has the characteristics of a vector or an array; when the indexing is over a sequence of integers, say

```
param avail {1..T};
```

the individual subscripted parameters are `avail[1]`, `avail[2]`, ..., `avail[T]`, and there is an obvious analogy to the vectors of linear algebra or the arrays of a programming language like Fortran or C. AMPL's concept of a vector is more general, however, since parameters may also be indexed over sets of strings, which need not even be ordered. Indexing over sets of strings is best suited for parameters that correspond to places, products and other entities for which no numbering is especially natural. Indexing over sequences of numbers is more appropriate for parameters that correspond to weeks, stages, and the like, which by their nature tend to be ordered and numbered; even for these, you may prefer to use ordered sets of strings as described in Section 5.6.

A parameter indexed over a set of pairs is like a two-dimensional array or matrix. If the indexing is over all pairs from two sets, as in

```
set ORIG;
set DEST;
param cost {ORIG,DEST};
```

then there is a parameter `cost[i,j]` for every combination of `i` from `ORIG` and `j` from `DEST`, and the analogy to a matrix is strongest — although again the subscripts are more likely to be strings than numbers. If the indexing is over a subset of pairs, however:

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

then `cost[i,j]` exists only for those `i` from `ORIG` and `j` from `DEST` such that `(i,j)` is a member of `LINKS`. In this case, you can think of `cost` as being a ''sparse'' matrix.

Similar comments apply to parameters indexed over triples and longer tuples, which resemble arrays of higher dimension in programming languages.

7.2 Arithmetic expressions

Arithmetic expressions in AMPL are much the same as in other computer languages. Literal numbers consist of an optional sign preceding a sequence of digits, which may or may not include a decimal point (for example, `-17` or `2.71828` or `+.3`). At the end of a literal there may also be an exponent, consisting of the letter `d`, `D`, `e`, or `E` and an optional sign followed by digits (`1e30` or `7.66439D-07`).

Literals, parameters, and variables are combined into expressions by the standard operations of addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`^`). The familiar conventions of arithmetic apply. Exponentiation has higher precedence than multiplication and division, which have higher precedence than addition and subtraction; successive operations of the same precedence group to the left, except for exponentiation, which groups to the right. Parentheses may be used to change the order of evaluation.

Arithmetic expressions may also use the `div` operator, which returns the truncated quotient when its left operand is divided by its right operand; the `mod` operator, which computes the remainder; and the `less` operator, which returns its left operand minus its right operand if the result is positive, or zero otherwise. For purposes of precedence and grouping, AMPL treats `div` and `mod` like division, and `less` like subtraction.

A list of arithmetic operators (and logical operators, to be described shortly) is given in Table 7-1. As much as possible, AMPL follows common programming languages in its choice of operator symbols, such as `*` for multiplication and `/` for division. There is sometimes more than one standard, however, as with exponentiation, where some languages use `^` while others use `**`. In this and other cases, AMPL provides alternate forms. Table 7-1 shows the more common forms to the left, and the alternatives (if any)

<table>
<tr><th>Usual style</th><th>alternative style</th><th>type of operands</th><th>type of result</th></tr>
<tr><td>if-then-else</td><td></td><td>logical, arithmetic</td><td>arithmetic</td></tr>
<tr><td>or</td><td>||</td><td>logical</td><td>logical</td></tr>
<tr><td>exists forall</td><td></td><td>logical</td><td>logical</td></tr>
<tr><td>and</td><td>&&</td><td>logical</td><td>logical</td></tr>
<tr><td>not (unary)</td><td>!</td><td>logical</td><td>logical</td></tr>
<tr><td>< <= = <> > >=</td><td>< <= == != > >=</td><td>arithmetic</td><td>logical</td></tr>
<tr><td>in not in</td><td></td><td>object, set</td><td>logical</td></tr>
<tr><td>+ - less</td><td></td><td>arithmetic</td><td>arithmetic</td></tr>
<tr><td>sum prod min max</td><td></td><td>arithmetic</td><td>arithmetic</td></tr>
<tr><td>* / div mod</td><td></td><td>arithmetic</td><td>arithmetic</td></tr>
<tr><td>+ - (unary)</td><td></td><td>arithmetic</td><td>arithmetic</td></tr>
<tr><td>^</td><td>**</td><td>arithmetic</td><td>arithmetic</td></tr>
</table>

Exponentiation and if-then-else are right-associative; the other operators are left-associative. The logical operand of if-then-else appears after if, and the arithmetic operands after then and (optionally) else.

Table 7-1: Arithmetic and logical operators, in increasing precedence.

to the right; you can mix them as you like, but your models will be easier to read and understand if you are consistent in your choices.

Another way to build arithmetic expressions is by applying functions to other expressions. A function reference consists of a name followed by a parenthesized argument or comma-separated list of arguments; an arithmetic argument can be any arithmetic expression. Here are a few examples, which compute the minimum, absolute value, and square root of their arguments, respectively:

```
min(T,20)
abs(sum {i in ORIG} supply[i] - sum {j in DEST} demand[j])
sqrt((tan[j]-tan[k])^2)
```

Table 7-2 lists the built-in arithmetic functions that are typically found in models. Except for min and max, the names of any of these functions may be redefined, but their original meanings will become inaccessible. For example, a model may declare a parameter named tan as in the last example above, but then it cannot also refer to the function tan.

The set functions card and ord, which were described in Chapter 5, also produce an arithmetic result. In addition, AMPL provides several ‘‘rounding’’ functions (Section 11.3) and a variety of random-number functions (Section 7.6 below). A mechanism for ‘‘importing’’ functions defined by your own programs is described in Appendix A.22.

`abs(`x`)`	absolute value, $\lvert x \rvert$
`acos(`x`)`	inverse cosine, $\cos^{-1}(x)$
`acosh(`x`)`	inverse hyperbolic cosine, $\cosh^{-1}(x)$
`asin(`x`)`	inverse sine, $\sin^{-1}(x)$
`asinh(`x`)`	inverse hyperbolic sine, $\sinh^{-1}(x)$
`atan(`x`)`	inverse tangent, $\tan^{-1}(x)$
`atan2(`y, x`)`	inverse tangent, $\tan^{-1}(y/x)$
`atanh(`x`)`	inverse hyperbolic tangent, $\tanh^{-1}(x)$
`cos(`x`)`	cosine
`cosh(`x`)`	hyperbolic cosine
`exp(`x`)`	exponential, e^x
`log(`x`)`	natural logarithm, $\log_e(x)$
`log10(`x`)`	common logarithm, $\log_{10}(x)$
`max(`$x, y, ...$`)`	maximum (2 or more arguments)
`min(`$x, y, ...$`)`	minimum (2 or more arguments)
`sin(`x`)`	sine
`sinh(`x`)`	hyperbolic sine
`sqrt(`x`)`	square root
`tan(`x`)`	tangent
`tanh(`x`)`	hyperbolic tangent

Table 7-2: Built-in arithmetic functions for use in models.

Finally, the indexed operators such as Σ and Π from algebraic notation are generalized in AMPL by expressions for iterating operations over sets. In particular, most large-scale linear programming models contain iterated summations:

```
sum {i in ORIG} supply[i]
```

The keyword `sum` may be followed by any indexing expression. The subsequent arithmetic expression is evaluated once for each member of the index set, and all the resulting values are added. Thus the sum above, from the transportation model of Figure 3-1a, represents the total supply available, at all origins. The `sum` operator has lower precedence than `*`, so the objective of the same model can be written

```
sum {i in ORIG, j in DEST} cost[i,j] * Trans[i,j]
```

to represent the total of `cost[i,j] * Trans[i,j]` over all combinations of origins and destinations. The precedence of `sum` is higher than that of `+` or `-`, however, so for the objective of the multiperiod production model in Figure 6-3 we must write

```
sum {p in PROD, t in 1..T}
   (sum {a in AREA[p]} revenue[p,a,t]*Sell[p,a,t] -
      prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t]);
```

The outer sum applies to the entire parenthesized expression following it, while the inner sum applies only to the term `revenue[p,a,t] * Sell[p,a,t]`.

Other iterated arithmetic operators are prod for multiplication, min for minimum, and max for maximum. As an example, we could use

```
max {i in ORIG} supply[i]
```

to describe the greatest supply available at any origin.

Bear in mind that, while an AMPL arithmetic function or operator may be applied to variables as well as to parameters or to numeric members of sets, most operations on variables are not linear. AMPL's requirements for arithmetic expressions in a linear program are described in Section 8.2. Some of the nonlinear functions of variables that can be handled by certain solvers are discussed in Chapter 18.

7.3 Logical and conditional expressions

The values of arithmetic expressions can be tested against each other by comparison operators:

=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

The result of a comparison is either ''true'' or ''false''. Thus `T > 1` is true if the parameter T has a value greater than 1, and is false otherwise; and

```
sum {i in ORIG} supply[i] = sum {j in DEST} demand[j]
```

is true if and only if total supply equals total demand.

Comparisons are one example of AMPL's logical expressions, which evaluate to true or false. Set membership tests using in and within, described in Section 5.4, are another example. More complex logical expressions can be built up with logical operators. The and operator returns true if and only if both its operands are true, while or returns true if and only if at least one of its operands is true; the unary operator not returns false for true and true for false. Thus the expression

```
T >= 0 and T <= 10
```

is only true if T lies in the interval $[0, 10]$, while the following from Section 5.5,

```
i in MAXREQ or n_min[i] > 0
```

is true if i is a member of MAXREQ, or n_min[i] is positive, or both. Where several operators are used together, any comparison, membership or arithmetic operator has higher precedence than the logical operators; and has higher precedence than or, while not has higher precedence than either. Thus the expression

```
not i in MAXREQ or n_min[i] > 0 and n_min[i] <= 10
```

is interpreted as

```
(not (i in MAXREQ)) or ((n_min[i] > 0) and (n_min[i] <= 10))
```

Alternatively, the `not in` operator could be used:

```
i not in MAXREQ or n_min[i] > 0 and n_min[i] <= 10
```

The precedences are summarized in Table 7-1, which also gives alternative forms.

Like `+` and `*`, the operators `or` and `and` have iterated versions. The iterated `or` is denoted by `exists`, and the iterated `and` by `forall`. For example, the expression

```
exists {i in ORIG} demand[i] > 10
```

is true if and only if at least one origin has a demand greater than 10, while

```
forall {i in ORIG} demand[i] > 10
```

is true if and only if every origin has demand greater than 10.

Another use for a logical expression is as an operand to the conditional or `if-then-else` operator, which returns one of two different arithmetic values depending on whether the logical expression is true or false. Consider the two collections of inventory balance constraints in the multiperiod production model of Figure 5-3:

```
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];
```

The `Balance0` constraints are basically the `Balance` constraints with `t` set to `first(WEEKS)`. The only difference is in the second term, which represents the previous week's inventory; it is given as `inv0[p]` for the first week (in the `Balance0` constraints) but is represented by the variable `Inv[p,prev(t)]` for subsequent weeks (in the `Balance` constraints). We would like to combine these constraints into one declaration, by having a term that takes the value `inv0[p]` when `t` is the first week, and takes the value `Inv[p,prev(t)]` otherwise. Such a term is written in AMPL as:

```
if t = first(WEEKS) then inv0[p] else Inv[p,prev(t)]
```

Placing this expression into the constraint declaration, we can write

```
subject to Balance {p in PROD, t in WEEKS}:
   Make[p,t] +
      (if t = first(WEEKS) then inv0[p] else Inv[p,prev(t)])
         = Sell[p,t] + Inv[p,t];
```

This form communicates the inventory balance constraints more concisely and directly than two separate declarations.

The general form of a conditional expression is

`if` *a* `then` *b* `else` *c*

where *a* is a logical expression. If *a* evaluates to true, the conditional expression takes the value of *b*; if *a* is false, the expression takes the value of *c*. If *c* is zero, the `else` *c* part can be dropped. Most often *b* and *c* are arithmetic expressions, but they can also be string or set expressions, so long as both are expressions of the same kind. Because `then` and `else` have lower precedence than any other operators, a conditional expression needs to be parenthesized (as in the example above) unless it occurs at the end of a statement.

AMPL also has an `if-then-else` for use in programming; like the conditional statements in many programming languages, it executes one or another block of statements depending on the truth of some logical expression. We describe it with other AMPL programming features in Chapter 13. The `if-then-else` that we have described here is not a statement, but rather an expression whose value is conditionally determined. It therefore belongs inside a declaration, in a place where an expression would normally be evaluated.

7.4 Restrictions on parameters

If `T` is intended to represent the number of weeks in a multiperiod model, it should be an integer and greater than 1. By including these conditions in `T`'s declaration,

```
param T > 1 integer;
```

you instruct AMPL to reject your data if you inadvertently set `T` to 1:

```
error processing param T:
        failed check: param T = 1
                is not > 1;
```

or to 2.5:

```
error processing param T:
        failed check: param T = 2.5
                is not an integer;
```

AMPL will not send your problem instance to a solver as long as any errors of this kind remain.

In the declaration of an indexed collection of parameters, a simple restriction such as `integer` or `>= 0` applies to every parameter defined. Our examples often use this option to specify that vectors and arrays are nonnegative:

```
param demand {DEST,PROD} >= 0;
```

If you include dummy indices in the indexing expression, however, you can use them to specify a different restriction for each parameter:

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

The effect of these declarations is to define a pair of parameters `f_max[j] >= f_min[j]` for every `j` in the set `FOOD`.

A restriction phrase for a parameter declaration may be the word `integer` or `binary` or a comparison operator followed by an arithmetic expression. While `integer` restricts a parameter to integral (whole-number) values, `binary` restricts it to zero or one. The arithmetic expression may refer to sets and parameters previously defined in the model, and to dummy indices defined by the current declaration. There may be several restriction phrases in the same declaration, in which case they may optionally be separated by commas.

In special circumstances, a restriction phrase may even refer to the parameter in whose declaration it appears. Some multiperiod production models, for example, are defined in terms of a parameter `cumulative_market[p,t]` that represents the cumulative demand for product `p` in weeks 1 through `t`. Since cumulative demand does not decrease, you might try to write a restriction phrase like this:

```
param cumulative_market {p in PROD, t in 1..T}
   >= cumulative_market[p,t-1];   # ERROR
```

For the parameters `cumulative_market[p,1]`, however, the restriction phrase will refer to `cumulative_market[p,0]`, which is undefined; AMPL will reject the declaration with an error message. What you need here again is a conditional expression that handles the first period specially:

```
param cumulative_market {p in PROD, t in 1..T}
   >= if t = 1 then 0 else cumulative_market[p,t-1];
```

The same thing could be written a little more compactly as

```
param cumulative_market {p in PROD, t in 1..T}
   >= if t > 1 then cumulative_market[p,t-1];
```

since ‘‘`else 0`’’ is assumed. Almost always, some form of `if-then-else` expression is needed to make this kind of self-reference possible.

As you might suspect from this last example, sometimes it is desirable to place a more complex restriction on the model's data than can be expressed by a restriction phrase within a declaration. This is the purpose of the `check` statement. For example, in the transportation model of Figure 3-1a, total supply must equal total demand:

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

The multicommodity version, in Figure 4-1, uses an indexed `check` to say that total supply must equal total demand for each product:

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

Here the restriction is tested once for each member `p` of `PROD`. If the check fails for any member, AMPL prints an error message and rejects all of the data.

You can think of the `check` statement as specifying a kind of constraint, but only on the data. The restriction clause is a logical expression, which may use any previously

defined sets and parameters as well as dummy indices defined in the statement's indexing expression. After the data values have been read, the logical expression must evaluate to true; if an indexing expression has been specified, the logical expression is evaluated separately for each assignment of set members to the dummy indices, and must be true for each.

We strongly recommend the use of restriction phrases and `check` statements to validate a model's data. These features will help you to catch data errors at an early stage, when they are easy to fix. Data errors not caught will, at best, cause errors in the generation of the variables and constraints, so that you will get some kind of error message from AMPL. In other cases, data errors lead to the generation of an incorrect linear program. If you are fortunate, the incorrect LP will have a meaningless optimal solution, so that — possibly after a good deal of effort — you will be able to work backward to find the error in the data. At worst, the incorrect LP will have a plausible solution, and the error will go undetected.

7.5 Computed parameters

It is seldom possible to arrange that the data values available to a model are precisely the coefficient values required by the objective and constraints. Even in the simple production model of Figure 1-4, for example, we wrote the constraint as

```
sum {p in PROD} (1/rate[p]) * Make[p] <= avail;
```

because production rates were given in tons per hour, while the coefficient of `Make[p]` had to be in hours per ton. Any parameter expression may be used in the constraints and objective, but the expressions are best kept simple. When more complex expressions are needed, the model is usually easier to understand if new, computed parameters are defined in terms of the data parameters.

The declaration of a computed parameter has an assignment phrase, which resembles the restriction phrase described in the previous section except for the use of an = operator to indicate that the parameter is being set equal to a certain expression, rather than merely being restricted by an inequality. As a first example, suppose that the data values provided to the multicommodity transportation model of Figure 4-1 consist of the total demand for each product, together with each destination's share of demand. The destinations' shares are percentages between zero and 100, but their sum over all destinations might not exactly equal 100%, because of rounding and approximation. Thus we declare data parameters to represent the shares, and a computed parameter equal to their sum:

```
param share {DEST} >= 0, <= 100;
param tot_sh = sum {j in DEST} share[j];
```

We can then declare a data parameter to represent total demands, and a computed parameter that equals demand at each destination:

```
param tot_dem {PROD} >= 0;
param demand {j in DEST, p in PROD}
        = share[j] * tot_dem[p] / tot_sh;
```

The division by tot_sh acts as a correction factor for a sum not equal to 100%. Once demand has been defined in this way, the model can use it as in Figure 4-1:

```
subject to Demand {j in DEST, p in PROD}:
   sum {i in ORIG} Trans[i,j,p] = demand[j,p];
```

We could avoid computed parameters by substituting the formulas for tot_sh and demand[j,p] directly into this constraint:

```
subject to Demand {j in DEST, p in PROD}:
   sum {i in ORIG} Trans[i,j,p]
      = share[j] * tot_dem[p] / sum {k in DEST} share[k];
```

This alternative makes the model a little shorter, but the computation of the demand and the structure of the constraint are both harder to follow.

As another example, consider a scenario for the multiperiod production model (Figure 4-4) in which minimum inventories are computed. Specifically, suppose that the inventory of product p for week t must be at least a certain fraction of market[p,t+1], the maximum that can be sold in the following week. We thus use the following declarations for the data to be supplied:

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

and then declare

```
param mininv {p in PROD, t in 0..T} = frac * market[p,t+1];
var Inv {p in PROD, t in 0..T} >= mininv[p,t];
```

to define and use parameters mininv[p,t] that represent the minimum inventory of product p for week t. AMPL keeps all = definitions of parameters up to date throughout a session. Thus for example if you change the value of frac the values of all the mininv parameters automatically change accordingly.

If you define a computed parameter as in the examples above, then you cannot also specify a data value for it. An attempt to do so will result in an error message:

```
mininv was defined in the model
context:  param  >>> mininv <<<  :=  bands 2  3000
```

However, there is an alternative way in which you can define an initial value for a parameter but allow it to be changed later.

If you define a parameter using the default operator in place of =, then the parameter is initialized rather than defined. Its value is taken from the value of the expression to the right of the default operator, but does not change if the expression's value later changes. Initial values can be overridden by data statements, and they also may be changed by subsequent assignment statements. This feature is most useful for writing AMPL scripts that update certain values repeatedly, as shown in Section 13.2.

If you define a parameter using the operator `default` in place of =, then you can specify values in data statements to override the ones that would otherwise be computed. For instance, by declaring

```
param mininv {p in PROD, t in 0..T}
   default frac * market[p,t+1];
```

you can allow a few exceptional minimum inventories to be specified as part of the data for the model, either in a list:

```
param mininv :=
       bands 2  3000
       coils 2  2000
       coils 3  2000 ;
```

or in a table:

```
param market:      1     2     3     4 :=
       bands       .  3000     .     .
       coils       .  2000  2000     . ;
```

(AMPL uses ‘‘`.`’’ in a data statement to indicate an omitted entry, as explained in Chapter 9 and A.12.2.)

The expression that gives the default value of a parameter is evaluated only when the parameter’s value is first needed, such as when an objective or constraint that uses the parameter is processed by a `solve` command.

In most = and `default` phrases, the operator is followed by an arithmetic expression in previously defined sets and parameters (but not variables) and currently defined dummy indices. Some parameters in an indexed collection may be given a computed or default value in terms of others in the same collection, however. As an example, you can smooth out some of the variation in the minimum inventories by defining the `mininv` parameter to be a running average like this:

```
param mininv {p in PROD, t in 0..T} =
   if t = 0 then inv0[p]
      else 0.5 * (mininv[p,t-1] + frac * market[p,t+1]);
```

The values of `mininv` for week 0 are set explicitly to the initial inventories, while the values for each subsequent week `t` are defined in terms of the previous week’s values. AMPL permits any ‘‘recursive’’ definition of this kind, but will signal an error if it detects a circular reference that causes a parameter’s value to depend directly or indirectly on itself.

You can use the phrases defined in this section together with the restriction phrases of the previous section, to further check the values that are computed. For example the declaration

```
param mininv {p in PROD, t in 0..T}
   = frac * market[p,t+1], >= 0;
```

will cause an error to be signaled if the computed value of any of the mininv parameters is negative. This check is triggered whenever an AMPL session uses mininv for any purpose.

7.6 Randomly generated parameters

When you're testing out a model, especially in the early stages of development, you may find it convenient to let randomly generated data stand in for actual data to be obtained later. Randomly generated parameters can also be useful in experimenting with alternative model formulations or solvers.

Randomly generated parameters are like the computed parameters introduced in the preceding section, except that their defining expressions are made random by use of AMPL's built-in random number generation functions listed in Table A-3. As an example of the simplest case, the individual parameter avail representing hours available in steel.mod may be defined to equal a random function:

```
param avail_mean > 0;
param avail_variance > 0, < avail_mean / 2;

param avail = max(Normal(avail_mean, avail_variance), 0);
```

Adding some indexing gives a multi-stage version of this model:

```
param avail {STAGE} =
   max(Normal(avail_mean, avail_variance), 0);
```

For each stage s, this gives avail[s] a different random value from the same random distribution. To specify stage-dependent random distributions, you would add indexing to the mean and variance parameters as well:

```
param avail_mean {STAGE} > 0;
param avail_variance {s in STAGE} > 0, < avail_mean[s] / 2;

param avail {s in STAGE} =
   max(Normal(avail_mean[s], avail_variance[s]), 0);
```

The max(...,0) expression is included to handle the rare case in which the normal distribution with a positive mean returns a negative value.

More general ways of randomly computing parameters arise naturally from the preceding section's examples. In the multicommodity transportation problem, you can define random shares of demand:

```
param share {DEST} = Uniform(0,100);
param tot_sh = sum {j in DEST} share[j];

param tot_dem {PROD} >= 0;
param demand {j in DEST, p in PROD}
       = share[j] * tot_dem[p] / tot_sh;
```

Parameters tot_sh and demand then also become random, because they are defined in terms of random parameters. In the multiperiod production model, you can define the demand quantities market[p,t] in terms of an initial value and a random amount of increase per period:

```
param market1 {PROD} >= 0;
param max_incr {PROD} >= 0;

param market {p in PROD, t in 1..T+1} =
   if t = 1 then market1[p]
      else Uniform(0,max_incr) * market[p,t-1];
```

A recursive definition of this kind provides a way of generating simple random processes over time.

All of the AMPL random functions are based on a uniform random number generator with a very long period. When you start AMPL or give a reset command, however, the generator is reset and the ‘‘random’’ values are the same as before. You can request different values by changing the AMPL option randseed to some integer other than its default value of 1; the command for this purpose is

```
option randseed n;
```

where n is some integer value. Nonzero values give sequences that repeat each time AMPL is reset. A value of 0 requests AMPL to pick a seed based on the current value of the system clock, resulting (for practical purposes) in a different seed at each reset.

AMPL’s reset data command, when applied to a randomly computed parameter, also causes a new sample of random values to be determined. The use of this command is discussed in Section 11.3.

7.7 Logical parameters

Although parameters normally represent numeric values, they can optionally be used to stand for true-false values or for character strings.

The current version of AMPL does not support a full-fledged ‘‘logical’’ type of parameter that would stand for only the values true and false, but a parameter of type binary may be used to the same effect. As an illustration, we describe an application of the preceding inventory example to consumer goods. Certain products in each week may be specially promoted, in which case they require a higher inventory fraction. Using parameters of type binary, we can represent this situation by the following declarations:

```
param fr_reg > 0;         # regular inventory fraction
param fr_pro > fr_reg;    # fraction for promoted items

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

The binary parameters `promote[p,t]` are 0 when there is no promotion, and 1 when there is a promotion. Thus we can define the minimum-inventory parameters by use of an `if-then-else` expression as follows:

```
param mininv {p in PROD, t in 0..T} =
   (if promote[p,t] = 1 then fr_pro else fr_reg)
      * market[p,t+1];
```

We can also say the same thing more concisely:

```
param mininv {p in PROD, t in 0..T} =
   (if promote[p,t] then fr_pro else fr_reg) * market[p,t+1];
```

When an arithmetic expression like `promote[p,t]` appears where a logical expression is required, AMPL interprets any nonzero value as true, and zero as false. You do need to exercise a little caution to avoid being tripped up by this implicit conversion. For example, in Section 7.4 we used the expression

```
if t = 1 then 0 else cumulative_market[p,t-1]
```

If you accidentally write

```
if t then 0 else cumulative_market[p,t-1]     # DIFFERENT
```

it's perfectly legal, but it doesn't mean what you intended.

7.8 Symbolic parameters

You may permit a parameter to represent character string values, by including the keyword `symbolic` in its declaration. A symbolic parameter's values may be strings or numbers, just like a set's members, but the string values may not participate in arithmetic.

A major use of symbolic parameters is to designate individual set members that are to be treated specially. For example, in a model of traffic flow, there is a set of intersections, two of whose members are designated as the entrance and exit. Symbolic parameters can be used to represent these two members:

```
set INTER;

param entr symbolic in INTER;
param exit symbolic in INTER, <> entr;
```

In the data statements, an appropriate string is assigned to each symbolic parameter:

```
set INTER := a b c d e f g ;

param entr := a ;
param exit := g ;
```

These parameters are subsequently used in defining the objective and constraints; the complete model is developed in Section 15.2.

Another use of symbolic parameters is to associate descriptive strings with set members. Consider for example the set of ‘‘origins’’ in the transportation model of Figure 3-1a. When we introduced this set at the beginning of Chapter 3, we described each originating city by means of a 4-character string and a longer descriptive string. The short strings became the members of the AMPL set `ORIG`, while the longer strings played no further role. To make both available, we could declare

```
set ORIG;
param orig_name {ORIG} symbolic;
param supply {ORIG} >= 0;
```

Then in the data we could specify

```
param: ORIG:   orig_name                       supply :=
        GARY   "Gary, Indiana"                  1400
        CLEV   "Cleveland, Ohio"                2600
        PITT   "Pittsburgh, Pennsylvania"       2900 ;
```

Since the long strings do not have the form of AMPL names, they do need to be quoted. They still play no role in the model or the resulting linear program, but they can be retrieved for documentary purposes by the `display` and `printf` commands described in Chapter 12.

Just as there are arithmetic and logical operators and functions, there are AMPL string operators and functions for working with string values. These features are mostly used in AMPL command scripts rather than in models, so we defer their description to Section 13.7.

Exercises

7-1. Show how the multicommodity transportation model of Figure 4-1 could be modified so that it applies the following restrictions to the data. Use either a restriction phrase in a `set` or `param` declaration, or a `check` statement, whichever is appropriate.

- No city is a member of both `ORIG` and `DEST`.
- The number of cities in `DEST` must be greater than the number in `ORIG`.
- Demand does not exceed 1000 at any one city in `DEST`.
- Total supply for each product at all origins must equal total demand for that product at all destinations.
- Total supply for all products at all origins must equal total demand for all products at all destinations.
- Total supply of all products at an origin must not exceed total capacity for all shipments from that origin.
- Total demand for all products at a destination must not exceed total capacity for all shipments to that destination.

7-2. Show how the multiperiod production model of Figure 4-4 could be modified so that it applies the following restrictions to the data.

– The number of weeks is a positive integer greater than 1.
– The initial inventory of a product does not exceed the total market demand for that product over all weeks.
– The inventory cost for a product is never more than 10% of the expected revenue for that product in any one week.
– The number of hours in a week is between 24 and 40, and does not change by more than 8 hours from one week to the next.
– For each product, the expected revenue never decreases from one week to the next.

7-3. The solutions to the following exercises involve the use of an `if-then-else` operator to formulate a constraint.

(a) In the example of the constraint `Balance` in Section 7.3, we used an expression beginning

```
if t = first(WEEKS) then ...
```

Find an equivalent expression that uses the function `ord(t)`.

(b) Combine the `Diet_Min` and `Diet_Max` constraints of Figure 5-1's diet model into one constraint declaration.

(c) In the multicommodity transportation model of Figure 4-1, imagine that there is more demand at the destinations than we can meet from the supply produced at the origins. To make up the difference, a limited number of additional tons can be purchased (rather than manufactured) for shipment at certain origins.

To model this situation, suppose that we declare a subset of origins,

```
set BUY_ORIG within ORIG;
```

where the additional tons can be bought. The relevant data values and decision variables could be indexed over this subset:

```
param buy_supply {BUY_ORIG,PROD} >= 0;  # available for purchase
param buy_cost {BUY_ORIG,PROD} > 0;     # purchase cost per ton

var Buy {i in BUY_ORIG, p in PROD} >= 0, <= buy_supply[i,p];
                                        # amount to buy
```

Revise the objective function to include the purchase costs. Revise the `Supply` constraints to say that, for each origin and each product, total tons shipped out must equal tons of supply from production plus (if applicable) tons purchased.

(d) Formulate the same model as in (c), but with `BUY_ORIG` being the set of pairs `(i,p)` such that product `p` can be bought at origin `i`.

7-4. This exercise is concerned with the following sets and parameters from Figure 4-1:

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

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

(a) Write `param` declarations, using the = operator, to compute parameters having the following definitions:

– `prod_supply[p]` is the total supply of product `p` at all origins.

– dest_demand[j] is the total demand for all products at destination j.
– true_limit[i,j,p] is the largest quantity of product p that can be shipped from i to j — that is, the largest value that does not exceed limit[i,j], or the supply of p at i, or the demand for p at j.
– max_supply[p] is the largest supply of product p available at any origin.
– max_diff[p] is the largest difference, over all combinations of origins and destinations, between the supply and demand for product p.

(b) Write set declarations, using the = operator, to compute these sets:

– Products p whose demand is at least 500 at some destination j.
– Products p whose demand is at least 250 at all destinations j.
– Products p whose demand is equal to 500 at some destination j.

7-5. AMPL parameters can be defined to contain many kinds of series, especially by using recursive definitions. For example, we can make s[j] equal the sum of the first j integers, for j from 1 to some given limit N, by writing

```
param N;
param s {j in 1..N} = sum {jj in 1..j} jj;
```

or, using a formula for the sum,

```
param s {j in 1..N} = j * (j+1) / 2;
```

or, using a recursive definition,

```
param s {j in 1..N} = if j = 1 then 1 else s[j-1] + j;
```

This exercise asks you to play with some other possibilities.

(a) Define fact[n] to be n factorial, the product of the first n integers. Give both a recursive and a nonrecursive definition as above.

(b) The Fibonacci numbers are defined mathematically by $f_0 = f_1 = 1$ and $f_n = f_{n-1} + f_{n-2}$. Using a recursive declaration, define fib[n] in AMPL to equal the n-th Fibonacci number.

Use another AMPL declaration to verify that the n-th Fibonacci number equals the closest integer to $(\frac{1}{2} + \frac{1}{2}\sqrt{5})^n / \sqrt{5}$.

(c) Here's another recursive definition, called Ackermann's function, for positive integers i and j:

$$\begin{aligned} A(i, 0) &= i + 1 \\ A(0, j + 1) &= A(1, j) \\ A(i + 1, j + 1) &= A(A(i, j + 1), j) \end{aligned}$$

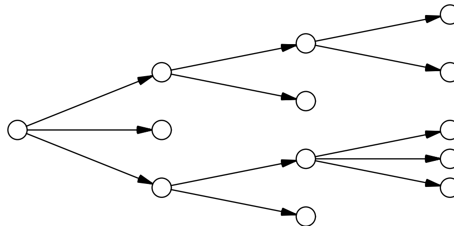
Using a recursive declaration, define ack[i,j] in AMPL so that it will equal $A(i, j)$. Use display to print ack[0,0], ack[1,1], ack[2,2] and so forth. What difficulty do you encounter?

(d) What are the values odd[i] defined by the following odd declaration?

```
param odd {i in 1..N} =
   if i = 1 then 3 else
      min {j in odd[i-1]+2 .. odd[i-1]*2 by 2:
         not exists {k in 1 .. i-1} j mod odd[k] = 0} j;
```

Once you've figured it out, create a simpler and more efficient declaration that gives a set rather than an array of these numbers.

(e) A ''tree'' consists of a collection of nodes, one of which we designate as the ''root''. Each node except the root has a unique predecessor node in the tree, such that if you work backwards from a node to its predecessor, then to its predecessor's predecessor, and so forth, you always eventually reach the root. A tree can be drawn like this, with the root at the left and an arrow from each node to its successors:



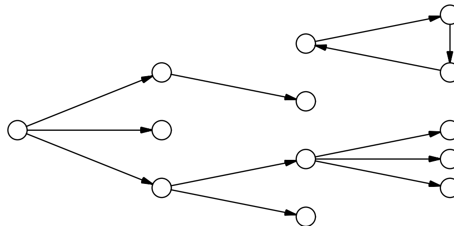
We can store the structure of a tree in AMPL sets and parameters as follows:

```
set NODES;
param Root symbolic in NODES;
param pred {i in NODES diff {Root}} symbolic in NODES diff {i};
```

Every node `i`, except `Root`, has a predecessor `pred[i]`.

The depth of a node is the number of predecessors that you encounter on tracing back to the root; the depth of the root is 0. Give an AMPL definition for `depth[i]` that correctly computes the depth of each node `i`. To check your answer, apply your definition to AMPL data for the tree depicted above; after reading in the data, use `display` to view the parameter `depth`.

An error in the data could give a tree plus a disconnected cycle, like this:



If you enter such data, what will happen when you try to display `depth`?