APPENDIX C

AMPL Modeling Language[1]

This appendix presents the principal syntactic rules of AMPL needed for the development and solution of complex mathematical programming models. For additional details, you may consult the basic language reference given at the end of this appendix (Fourer and Associates, 2003). You may also consult *www.ampl.com* for additional resources as well as the latest news and updates.

C.1 RUDIMENTARY AMPL MODEL

AMPL provides a facility for modeling mathematical programs (linear, integer, and nonlinear) in a longhand format. Figure C.1 gives the (self-explanatory) LP code for the Reddy Mikks model (file *RM1.txt*). All reserved words are in bold. The remaining symbols, other than the special operators $(+\ -\ *, ;:><=)$, are generated by the user.

AMPL uses command lines and operates in a DOS environment. A recent beta version of a Windows interface can be found in *www.OptiRisk-Systems.com.*

```
var x1 >= 0;
var x2 >= 0;
maximize z: 5*x1 + 4*x2;
subject to
  c1: 6*x1 + 4*x2 <= 24;
  c2: x1 + 2*x2 <= 6;
  c3: -x1 + x2 <= 1;
  c4: x2 <= 2;
solve;
display z,x1,x2;
```

FIGURE C.1
Rudimentary AMPL model (file *RM1.txt*)

[1]Folder AppenCFiles on the website includes all the files for this appendix.

You can execute a model by clicking on **ampl.exe** in the AMPL folder and, at the **ampl** prompt, typing the following command followed by Return:

```
ampl: model RM1.txt;
```

The output will be displayed on the screen as[2]

```
MINOS 5.5: Optimal solution found.
2 iterations
z = 21
x1 = 3
x2 = 1.5
```

The rudimentary longhand format given here is not recommended for solving practical problems because it is problem specific. The remainder of this appendix provides the details of how AMPL is used in practice.

C.2 COMPONENTS OF AMPL MODEL

Figure C.2 specifies the general structure of an AMPL model. The model is comprised of two basic segments: The top segment (elements 1 through 4) is the algebraic representation of the model, and the bottom segment (elements 5 through 7) supplies the data that drive the algebraic model. Thus, in LP, the algebraic representation in AMPL exactly parallels the following mathematical model:

$$\text{Maximize } z = \sum_{j=1}^{n} c_j x_j$$

FIGURE C.2
Basic structure of an AMPL model

Algebraic Representation	1. Sets definitions. 2. Parameters definitions. 3. Variables definitions. 4. Model representation (objective and constraints).
Model implementation	4. Input data. 5. Solution of the model. 6. Output results.

[2]Every version of AMPL has a default *solver* that carries out the computations needed to optimize the AMPL model. In the student version, MINOS is the default solver, and it can handle linear and nonlinear problems. The website includes other solvers: CPLEX, KNITRO, LPSOLVE, and LOQO. CPLEX handles linear, integer, and quadratic problems. LPSOLVE handles linear and integer problems. KNITRO and LOQO handle linear and nonlinear problems.

subject to

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i,\ i = 1, 2, \ldots, m$$

The advantage of this arrangement is that the same algebraic model can be used to solve a problem of any size simply by changing the input data: m, n, c_j, a_{ij}, and b_i.

A number of syntax rules apply to the development of an AMPL model:

1. AMPL files must be plain text (Windows Notepad editor creates plain text).
2. Commented text may appear anywhere in the model preceded with #.
3. Each AMPL statement, comments excluded, must terminate with a semicolon (;).
4. An AMPL statement may occupy more than one line. Breakpoints occur at a proper separator, such as a blank space, colon, comma, parenthesis, brace, bracket, or mathematical operator. An exception to this rule occurs with strings (enclosed in quotes `' '` or `" "`) where a breakpoint is designated by adding a backslash (\).
5. All keywords (with few exceptions) are in lower case.
6. User-generated names are case sensitive. A name must be alphanumeric, interspersed with underscores, if desired. No other special characters are allowed.

We will use the Reddy Mikks problem of Section 2.1 to show how AMPL works. Figure C.3 gives the corresponding model (file *RM2.txt*). For convenience, key (or reserved) words are emphasized in bold.

The algebraic model starts with the *sets* that define the indices of the general LP model. The user-generated names `resource` and `paint` each preceded by the keyword `set` correspond to the sets $\{i\}$ and $\{j\}$ in the general LP model. The specific elements of the sets `resource` and `paint` that define the Reddy Mikks model are given in the input data section of the model.

The *parameters* are user-generated names preceded by the keyword `param` that define the coefficients of the objective function and the constraints as a function of the variable and constraint sets. The parameters `unitprofit{paint}, aij{resource, paint}`, and `rhs{resource}` correspond, respectively, to the mathematical symbols c_j, a_{ij}, and b_i in the general LP model. The subscripts i and j are represented by AMPL sets `resource` and `paint`, respectively. The input data provide specific values of the parameters.

The *variables* of the model, x_j, are given the name `product` preceded by the keyword `var`. Again, `product` is a function of the set `paint`. We can add the nonnegativity condition (`>=0`) in the same statement. Else, the default is that the variables are unrestricted in sign.

Having defined the sets, parameters, and variables of the model, the next step is to express the optimization problem in terms of these elements. The objective-function statement specifies the sense of optimization using the keyword `maximize` or `minimize`. The objective value z is given the user name `profit` followed by a

```
#********************ALGEBRAIC MODEL******************
#-------------------------------------------------sets
set paint;
set resource;
#-----------------------------------------parameters
param  unitprofit{paint};
param  rhs {resource};
param  aij {resource,paint};
#-----------------------------------------variables
var product{paint} >= 0
#-----------------------------------------model
maximize profit: sum{j in paint} unitprofit[j]*product[j];
subject to limit{i in resource}:
       sum{j in paint} aij[i,j]*product[j]<=rhs[i];
#************************DATA**************************
data;
set paint := exterior interior;
set resource := m1 m2 demand market;
param unitprofit :=
               exterior 5
               interior 4;
param rhs:=
               m1        24
               m2        6
               demand    1
               market    2;

param aij: exterior interior : =
        m1        6     4
        m2        1     2
        demand    -1    1
        market    0     1;

#**********************SOLUTION************************
solve;
#-------------------------------------output results
display profit, product, limit.dual, product.rc;
```

FIGURE C.3
AMPL model for the Reddy Mikks problem (file *RM2.txt*)

colon (:), and its AMPL statement is a direct translation of the mathematical expression $\sum_j c_j x_j$:

```
sum{j in paint} unitprofit[j]*product[j];
```

The index `j` is user specified. Note the use of braces in `{j in paint}` to indicate that `j` is a member of the set `paint`, and the use of brackets in `[j]` to represent a subscript.

A model may include one or more constraint statements, and each such statement can be preceded by the keywords `subject to` or simply `s.t.` Actually, `s.t.` and `subject to` are optional, and AMPL assumes that any statement that does not start with a keyword is a constraint. The Reddy Mikks model has only one set of constraints named `limit` and indexed over the set `resource`:

```
limit{i in resource}:
     sum{j in paint} aij[i,j]*product[j] <= rhs[i];
```

The statement is a direct translation of constraint i, $\sum_j a_{ij}x_{ij} \leq b_i$.

The idea of declaring variables as nonnegative can be generalized to allow establishing upper and lower bounds on the variables, thus eliminating the need to declare these bounds as explicit constraints. First, the two bounds are declared with the user-generated names `lowerbound` and `upperbound` as

```
param lowerbound{paint};
param upperbound{paint};
```

Next, the variables are defined as

```
var product{j in paint}>=lowerbound[j],<=upperbound[j];
```

Notice that the syntax does not allow comparing "vectors." Thus, an error is generated if we use

```
var product{paint}>=lowerbound{paint},<=upperbound{paint};
```

We can use the same syntax to set conditions on *parameters* as well. For example, the statement

```
param upperbound{j in paint}>=lowerbound[j];
```

will guarantee that `upperbound` is never less than `lowerbound`. Else AMPL will issue an error. The main purpose of using bounds on parameters is to prevent entering conflicting data inadvertently. Another instance where such checks may be used is when a parameter is required to assume nonnegative values only.

The algebraic model in Figure C.3 is general in the sense that it applies to any number of variables and constraints. It can be tailored to the Reddy Mikks situation by specifying the data of the problem. Following the statement `data;` we first define the members of the sets, and then use these definitions to assign numeric values to the different parameters.

The set `paint` includes the names of two variables, which we suggestively call `exterior` and `interior`. Members of the set `resource` are given the names `m1`, `m2`, `demand`, and `market`. The associated statements in the data section are thus given as

```
set paint := exterior interior;
set resource := m1 m2 demand market;
```

Members of each set appear to the right of the reserved operator := separated by a blank space (or a comma). String indices must be enclosed in double quotes when used *outside* the data segment—that is, paint["exterior"], paint["interior"], limit["m1"], limit["m2"], limit["demand"], and limit["market"]. Otherwise, the string index will be incorrectly interpreted as a (numeric) parameter.

We could have *defined* the sets at the start of the algebraic model (instead of in the data segment) as

```
set resource ={"m1", "m2", "demand", "market"};
set paint ={"exterior", "interior"};
```

(Note the mandatory use of the double quotes " ", the separating commas, and the braces.) This convention is not advisable in general because it is problem specific, which may limit tailoring the model to different input data scenarios. When this convention is used, AMPL will not allow modifying the set members in the data segment.

The use of alphanumeric names for the members of the sets resource and paint can be cumbersome in large problems. For this reason, AMPL allows the use of purely numeric sets—that is, we can use

```
set paint:= 1 2;
set resource:= 1..4;
```

The range 1..4 replaces the explicit 1 2 3 4 representation and is useful for sets with a large number of members. For example, 1..1000 is a set with 1000 members starting with 1 and ending with 1000 in increments of 1.

The range representation can be made more general by first defining m and n as parameters

```
param m;
param n;
```

In this case, the sets 1..m and 1..n can be used directly throughout the entire model as shown in Figure C.4 (file *RM2a.txt*), eliminating altogether the need to use the set names resource and paint.

Actually, the syntax 1..m (or 1..n) has the general format

```
start..end by step
```

where start, end, and step are defined AMPL parameters whose values are specified under data. If start < end and step > 0, then members of the set begin with start and advance by the amount step to the highest value less than or equal to end. The opposite occurs if start > end and step <0. For example, 3..10 by 2 produces the members 3, 5, 7, and 9, and 10..3 by -2 produces the members 10, 8, 6, and 4. The default for step is 1, which means that start..end by 1 is the same as start..end.

Actually, the parameters start, end, and step can be any legitimate AMPL mathematical expressions computed during execution. For example, given the parameters m and n, the set j in 2*n..m + n^2 by n/2 is perfectly legal. Note, however, that

```
param m;
param n;
param  unitprofit{1..n};
param  rhs{1..m};
param  aij{1..m,1..n};
#-----------------------------------------variables
var product{1..n}>=0;
#-------------------------------------------------model
maximize profit:sum{j in 1..n}unitprofit[j]*product[j];
subject to limit{i in 1..m}:
       sum{j in 1..n}aij[i,j]*product[j]<=rhs[i];
data;
param m:=4;
param n:=2;
param unitprofit := 1 5 2 4;
param rhs:=1 24 2 6 3 1 4 2;
param aij:  1     2:=
     1      6     4
     2      1     2
     3      -1    1
     4      0     1;
solve;
display profit, product, limit.dual, product.rc;
```

FIGURE C.4

AMPL model for the Reddy Mikks problem (file *RM2a.txt*)

a fractional `step` is used directly to create the members of the set. For example, for `m = 5, n = 13`, the members of the set `m..n step m/2` are 5, 7.5, 10, and 12.5.

The Reddy Mikks model includes single- and two-dimensional parameters. The parameters `unitprofit` and `rhs` fall in the first category and the parameter `aij` in the second. In the first category, data are specified by listing each set member followed by a numeric value, as the following statements show:

```
param unitprofit :=
     exterior 5
     interior 4;
param rhs:=
     m1        24
     m2        6
     demand    1
     market    2;
```

The elements of the list may be "strung" into one line, if desired. The only requirement is a separation of at least one blank space. The format given here promotes better readability.

Input data for the two-dimensional parameter `aij` are prepared similar to that of the one-dimensional case, except that the order of the columns must be specified after `aij:` to eliminate ambiguity, as the following statement shows:

```
param aij:          exterior   interior :=
          m1               6          4
          m2               1          2
          demand          -1          1
          market           0          1;
```

Again, the list is totally free-formatted so long as the logical sequential order is preserved and the elements are separated by blank spaces.

AMPL allows assigning default values to all the elements of a parameter. For example, suppose that for a parameter c, $c_1 = 11$ and $c_8 = 22$ with $c_i = 0$ for $i = 2, 3, \ldots, 7$. We can use the following statements to take care of this situation:

```
param c{1..8};
.
.
.
data;
param c:=1 11 2 0 3 0 4 0 5 0 6 0 7 0 8 22;
```

A more compact way of achieving the same result is to use the following statements:

```
param c{1..8} default 0;
.
.
.
data;
param c:=1 11 8 22;
```

Initially, `c[1]` through `c[8]` assume the default value 0, with `c[1]` and `c[8]` changed to 11 and 22 in the `data` segment. In general, `default` may be followed by any mathematical expression. This expression is evaluated only once at the start of the execution.

The final segment of the AMPL model deals with obtaining the solution and the presentation of the output. The command `solve` is all that is needed to solve the model. Once completed, specific output results may be requested. The command `display` followed by an output list is but one way to view the results. In the Reddy Mikks model, the statement

```
display profit, product, limit.dual, product.rc;
```

requests the optimal values of the objective function and the variables, `profit` and `product`; the dual values of the constraints, `limit.dual`; and the reduced costs of the variables, `product.rc`. The keywords `dual` and `rc` are suffixed to the names of the constraints `limit` and variables `product` separated by a period. They may not be used as stand-alone keywords. The output defaults to the screen. It may be directed to an external file by inserting *>filename* immediately before the semicolon. Section C. 5 provides more details about how output is directed to files and spreadsheets.

```
MINOS 5.5: optimal solution found.
2 iterations. objective 21

profit = 21

:          product  limit.dual  product.rc   :=
demand        .         0            .
exterior      3          .       6.66134e-16
interior      1.5        .       0
m1            .        0.75           .
m2            .        0.5            .
market        .         0             .
;
```

FIGURE C.5

AMPL output using `display profit, product, limit.dual. product.rc;` in the Reddy Mikks model

The execution command in DOS is

```
ampl: model RM2.txt;
```

The associated output is displayed on the screen, as the snapshot in Figure C.5 shows.

The layout of the output in Figure C.5 is a bit "cluttered" because it mixes the indices of the constraints and the variables. We can streamline the output by placing its elements in groups of the same dimension using the following two `display` statements:

```
display profit, product, product.rc;
display limit.dual;
```

In a typical AMPL model, such as the one in Figure C.3, the segment associated with the logic of the model preferably should remain static. The data and output segments are changed as needed to match specific LP scenarios. For this purpose, the AMPL model is represented by two separate files: *RM2b.txt* providing the logic of the model and *RM2b.dat* accounting for the input data and the output results.[3] In this case, the DOS line commands are entered sequentially as:

```
ampl: model RM2b.txt;
ampl: data RM2b.dat;
```

We will see in Section C.7 how commands such as `solve` and `display` can be issued interactively rather than being hard-coded in the model.

The Reddy Mikks model provides only a "glimpse" of the capabilities of AMPL. We will show later how input data may be read from external files and spreadsheet

[3]Actually, the output command may be processed separately instead of being included in the .dat file, as will be explained in Section C.7

tables. We will also show how tailored (formatted) output can be sent to these media. Also, AMPL interactive commands are important debugging and execution tools, as will be explained in Section C.7.

PROBLEM SET C.2A

1. Modify the Reddy Mikks AMPL model of Figure C.3 (file *RM2.txt*) to account for a third type of paint named "marine." Requirements per ton of raw materials `m1` and `m2` are .5 and .75 ton, respectively. The daily demand for the new paint lies between .5 ton and 1.5 tons. The revenue per ton is $3.5 (thousand). No other restrictions apply to this product.
2. In the Reddy Mikks model of Figure C.3 (file *RM2.txt*), rewrite the AMPL code using the following set definitions:
   (a) `paint` and {`1..m`}.
   (b) {`1..n`} and `resource`.
   (c) {`1..m`} and {`1..n`}.
3. Modify the definition of the variables in the Reddy Mikks model of Figure C.3 (file *RM2.txt*) to include a minimum demand of 1 ton of exterior paint and maximum demands of 2 and 2.5 tons of exterior and interior paints, respectively.
4. In the Reddy Mikks model of Figure C.3, the command

   ```
   display profit;
   ```

   provides the value of the objective function. We can use the same command to display the contribution of each variable to the total profit as follows:

   ```
   display profit, {j in paint} unitprofit[j]*product[j];
   ```

   Another convenient way to accomplish the same result is to use **defined variable** statements as follows:

   ```
   var extProfit=unitprofit["exterior"]*product["exterior"]
   var intProfit=unitprofit["interior"]*product["interior"]
   ```

   In this case, the objective function and display statements may be written in a less complicated form as

   ```
   maximize profit: extProfit + intProfit;
   display profit, extProfit, intProfit;
   ```

   In fact, defined variables can be in indexed form as:

   ```
   var varProfit{j in paint} = unitprofit[j]*product[j];
   ```

   The resulting objective function and display statement will then read as

   ```
   maximize profit: sum {j in paint} varProfit[j];
   display profit, varProfit;
   ```

   Use *defined variables* with the Reddy Mikks model to allow displaying each variable's profit contribution and resource consumption of raw materials `m1` and `m2`.
5. Develop and solve an AMPL model for the diet problem of Example 2.2-2, and find the optimum solution. Determine and interpret the associated dual values and the reduced costs.

C.3 MATHEMATICAL EXPRESSIONS AND COMPUTED PARAMETERS

We have seen that AMPL allows placing upper and lower bounds on parameters. Actually, the language affords more flexibility in defining parameters as complex mathematical expressions, modified conditionally, if desired.

To illustrate the use of computed parameters, consider the case of a bank offering n types of loans that charges an interest rate r_i for loan i, $0 < r_i < 1, i = 1, 2, \ldots, n$. Unrecoverable bad debt, both principal and interest, for loan i equals v_i of the amount of loan i. The objective is to determine the amount x_i the bank allocates to loan i to maximize the total return subject to a set of restrictions.

The use of computed parameters will be demonstrated by concentrating on the objective function. Algebraically, the objective function is expressed as

$$\text{Maximize } z = \sum_{i=1}^{n} r_i(1 - v_i)x_i - \sum_{i=1}^{n} v_i x_i = \sum_{i=1}^{n} [r_i - v_i(r_i + 1)]x_i$$

A direct translation of z into AMPL is the following:

```
param r{1..n}>0, <1;
param v{1..n}>0, <1;
var x{1..n}>=0;

maximize z: sum{i in 1..n}(r[i]-v[i]*(r[i]+1))*x[i];

(constraints)
```

Another way to handle the bank situation is to use a computed parameter to represent the objective function coefficients in the following manner:

```
param r{1..n}>0, <1;
param v{1..n}>0, <1;
param c{i in 1..n}=(r[i]-v[i]*(r[i]+1));
var x{1..n}>=0;

maximize z: sum{i in 1..n}c[i]*x[i];

(constraints)
```

AMPL will compute the parameter `c[i]` and use its value in the objective statement z. The new formulation enhances readability. But in some cases the use of computed parameters may be essential.

In general, the expression defining the value of a computed parameter can be of any complexity and may include any of the built-in arithmetic functions familiar to any programming language (e.g., `sin`, `max`, `log`, `sqrt`, `exp`). An important requirement is that the expression evaluate to a numeric value.[4]

Computed parameters may also be evaluated conditionally using the construct

parameter `=` `if` *condition* `then` *expression*1 `else` *expression*2`;`

[4]AMPL manual provides an "exception" when a parameter is declared `binary`, in which case it can also be treated as *logical*. This distinction is artificial, because treating such a parameter as numeric still produces the same result.

The *condition* compares arithmetic quantities and strings using the familiar operators =, <, >, <=, >=, and <> (together with `and/or`). Note that nonlinearity will result if *condition* is a function of the model variables. As in other programming languages, the construct may be used without `else` *expression2*. Nested `if` is also allowed following `then` and `else`.

The `if-then-else` construct gives the computed parameters the numeric value of either *expression*1 or *expression*2. This is the reason the `if-then-else` presented here is an *expression* and not a *statement*. (Section C.7 introduces the `if-then-else` *statement* together with the loop statements `for{}`, `repeat while{}`, and `repeat until{}`. These statements are used mainly for automating solution scenarios and formatting output.)

We will use a simple case to demonstrate the use of the `if` expression. In a multiperiod manufacturing situation, units of a certain item are produced to meet variable demand. Unit production cost is estimated at p dollars for the first m periods, and increases by 10% for the next m periods and by 20% for the following m periods.

The constraints of this model deal with capacity restrictions for each period and the balance equations that relate inventory, production, and demand. To demonstrate the use of the `if` expression, we will concentrate on the objective function. Let

$$x_j = \text{units produced in period } j, j = 1, 2, \ldots, 3m$$

The objective function is given as

Minimize

$$z = p(x_1 + x_2 + \cdots + x_m) + 1.1p(x_{m+1} + x_{m+2} + \cdots + x_{2m}) + 1.2p(x_{2m+1} + x_{2m+2} + \cdots + x_{3m})$$

We can model this function in AMPL as

```
param p;
var x{1..3*m}>=0;

minimize cost: p*(sum{j in 1..m}x[j]+1.1*sum{j in m+1..2*m}x[j]+
      1.2*sum{j in 2*m+1..3*m}x[j]);

(constraints)
```

A more compact way that also enhances readability is to use `if-then-else` to represent the objective-function parameter `c[j]`:

```
param m;
param n=3*m;
param p;
param c{j in 1..n}= if j<=m then p else
                    (if j>m and j<=2*m then 1.1*p else 1.2*p);

var x{j in 1..n};

minimize z: sum{j in 1..n}c[j]*x[j];

(constraints)
```

Note the nesting of the conditions. The parentheses () enclosing the second `if` are not necessary and are used to enhance readability. Observe that `then` and `else` are always followed by what must evaluate to numeric values. Note also that c can be defined as

```
param c{j in 1..n}=p*( if j<=m then 1 else
                  (if j>m and j<=2*m then 1.1 else 1.2));
```

A particularly useful implementation of `if-then-else` occurs in the situation where parameters or variables are defined *recursively*. A typical example of such a parameter occurs in determining the inventory level I_t in period t, $t = 1, 2, \ldots, n$, with initial zero inventory. The production amount and demand in period t are p_t and d_t, respectively. Thus, the inventory level is

$$I_0 = 0$$
$$I_t = I_{t-1} + p_t - d_t,\ t = 1, 2, \ldots, n$$

The amount I_t can be computed recursively in AMPL as follows:

```
param p{1..n};
param d{1..n};
var I{t in 1..n}= if i=1 then 0 else I[t-1])+p[t]-d[t];
```

Notice that it would be somewhat cumbersome to compute I_t were it not for the use of the `if-then-else` expression (see also Set C.3a).

The bus scheduling problem (Example 2.4-5) demonstrates the use of `if . . . then . . . else` within the context of a complete model. These two models are detailed in Section C.9.

PROBLEM SET C.3A

***1.** Consider the following set of constraints:

$$x_i + x_{i+1} \geq c_i, i = 1, 2, \ldots, n - 1$$
$$x_1 + x_n \geq c_n$$

Use `if-then-else` to develop a single set of constraints that represents all n inequalities.

2. In a multiperiod production-inventory problem, let x_t, z_t, and d_t be, respectively, the amount of entering inventory, production quantity, and demand for period t, $t = 1, 2, \ldots, T$. The balance equation associated with period t is $x_t + z_t - d_t - x_{t+1} = 0$. In a specific situation, $x_1 = c$ (>0) and $x_{T+1} = 0$. Write the AMPL constraints corresponding to the balance equations using `if-then-else` to account for $x_1 = c$ and $x_{T+1} = 0$.

3. Consider the parameters a_i and $b_{i,j}$, $i \in \{1, n\}$ and $j \in \{1, a_i\}$. Define a_i and b_j as AMPL parameters.

C.4 SUBSETS AND INDEXED SETS

Subsets. Suppose that we have the following constraint:

$$x_1 + x_2 + x_5 + x_6 + x_7 \leq 15$$

There are 7 variables in the model, and this particular constraint does not include the variables x_3 and x_4.

We can model this constraint by using subsets in a number of ways (all *new* keywords are in bold):

```
#---------------------- method 1 ------------------------------------
var x{1..7}>=0;
subject to lim: sum{j in 1..7: j<=2 or j>=5}x[j]<=15;
#---------------------- method 2 ------------------------------------
var x{1..7}>=0;
subject to lim: sum{j in 1..2 union 5..7}x[j]<=15;
#---------------------- method 3 ------------------------------------
var x{1..7}>=0;
subject to lim: sum{j in 1..7 diff 3..4}x[j]<=15;
#---------------------- method 4 ------------------------------------
var x{1..7}>=0;
subject to lim: sum{j in 1..7 diff (1..4 inter 3..7)}x[j]<=15;
#--------------------------------------------------------------------
```

In method 1, the set `{j in 1..7}` deletes the elements 3 and 4 by imposing restrictions on `j`. A colon separates the modified set from the condition(s). Keywords `union`, `diff`, and `inter` play the roles of $A \cup B$, $A - B$, and $A \cap B$, respectively. Method 4 is a convoluted set representation. Nevertheless, it serves to represent the use of the operator `inter`.

Indexed sets. A powerful feature of AMPL allows indexing sets over the elements of a regular set. Suppose that two components A and B are used to produce products 1, 2, 3, 4, and 5. Component A is used in products 1, 3, and 5, and component B is used in products 1, 2, 4, and 5. Each product requires one unit of the specified components. The maximum availabilities of components A and B are 200 and 300 units, respectively. The problem deals with determining the number of assembly units of each product. Other pertinent data will be needed to complete the description of the problem, but we will concentrate only on the constraints dealing with the components' availability.

Let x_i be the production quantity of product i, $i = 1, 2, \ldots, 5$. Then the constraints for components A and B can be expressed mathematically as

$$\text{Component } A\text{: } x_1 + x_3 + x_5 \leq 200$$

$$\text{Component } B\text{: } x_1 + x_2 + x_4 + x_5 \leq 300$$

The AMPL representation of the constraints can be achieved using indexed sets as follows:

```
set comp;
set prod{comp};
param d{comp};
var x{1..5}>=0;
#———objective function here
```

```
subject to
     C{i in comp}:sum{j in prod[i]}x[j]<=d[i];
#-———other constraints here
data;
set comp:= A B;
set prod[A]:=1 3 5;
set prod[B]:=1 2 4 5;
param d:= A 200 B 300;
```

The indices of set `prod` are the elements `A` and `B` of set `comp`, thus defining the two indexed sets `prod[A]` and `prod[B]`. Next, the data of the problem define the elements of `prod[A]` and `prod[B]`. With these data, the constraints of the components (regardless of how many there are) are defined by the single statement:

```
C{i in comp}:sum{j in prod[i]}x[j]]<=d[i];
```

The applications of indexed sets are demonstrated aptly in the *AMPL moments* following Examples 6.6-4 and 9.1-2.

PROBLEM SET C.4A

1. Use subsets to express the left-hand side by means of a single `sum{}` function:

(a) $\sum_{j=1}^{m} x_j + \sum_{j=m+k}^{n} x_j + \sum_{j=n+p}^{q} x_j \geq c$

(b) $\sum_{i=m}^{n} x_i + \sum_{i=n+k}^{2n+k} x_i \leq c, k > 1$

***2.** Suppose that 5 components (one unit per product unit) are used in the production of 10 products according to the following schedule:

Component	Products that use the component	Minimum availability
1	1, 2, 5, 10	500
2	3, 6, 7, 8, 9	400
3	1, 2, 3, 5, 6, 7, 9	900
4	2, 4, 6, 8, 10	700
5	1, 3 4, 5, 6, 7, 9, 10	100

The unit assembly cost of each product is a function of the component used: \$9, \$4, \$6, \$5, and \$8 for components 1 through 5, respectively. The maximum demand for any of the products is 300 units. Use AMPL *indexed sets* to determine the optimal product mix that minimizes the installation cost. (*Hint*: Let x_{ij} be the number of units of product i that use component j.)

3. Repeat Problem 2 assuming that the unit installation cost of the components is a function of the assembled product: \$1, \$3, \$2, \$6, \$4, \$9, \$2, \$5, \$10, and \$7 for products 1 through 10, respectively.

C.5 ACCESSING EXTERNAL FILES

So far, we have used "hard-coded" data to drive AMPL models. Actually, AMPL data may be accessed from external files, spreadsheets, and/or databases. The same is true for retrieving output results. This section deals with reading data from or writing output to

1. External files, including screen and keyboard.
2. Spreadsheets.

More details can be found in Fourer and Associates, 2003, Chapter 10.

C.5.1 Simple read files

The statement for reading data from an unformatted external file is

read *item-list* <*filename*;

The *item-list* is a comma-separated list of nonindexed or indexed parameters. In the indexed case, the syntax is {*indexing*}*paramName*[*index*]. The list can include parameters only. This means that any set members must be accounted for under data *prior to* invoking the `read` statement. (We will see in Sections C.5.3 and C.5.4 how set members are read from formatted files and spreadsheets.)

To illustrate the use of `read`, consider the Reddy Mikks model where all the data for the parameters `unitprofit`, `rhs`, and `aij` are read from a file named *RM3.dat* per the model in file *RM3.txt*. The associated read statement is:

```
read {j in paint}unitprofit[j],
     {i in resource}rhs[i],
     {i in resource, j in paint}aij[i,j]<RM3.dat;
```

File *RM3.dat* lists the data in the exact order in which the items appear in the read list—that is,

```
5       4
24      6       1       2
6       4
1       2
-1      1
0       1
```

The multiple-row organization of the data enhances readability, in the sense that we could have had all the elements on one line (separated by blank spaces).[5] Note that this file happens to be all numeric. For convenience, nonnumeric data (such as parameter

[5]Hidden codes in `.dat` files (and in `.tab` files, which will be presented later in this section) can trigger AMPL errors such as "too few elements in line xx" or "unexpected end of file" (xx stands for a numeric value) even though the text file may appear perfectly legal. To get rid of these hidden codes, click immediately to the right of the last data element in the file, then press the following keys in succession: Return, Backspace, and Return.

names) can appear in the data file provided that they are declared `symbolic` (for details, see Sections 7.8 and 9.5 in Fourer and Associates, 2003).

The `read` statement allows accessing data from the keyboard. In this case, the *filename* is replaced with a minus sign—that is, using `<-`. The execution of `read` in this case will produce the DOS prompt `ampl?`, and it will be repeated until all the data requested by `read` have been accounted for.

PROBLEM SET C.5A

1. Prepare the input file *RM3x.dat* for the Reddy Mikks model (file *RM3.txt*), assuming that the `read` statement is given as

```
read  {j in paint}
      {i in resource}
      (
      rhs[i],
      {j in paint}aij[i,j]
      )<RM3x.dat;
```

*__2.__ For the Reddy Mikks model, explain why the following `read` statement is cumbersome:

```
read {i in resource}
     (
     rhs[i],
     {j in paint}(unitprofit[j],aij[i,j])
     )<RM3xx.dat;
```

C.5.2 Using `print` or `printf` to retrieve output

A simple way to retrieve output data in AMPL is to use preformatted `print` or formatted `printf`. As an illustration, in the Reddy Mikks model we can use the following statements to send output data to a file we name `file.out` (output defaults to the screen if a file is not designated):

```
printf "Objective value is %6.2f\n",profit >file.out;
printf {j in paint}:
     "%8s%8.2f%8.3f\n",j,product[j],product[j].rc >file.out;
```

The output format always precedes the output list and must be enclosed in double quotes. The same statement can be used with `print` simply by removing the format code.

In the first `printf` statement, the format includes the optional descriptive text `Objective value` is and mandatory specifications of how the output list is printed. The code `%6.2f` says that the value of `profit` is printed in a field of length 6 with two decimal points. The code `\n` moves printing to the next line in the file. These format codes are the same as in C programming.

In the second print statement, the output list includes `j`, `product[j]`, `product[j].rc`, where `j` is one of the members (`exterior, interior`) in the AMPL set `paint`. The code `%8s` reserves the first eight fields for printing the name `exterior`

`or interior`. If `j` were numeric (e.g., `{j in 1..2}`), then the format specification would have to be integer, for example, `%3i`.

The format specifications in this section are limited to `%s`, `%i`, `%f`, and `\n`. AMPL provides other specifications (see Table A-10 in Fourer and Associates, 2003).

PROBLEM SET C.5B

1. Use `printf` statements to present the optimal solution of the Reddy Mikks model (file *RM2.txt*) in the following format where the suffixes `.slack` and `.dual` are used to retrieve slack amount and the dual price:

```
Objective value =
-------------------------------------
Product      Quantity       Profit($)
-------------------------------------
     .            .              .
     .            .              .
     .            .              .
                                 .
Constraint   Slack amount   Dual price
-------------------------------------
     .            .              .
     .            .              .
     .            .              .
-------------------------------------
```

C.5.3 Input table files

The `read` statement in Section C.5.1 does not allow reading set members. This situation is accounted for using `table` statements.

In `table` files, the data are presented as tables with properly labeled rows and columns using the members of the defining sets. Access to `table` files requires a companion `read` statement. The `table` statement formats the data, and the `read` statement makes the data available to the model.

The syntax of `table` and `read` statements is as follows:

```
table tableName IN "fileName":SetName<-[SetColHdng], parameters~ParamColHdng;
read table tableName;
```

This syntax allows reading *both* the members of AMPL sets and the parameters from *tableName* in *fileName*.

The default *fileName* where the text table is stored is *tableName*`.tab`. It may be overridden by explicitly specifying *fileName* (in double quotes) with mandatory `.tab` extension following the keyword `IN`. `IN` (in caps) means INput (as contrasted with `OUT`, which, as shown later, is used to OUTput data to a table file). *SetColHdng* may be an arbitrary heading name in the table which is cross-referenced to the elements of *SetName* using `<-`. Similarly, AMPL parameters are cross-referenced to the arbitrary names *ParamColHdng* using `~`. If *SetColHdng* happens to be the same as *SetName*, the syntax *SetName*`<-` [*SetColHdng*] may be replaced with `[`*SetName*`]` `IN`. In the case of parameters, `~`*ParamColHdng* is deleted from the statement.

To illustrate the use of tables, Figure C.6 gives the contents of the files named *RM4profit.tab, RM4rhs.tab*, and *RM4aij.tab* for inputting the parameters `unitprofit`,

File *RM4profit.tab*:

```
ampl.tab 1 1
COL1            COL2
exterior        5
interior        4
```

File *RM4rhs.tab*:

```
ampl.tab 1 1
resource rhs
m1       24
m2       6
demand   1
market   2
```

File *RM4aij.tab*:

```
ampl.tab 2 1
resource paint       aij
m1       exterior    6
m1       interior    4
m2       exterior    1
m2       interior    2
demand   exterior    -1
demand   interior    1
market   exterior    0
market   interior    1
```

FIGURE C.6
Contents of the table files for inputting the parameters `unitprofit`, `rhs`, and `aij` of the Reddy Mikks model

`rhs`, and `aij` of the Reddy Mikks model. The first line in each file must always follow the format

`ampl.tab` *nbr_indexing_sets nbr_read_parameters*

The first element, `ampl.tab`, identifies the table as a `.tab` file, with the succeeding two elements providing the number of indexing sets of the parameters that will be read from the table. In `RMprofit.tab` and `RMrhs.tab`, only *one* set is needed to define the parameters `unitprofit` and `rhs`, and for this reason `ampl.tab 1 1` is used as the header line in these two files. For the parameter `aij`, *two* sets are needed, which requires the use of the header line `ampl.tab 2 1`.

The header line is followed by a list of the *exact* or *substitute* names of the sets and the parameters. The succeeding rows in the respective file list the values of the input parameter as an explicit function of its indexing set(s) using blank space(s) as separators. For `unitprofit` and `rhs`, the listing is straightforward. For the double-indexed parameter `aij`, each parameter list is identified by two *explicit* indices, even at the expense of redundancy.

For the Reddy Mikks model, the associated tables are defined as follows:

```
table RM4profit IN: paint <- [COL1], unitprofit~COL2;
table RM4rhs IN: [resource] IN, rhs;
table RM4aij IN: [resource, paint], aij;
```

Following the declaration of the `table` statements, we can use the following statements to read in the data:

```
read table RMprofit;
read table RMrhs;
read table RMaij;
```

For readability, it is recommended that the table declaration statements follow the constraints segment. The `read` statements are then placed immediately below the table declarations (see file *RM4.txt*).

The table statements above illustrate four syntactic rules:

1. In all three tables, the default file name is the table name with `.tab` extension (else a file name enclosed in `""` must be given immediately before the semicolon).
2. In the `profit` statement, the syntax `paint<-[COL1]` tells AMPL that the entries in the `COL1`-column in file *`RM4profit.tab`* define the members of the set `paint`.
3. In the `profit` statement, the syntax `unitprofit~COL2` cross-references the entries in `COL2` with the parameter `unitprofit`.
4. In the `rhs` statement, [`resource`] `IN` automatically defines the members of the set `resource` because the name `resource` is used as a column heading in the table.
5. In the `aij` statement, `aij` has (at least) two dimensions, hence the statement *cannot* be used to read the members of the associated sets. Instead, these sets must be read from the single-dimensional tables `RM4profit` and `RM4rhs`. Thus, `[resource, paint]` in the `aij` statement are used only to define the indices of the parameter `aij`.

In general, if a model has no single-indexed parameters, a table can be declared for the sole purpose of reading in the members of a set from a file. In this case, the header line in the `.tab` file must read `ampl.tab 1 0`, indicating that the file includes one column for the set members and no parameters. For example, the following statement declares the table for reading the elements of the set `paint` from file *`paintSet.tab`*:

```
table paintSet IN: [paint] IN;
```

In this case, the contents of *`paintSet.tab`* will be

```
ampl.tab 1 0
paint
exterior
interior
```

An alternative method for reading in `aij` without reading in the elements of `resource` and `paint` first is to declare

```
set Z dimen 2;
param aij{Z};
```

In this case, we can read in `aij` from table RM4aij in the following manner

```
table RM4aij IN: Z<-[resource, paint], aij;
read table RM4aij;
```

Keep in mind that these statements do not define the elements of the sets `resource` and `paint`. To define the elements of the sets resource and paint

```
set resource=setof{(i, j) in Z} i;
set paint=setoff{(i, j) in Z} j;
```

File *RM4x.txt* implements these ideas. It should be clear, however, that this is a convoluted way for defining the sets `resource` and `paint`. The use of the two-dimensional set `Z` as given above is advisable only if the sets `resource` and `paint` are not needed to define variables or parameters in the model. In such a case, we can use dummy sets X and Y to define the two-dimensional set `z`; that is, `z<-[X, Y]`.

In some cases it may be convenient to read the data of a two-dimensional parameter as an array in place of two indexed single elements, as given above for aij. AMPL allows this by changing the definition of the table to:

```
table RM4arrayAij IN:[i~resource], {j in paint}<aij[i, j]~(j)>;
```

(The new table definition is somewhat "overcoded" in the sense that `~(j)` appears redundant. Nevertheless, it gets the job done.) In this case, the file *RM4arrayAij.tab* must appear as

```
ampl.tab 1 2
resource    exterior   interior
m1                 6          4
m2                 1          2
demand            -1          1
market             0          1
```

Note that the header `ampl.tab 1 2` indicates that table `RM4arrayAij` has one key index (namely, `[i~resource]`) and two data columns with the headings `exterior` and `interior`. The new table, `RM4arrayAij`, does not permit reading the members of the sets `resource` and `paint`, the same restriction table `RM4aij` has. (See file *RM4.txt*.)

C.5.4 Output table files

Table files may also receive output from AMPL *after* the `solve` command has been executed. The syntax is similar to that of the input files, except that in the table declaration, `IN` is replaced with `OUT`. For example, in the Reddy Mikks model, suppose that we are interested in retrieving the following information:

1. Values of the variables and their reduced costs.
2. Slack and dual values associated with the constraints.

This information requires the use of *two* tables because the two item are functions of distinct sets: `paint` and `resource`:

```
table varData OUT:[paint], product, product.rc;
table conData OUT:
      [resource], limit.slack~slack, limit.dual~Dual;
```

The OUT-table declaration statements should be placed after the constraints segment to ensure that names of variables and constraints used in these statements have already been defined (see file *RM4.txt*). The syntax `limit.slack~slack` and `limit.dual~Dual` assign the descriptive header names `slack` and `Dual` to the columns where the corresponding data are written in the file. Otherwise, the default header names will be `limit.slack` and `limit.dual`.

To retrieve the output, we need to issue the command `solve` and then follow it with the following `write` statements:

```
write table varData;
write table conData;
```

The output will be sent to files `varData.tab` and `conData.tab`, respectively. As with the input case, we can override the default file name by entering (in double quotes) a specific name (with `.tab` extension) following the keyword `OUT` and immediately before the colon.

Output tables can also be used to send two-dimensional arrays to a file. For example, either one of the following two definitions can be used to send the array `aij` to a `.tab` file:

```
table AijMatrix OUT: [resource, paint], aij;
table Aijout OUT:{i in resource}->[RESOURCES], {j in paint}<aij[i, j]~(j)>;
```

In the first definition, file *AijMatrix.tab* lists each element of `aij` with its two indices on the same row. In the second, file Aijout.tab lists `aij` in an array format, with the user-specified name `RESOURCES` being the heading of the first (key) column.

PROBLEM SET C.5C

***1.** In *RM4.txt*, suppose the statements

```
read table RM4profit;
read table RM4rhs;
read table RM4aij;
```

are replaced with

```
read table RM4aij;
data;
param unitprofit:= exterior 5 interior 4;
param rhs:=m1 24 m2 6 demand 1 market 2;
```

Explain why AMPL will not execute properly with the proposed change. (*Hint*: The best way to find out the answer is to experiment with the model.)

2. Suppose that the contents of file *RM4rhs.tab* read as

```
ampl.tab 1 1
constrName Availability
m1    24
m2     6
demand 1
market 2
```

Make the necessary changes in *RM4.txt*, and execute the model.

C.5.5 Spreadsheet input/output tables

Accessing data from and sending data to a spreadsheet uses syntax similar to that of the table files presented in Section C.5.4. The following statements show how the input data of the Reddy Mikks model can be accessed from an Excel spreadsheet file named *RM5.xls*:

```
table profitVector IN "ODBC" "RM5.xls":paint<-[COL1], unitprofit~COL2;
table rhsVector IN "ODBC" "RM5.xls":[resource] IN, rhs;
table aijMatrix IN "ODBC" "RM5.xls":[resource, paint], aij;
```

The user-generated names `profitVector`, `rhsVector`, and `aijMatrix` are those of the tables within the spreadsheet *RM5.xls*. These names define the ranges in the spreadsheet that correspond to the respective data tables.[6] `"ODBC"` is the standard data-handling interface for the spreadsheet. A `read table` then inputs the data to the model (see file *RM5.txt*). Note the use of COL1 and COL2 in table `profitVector`, which correspond to the (arbitrary) column names in the spreadsheet. The syntax is the same as in input tables (Section C.5.3). Each data table of the model may be stored in a separate sheet, if desired.

As in Section C.5.3, two-dimensional data can be read in an array format using the following table definition:

```
table aijArray IN "ODBC" "RM5.xls":[i~resource], {j in paint}<aij[i, j]~(j)>;
```

In this case, the array `aij` appears in the range `aijArray` of *RM5.xls* and must include the proper row and column headings. *It is also important to remember that numeric column headings when used in the table must be converted to strings by using Excel* `TEXT` *function, else AMPL will issue some undecipherable error messages.*

The same table declaration can be used to export output data to a spreadsheet. The only difference is to replace `IN` with `OUT`, exactly as in the case of table files. In this case, a `write table` command (following the `solve` command) will send the output to the spreadsheet. The following examples demonstrate the use of `OUT` tables:

```
table variables OUT "ODBC" "RM5a.xls":
      [paint], product~solution, product.rc~reducedCost;
```

[6] To name a range, highlight it and type its name in the "name box" to the left of the Excel formula bar, then click Enter, or use Excel's Insert/Names/Define.

```
table constraints OUT "ODBC" "RM5a.xls":
      [resource], limit.slack~slack, limit.dual~dual;
```

The output tables `variables` and `constraints` will go to Excel file *RM5a.xls* following the execution of the `write table` command, each appearing automatically in the northwest corner of a separate sheet.

C.6 INTERACTIVE COMMANDS

AMPL allows the user to solve the model interactively and to check/modify data and retrieve output to the screen or to a file. The following is a partial list of a number of useful commands:

`delete` *comma-separated names of objective function and constraints;*
`drop` *comma-separated names of objective function and constraints;*
`restore` *comma-separated names of objective function and constraints;*
`display` *comma-separated item_list;*
`print/printf` *unformatted/formatted item_list;*
`expand` *comma-separated names of objective function and constraints;*
`let` *parameter or variable (indexed or nonindexed):= value;*
`fix` *variable (indexed or nonindexed):= value;*
`unfix` *variable (indexed or noindexed);*
`reset;`
`reset data;`
`solve;`

Such commands are entered interactively at the `ampl:` prompt. Some, such as `display` and `print`, may appropriately be hard-coded in the model, if desired.

The `delete` command completely removes the listed objective function and/or constraints, whereas `drop` temporarily yanks them out of the model. The `drop` command may be annulled by the `restore` command. A new objective function or constraint may be added to the model by entering it from the keyboard, exactly as we do in a hard-coded model. (See Example 9.2-1 for an application to the B&B algorithm.)

We have used `display` with the Reddy Mikks model. The output may be directed to an external file using >*filename* immediately before the terminating semicolon. Else, the output defaults to the screen.

The `print/printf` command has been discussed earlier in Section C.5.2. The output defaults to the screen, or it may be directed to an output file as in `display`.

The `expand` command provides a longhand representation of the objective function and the constraints. For example, in the Reddy Mikks model, the command

```
expand profit;
```

prints out the objective function as

```
maximize profit:5*product["exterior"]+4*product["interior"];
```

In this manner, the user can see if the model has retrieved the input data correctly. In a similar manner, the command

```
expand limit;
```

will expand all the constraints of the model. If you are interested in a specific constraint, then `limit` must be properly indexed. For example,

```
expand limit["m1"];
```

will display the first constraint of the model.

The `let` command allows entering new values of parameters and variables (using : = as assignment operator). The right-hand side may be a simple numeric value or a mathematical expression. It is used to test different solution scenarios as we will show in Section C.7.

The `fix` command is used to assign a specific value to a variable prior to solving the model. For example, suppose that the following statements are issued interactively prior to solving the Reddy Mikks model.

```
ampl: fix product["exterior"]:=1.5;
ampl: solve;
```

With these commands, AMPL solves the problem with the added restriction `product["exterior"]` = 1.5. The change caused by `fix` can be undone by issuing the `unfix` command as

```
ampl: unfix product["exterior"];
```

The `fix`/`unfix` commands can be useful in experimenting with the model when some of the variables are either eliminated (=0) or held constant. (See *AMPL moment* following Example 9.3-4 for an application to the traveling salesperson problem.)

The command `reset` removes all reference to the current model from AMPL. A fresh `model` command will thus be necessary to restart the model. Also, the command `reset data;` will delete all the data of the model. Specific data elements may be selectively removed by listing them after the command. For example, **`reset data`** `a b c`; will delete the values of the parameters `a`, `b`, and `c`.

It is important to note that when two AMPL models are executed in the same session, the command `reset;` must separate the execution of successive models. For example, the two AMPL models *a1.txt* and *a2.txt* are executed as follows:

```
ampl: model a1.txt;
ampl: reset; model a2.txt;
```

If `reset;` is not used, AMPL will spew an enormous list of undecipherable errors.

There is a large number of useful interactive commands in AMP, but their detailed presentation is beyond the scope of this abridged presentation.

C.7 ITERATIVE AND CONDITIONAL EXECUTION OF AMPL COMMANDS

Suppose in the Reddy Mikks model we are interested in studying the sensitivity of the optimal solution to changes in specific parameters. For example, in file *RM2.txt*, how is the optimal solution affected when the availability of raw material m1 (`=rhs["m1"]`) is changed from its current value of 24 tons to the new values of 27 and 30 tons? After executing *RM2.txt* and getting the solution for rhs`["m1"]=24`, we can enter the following statements interactively:

```
ampl: let rhs["m1"]:=27;
ampl: solve;
ampl: display profit, product;
```

The output will be displayed on the screen (it can also be sent to a file, if desired, as we explained earlier). To secure results for rhs`["m1"]=30`, the same statements are repeated with the `let` statement specifying the new value. This, however, is not the most efficient way to do the task.

AMPL allows building convenient `commands` files that will eliminate the unnecessary chore of retyping commands. Specifically, for the present example, a command file (which we arbitrarily name *cmd.txt*) may have the following statements:

```
for (i in 1..2}
   {
   let rhs["m1"]:=rhs["m1"]+3;
   solve;
   display profit, product;
   }
```

Following the execution of the model (with `rhs["m1"]=24`), we can execute the remaining two cases by entering

```
ampl: commands cmd.txt;
```

Of course, we can modify *cmd.txt* to include `rhs["m1"]=24` as well. See Problem 1, Set C.7a.

We can use the statement `repeat while` *`condition`*`{...};` or `repeat until` *`condition`*`{...};` to replace `for{...}` as follows:

```
repeat while rhs["m1"]<=30
{
let rhs["m1"]:=rhs["m1"] +3;
solve;
display profit, product;
};
```

Alternatively, we may use

```
repeat until rhs["m1"]>30
{
let rhs["m1"]:=rhs["m1"]+3;
```

```
solve;
display profit, rhs["m1"], product;
};
```

Note that repeat `while` will loop so long as the condition is true, whereas `repeat` until will loop so long as the condition is false.

Another useful statement in commands file is `if-then-else`. In this case, `if` may be followed by any legitimate condition, whereas `then` and `else` can be followed only by command statements. With the `if` statement, AMPL commands `continue;` and `break;` may be used within the loop construct to either skip to the next index of the loop or exit the loop altogether.

Example 9.3-5 (Figure 9.14) provides a good illustration of the use of the loop and conditional statements to print formatted output.

PROBLEM SET C.7A

1. Modify *RM2.txt* so that `rhs["m1"]` will assume the values 20 to 35 tons in steps of 5 tons. All `solve` commands must be executed from within the command file *cmd.txt* in the following manner:

```
ampl: model RM2.txt;
ampl: commands cmd.txt;
```

The command file *cmd.txt* is developed using the three different versions to construct the loop:

(a) `for{}.`
(b) `repeat while{};.`
(c) `repeat until{};.`

C.8 SENSITIVITY ANALYSIS USING AMPL

We have seen previously how the dual values and the reduced costs can be determined in an AMPL LP model by using the *ConstraintName* `.dual` and *VariableName* `.rc` in the `display` command. To complete the standard LP sensitivity analysis report, AMPL additionally provides facilities for the determination of the optimality ranges for the objective-function coefficients and the feasibility ranges for the (constant) right-hand sides of the constraints. We will use file *RM2.txt* (see Figure C.3) to demonstrate how AMPL generates the sensitivity analysis report.

In the model in Figure C.3, replace the `solve` and `display` statements with

```
option solver cplex;
option cplex_options 'sensitivity';
solve;
display limit.down, limit.current, limit.up, limit.dual;
display product.down, product.current, product.up, product.rc;
```

The output can be directed to a file if desired (see file *RM6.txt*). The two `option` statements must precede the `solve` command. The first `display` command provides the

feasibility ranges for all the constraints (named `limit` in the model). The suffixes `.down`, `.current`, and `.up` give the lower, current, and upper values, respectively, for the right-hand side of each member constraint. In a similar manner, the second `display` command provides the optimality ranges for the objective-function coefficients. The following output is self-explanatory.

```
profit = 21
         product.down  product.current  product.up  product.rc
exterior    2                5             6          0
interior    3.33333          4            10          0

        limit.down  limit    limit.up
demand     -1.5       0        1e+20
m1         20         0.75     36
m2          4         0.5       6.66667
market      0         0         0
```

C.9 SELECTED AMPL MODELS

This section fully develops AMPL models for a number of examples used throughout the book. For convenience, the models are categorized by chapter. In addition to demonstrating the use of AMPL programming facilities given in Sections C.1 through C.8 in real applications, the examples also serve to introduce new AMPL features.

Chapter 2

The bus scheduling problem (Example 2.4-5). The constraints of the bus scheduling problem have a special structure that allows formulating the left-hand side of the constraints in a compact generalized formulation. Specifically, the left-hand side of constraint 1 can be written as $x_1 + x_m$, where m is the total number of periods in a 24-hour day (=6 in the present example). For the remaining constraints, the left-hand side takes the form $x_{i-1} + x_i, i = 2, 3, \ldots, m$. Using `if...then...else` (as explained in Section C.3), all m constraints can be represented compactly as given in Figure C.7 (file *BusSched.txt*).

Chapter 5

Transportation model (Example 5.3-1). Figure C.8 provides the AMPL model for the transportation model of Example 5.3-1 (file *TansportA.txt*). The names used in the model are self-explanatory. Both the constraints and the objective function follow the format of the LP model presented in Example 5.1-1.

The model uses the sets `sNodes` and `dNodes` to conveniently allow the use of the alphanumeric set members `{S1, S2, S3}` and `{D1, D2, D3, D4}`, which are entered in the data section. All the input data are then entered in terms of these set members as shown in Figure C.8.

Although the alphanumeric code for set members is more readable, generating them for large problems may not be convenient. File *TransportB.txt* shows how the same sets can be defined as `{1..m}` and `{1..n}`, where m and n represent the number of sources and the number of destinations. By simply assigning numeric values for `m` and `n`, the sets are automatically defined for any size model.

```
param m;
param min_nbr_buses{1..m};
var x_nbr_buses{1..m} >= 0;
minimize tot_nbr_buses: sum {i in 1..m} x_nbr_buses[i];
subject to constr_nbr{i in 1..m}:
      if i=1 then
          x_nbr_buses[i]+x_nbr_buses[m]
      else
          x_nbr_buses[i-1]+x_nbr_buses[i] >= min_nbr_buses[i];

data;
param m:=6;
param min_nbr_buses:= 1 4  2 8  3 10  4 7  5 12  6 4;

solve;
display tot_nbr_buses, x_nbr_buses;
```

FIGURE C.7

AMPL model of the bus scheduling problem of Example 2.4-5 (file *BusSched.txt*)

FIGURE C.8

AMPL model of the transportation model of Example 5.3-1 (File *TransportA.txt*)

```
#----- Transporation model (Example 5.3-1)-----
    set sNodes;
    set dNodes;
    param c{sNodes,dNodes};
    param supply{sNodes};
    param demand{dNodes};
    var x{sNodes,dNodes}>=0;
    minimize z:sum {i in sNodes,j in
    dNodes}c[i,j]*x[i,j];
    subject to
    source{i in sNodes}:sum{j in dNodes}x[i,j]=supply[i];
    dest{j in dNodes}:sum{i in sNodes}x[i,j]=demand[j];
    data;
    set sNodes:=S1 S2 S3;
    set dNodes:=D1 D2 D3 D4;
    param c:
    D1  D2  D3  D4  :=
    S1  10  2   20 11
    S2  12  7   9  20
    S3  4   14  16 18;
    param supply:= S1 15 S2 25 S3 10;
    param demand:= D1 5 D2 15 D3 15 D4 15;
    solve;display z, x;
```

The data of the transportation model can be retrieved from a spreadsheet (file *TM.xls*) using the AMPL `table` statement. File *TansportC.txt* provides the details. To study this model, you will need to review the material in Section C.5.5.

Chapter 6

Figure C.9 provides the AMPL model for solving Example 6.3-6 (file *ShortestRouteA.txt*). The variable `x[i, j]` assumes the value 1 if arc `[i, j]` is on the shortest route and 0 otherwise. The model is general in the sense that it can be used to find the shortest route between any two nodes in a problem of any size.

As explained in Example 6.3-6, AMPL treats the problem as a network in which an external flow unit enters and exits at specified `start` and `end` nodes. The main input

FIGURE C.9
AMPL shortest route model (file *ShortestRouteA.txt*)

```
#--------- shortest route model (Example 6.3-6)---------
param n;
param start;
param end;
param M=999999; #infinity
param d{i in 1..n, j in 1..n} default M;
param rhs{i in 1..n}=if i=start then 1
                  else (if i=end then -1 else 0);
var x{i in 1..n,j in 1..n}>=0;
var outFlow{i in 1..n}=sum{j in 1..n}x[i,j];
var inFlow{j in 1..n}=sum{i in 1..n}x[i,j];

minimize z: sum{i in 1..n, j in 1..n}d[i,j]*x[i,j];
subject to limit{i in 1..n}:outFlow[i]-inFlow[i]=rhs[i];

data;
param n:=5;
param start:=1;
param end:=2;
param d:
   1  2    3   4   5:=
1  .  100  30  .   .
2  .  .    20  .   .
3  .  .    .   10  60
4  .  15   .   .   50
5  .  .    .   .   .;
solve;
print "Shortest length from",start,"to",end,"=",z;
printf "Associated route: %2i",start;
for {i in 1..n-1} for {j in 2..n}
  {if x[i,j]=1 then printf" - %2i",j;} print;
```

data of the model is an $n \times n$ matrix representing the distance `d[i, j]` of the arc joining nodes `i` and `j`. Per AMPL syntax, a dot entry in `d[i, j]` is a placeholder that signifies that no distance is specified for the corresponding arc. In the model, the dot entry is overridden by the infinite distance M (= 999999) in

```
param d{i in 1..n, j in 1..n}default M;
```

The constraints represent flow conservation through each node:

$$\text{(Input flow)} - \text{(Output flow)} = \text{(External flow)}$$

From `x[i, j]`, we can define the input and output flow for node *i* using the statements

```
var inFlow{j in 1..n}=sum{i in 1..n}x[i, j];
var outFlow{i in 1..n}=sum{j in 1..n}x[i, j];
```

The left-hand side of the constraint *i* is thus given as `outFlow[i]-inFlow[i]`.

The right-hand side of constraint *i* (external flow at node *i*) is defined as

```
param rhs{i in 1..n}=if i=start then 1 else(if i=end then -1 else 0);
```

(See Section C.3 for details of `if...then...else`.) With this statement, specifying `start` and end nodes automatically assigns `1, -1, or 0` to rhs, the right-hand side of the constraints. This statement allows finding the shortest distance between any two nodes in the network.

The objective function seeks the minimization of the sum of `d[i, j]*x[i, j]` over all `i` and `j`.

In the present example, `start=1` and `end=2`, meaning that we want to determine the shortest route from node 1 to node 2. The associated output is

```
Shortest length from 1 to 2 =55
Associated route: 1 - 3 - 4 - 2
```

Remarks. The AMPL model as given in Figure C.9 has one flaw: The number of *active* variables x_{ij} is n^2, which could be significantly much larger than the actual number of (positive-distance) arcs in the network, thus resulting in a much larger LP. The reason is that the model accounts for the nonexisting arcs by assigning them an infinite distance M (= 999999) to guarantee that they will be zero in the optimum solution.

The situation can be remedied by using a subset of `{i in 1..n, j in 1..n}` that excludes nonexisiting arcs, as the following statement shows:

```
var x{i in 1..n, j in 1..n:d[i, j]<M}>=0;
```

(See Section C.4 for the use of conditions to define subsets.) The same logic must be applied to the constraints as well by using the following statements:

```
var inFlow{j in 1..n}=sum{i in 1..n:d[i, j]<M}x[i, j];
var outFlow{i in 1..n}=sum{j in 1..n:d[i, j]<M}x[i, j];
```

File *ShortestRouteB.txt* gives the complete model.

Maximal flow model (Example 6.4-2). Figure C.10 provides the AMPL model for the maximal flow problem. The data applies to Example 6.4-2 (file *MaxFlow.txt*). The overall idea of determining the input and output flows at a node is similar to the one detailed for the shortest-route model. However, because the model is designed to find the maximum flow between *any* two nodes, `start` and `end`, two additional constraints are needed to ensure that no external flow *enters* `start` and no external flow *leaves* `end`. Constraints `inStart` and `outEnd` in the model ensure this result. These two constraints are not needed when `start=1` and `end=5` because the nature of the data guarantees the desired

FIGURE C.10
AMPL model of the maximal flow problem of Example 6.4-2 (file *MaxFlow.txt*)

```
#---------- Maximal Flow model (Example 6.4-2)----------
param n;
param start;
param end;
param c{i in 1..n, j in 1..n} default 0;

var x{i in 1..n,j in 1..n:c[i,j]>0}>=0,<=c[i,j];
var outFlow{i in 1..n}=sum{j in 1..n:c[i,j]>0}x[i,j];
var inFlow{i in 1..n}=sum{j in 1..n:c[j,i]>0}x[j,i];

maximize z: sum {j in 1..n:c[start,j]>0}x[start,j];
subject to
limit{i in 1..n:
      i<>start and i<>end}:outFlow[i]-inFlow[i]=0;
inStart:sum{i in 1..n:c[i,start]>0}x[i,start]=0;
outEnd:sum{j in 1..n:c[end,j]>0}x[end,j]=0;

data;
param n:=5;
param start:=1;
param end:=5;
param c:
   1  2   3   4   5  :=
1  .  20  30  10  0
2  .  .   40  0   30
3  .  .   0   10  20
4  .  .   5   .   20
5  .  .   .   .   .;

solve;
print "MaxFlow between nodes",start,"and",end, "=",z;
printf "Associated flows:\n";
for {i in 1..n-1} for {j in 2..n:c[i,j]>0}
    {if x[i,j]>0 then
    printf"(%2i-%2i)=%5.2f\n",i,j,x[i,j];} print;
```

result. However, for `start =3`, node 3 allows both input and output flow (arcs 4-3 and 3-4) and, hence, constraint `inStart` is needed (try the model without `inStart`!).

The objective function maximizes the sum of the output flow at node `start`. Equivalently, we can choose to maximize the sum of the input flow at node `end`.

CPM model (Example 6.5-2). Figure C.11 provides the AMPL model for any CPM network (file *CPM.txt*). The model is driven by the data of Example 6.5-2. It makes use

FIGURE C.11
AMPL model for Example 6.6-2 (file *CPM.txt*)

```
------------------------ CPM (Example 6.5-2)------------------------
param n;
param D{1..n,1..n} default -1;
set into{1..n};
set from{1..n};
var x{i in 1..n,j in from[i]}>=0;
var ET{i in 1..n};
var LT{i in 1..n};
var TF{i in 1..n, j in from[i]};
var FF{i in 1..n, j in from[i]};
data;
param n:=6;
param D: 1 2 3 4 5 6:=
1  .  5  6  .  .  .
2  .  .  .  3  8  .
3  .  .  .  .  2  11
4  .  .  .  .  0  1
5  .  .  .  .  .  12
6  .  .  .  .  .  .;
for {i in 1..n} {let from[i]:={j in 1..n:D[i,j]>=0}};
for {j in 1..n} {let into[j]:={i in 1..n:D[i,j]>=0}};
------------nodes earliest and latest times and floats
let ET[1]:=0;            #earliest node time
for {i in 2..n}let ET[i]:=max{j in into[i]}(ET[j]+D[j,i]);
let LT[n]:=ET[n];  #latest node time
for {i in n-1..1 by -1}let LT[i]:=min{j in from[i]}(LT[j]-D[i,j]);
printf "%1s-%1s %5s %5s %5s %5s %5s %5s %5s \n\n",
   "i","j","D","ES","EC","LS","LC","TF","FF" >Ex6.6-2out.txt;
for {i in 1..n, j in from[i]}
{
let TF[i,j]:=LT[j]-ET[i]-D[i,j];
let FF[i,j]:=ET[j]-ET[i]-D[i,j];
printf "%1i-%1i %5i %5i %5i %5i %5i %5i %5i %3s\n",
   i,j,D[i,j],ET[i],ET[i]+D[i,j],LT[j]-D[i,j],LT[j],TF[i,j],FF[i,j],
   if TF[i,j]=0 then "c" else "" >Ex6.6-2out.txt;
}
```

of *indexed sets* (see Section C.4) and requires no optimization. In essence, no `solve` command is needed, and AMPL is implemented as a pure programming language similar to Basic or C.

The nature of the computations in CPM requires representing the network by associating two *indexed* sets with each node: `into` and `from`. For node `i`, the set `into[i]` defines all the input nodes that feed into node `i`, and the set `from[i]` defines all the output nodes that are reached from node `i` (recall that in CPM all the arcs are *directional*, hence it makes sense to speak of input and output nodes). For example, in Example 6.5-2, `from[1]={2, 3}`, and `into[1]` is empty.

The determination of subsets `from` and `into` is achieved in the model as follows: Because `D[i, j]` can be zero when a CPM network uses dummy activities, the default value for `D[i, j]` is `-1` for all nonexisting arcs. Thus, the set `from[i]` represents all the nodes `j` in the set `{1..n}` that can be reached from node `i`, which can happen only if `D[i, j]>=0`. This says that `from[i]` is defined by the subset `{j in 1..n:D[i, j]>=0}`. Similar reasoning applies to the determination of subsets `into[i]`. The following AMPL statements automate the determination of these sets and must follow the `D[i, j]` data, as shown in Figure 6.48:

```
for {i in 1..n} {let from[i]:={j in 1..n:D[i, j]>=0}};
for {j in 1..n} {let into[j]:={i in 1..n:D[i, j]>=0}};
```

Once the sets `from` and `into` have been determined, the model goes through the forward pass to compute the earliest time, `ET[i]`. With the completion of this pass, we can initiate the backward pass by using

```
let LT[n]:=ET[n];
```

The rest of the model is needed to obtain the output shown in Figure C.12. This output determines all the data needed to construct the CPM chart. The logic of this segment is based on the computations given in Examples 6.5-2 and 6.5-4.

FIGURE C.12

Output of AMPL model for Example 6.5-2 (file *CPM.txt*)

```
i-j     D     ES     EC     LS     LC     TF     FF
1-2     5      0      5      0      5      0      0    c
1-3     6      0      6      5     11      5      2
2-3     3      5      8      8     11      3      0
2-4     8      5     13      5     13      0      0    c
3-5     2      8     10     11     13      3      3
3-6    11      8     19     14     25      6      6
4-5     0     13     13     13     13      0      0    c
4-6     1     13     14     24     25     11     11
5-6    12     13     25     13     25      0      0    c
```

Chapter 8

Preemptive goal programming mode (Example 8.2-2). AMPL lends itself readily to applying the idea presented in Example 8.2-2, where constraints are added to ensure that higher-priority solutions are not degraded. Figure C.13 provides a generic AMPL code that allows the application of the preemptive method interactively (file *GoalProg.txt*).

The design of the model is standard except for the provisions that allow applying the preemptive method interactively. Specifically, the model assumes that the first r constraints are goal constraints and the remaining $m - r - 1$ are strict constraints. The model has r distinct goal objective functions, which can be included in the same model by using the following indexed AMPL statement (only an *indexed* name is allowed for multiple objective functions):

```
minimize z{i in 1..r}:p*sminus[i]+q*splus[i];
```

FIGURE C.13

AMPL model for interactive application of the preemptive method (file *GoalProg.txt*)

```
param n;
param m;
param r;
param p;
param q;
param a{1..m,1..n};
param b{1..m};
param um{1..m} default 100000;
param up{1..m} default 100000;
var x{1..n} >=0;
var sminus{i in 1..r}>=0,<=um[i];
var splus{i in 1..r}>=0,<=up[i];

minimize z{i in 1..r}: p*sminus[i]+q*splus[i];
subject to
c1{i in 1..r}:
  sum{j in 1..n}a[i,j]*x[j]+sminus[i]-splus[i]=b[i];
c2{i in r+1..m}: sum{j in 1..n} a[i,j]*x[j]<=b[i];
data;
param m:=4;
param n:=4;
param r:=4;
param a:  1       2       3     4:=
      1   550    35      55     .075
      2   55    -31.5     5.5   .0075
      3   110     7     -44     .015
      4   0       0       0     1;
param b:=1 16 2 0 3 0 4 2;
```

The given definition of the objective function accounts for minimizing $z_i = s_i^-$ and $z_i = s_i^+$ by setting $(p = 1, q = 0)$ and $(p = 0, q = 1)$, respectively.

Instead of adding a new constraint each time we move from one priority level to the next, we use a programming trick that allows modifying the upper bounds on the deviational variables. The parameters `um[i]` and `up[i]` represent the upper bounds on `sminus[i]` (s_i^-) and `splus[i]` (s_i^+), respectively. These parameters are modified to impose implicit constraints of the types `sminus[i]<=um[i]` and `splus[i]<=up[i]`, respectively. The values of `um` and `up` in priority goal i are determined from the solutions of the problems of priority goals 1, 2, and $i - 1$. The initial (default) value for um and up is infinity.

We will show shortly how AMPL activates any of the `r` objective functions, specifies the values of p and q, and sets the upper limits on s_i^- and s_i^+, all interactively, which makes AMPL ideal for carrying out goal programming computations.

Using the data of Example 8.1-1, the goals of the model are

$$\text{Minimize } G_1 = s_1^-$$

$$\text{Minimize } G_2 = s_2^-$$

$$\text{Minimize } G_3 = s_3^-$$

$$\text{Minimize } G_4 = s_4^+$$

Suppose that the goals are prioritized as

$$G_2 > G_1 > G_3 > G_4$$

The implementation of AMPL model thus proceeds in the following manner: For G_2, set $p = 1$ and $q = 0$ because we are minimizing `z[2]=sminus[2]`. The following commands are used to carry out the calculations:

```
ampl: model GoalProg.txt;
ampl: let p:=1;let q:=0;
ampl: objective z[2];
ampl: solve; display z[2], x, sminus, splus;
```

These commands produce the following results:

```
z[2] =0
:       x       sminus  splus :=
1  2.62361e-28    16    0
2  0               0    0
3  8.2564e-28      0    0
4  0               2    0
```

The solution ($x_1 = 0, x_2 = 0, x_3 = 0$, and $x_4 = 0$) shows that goal G_2 is satisfied because `z[2] = 0` (that is, $s_2^- = 0$). However, the next-priority goal, G_1, is not satisfied because $s_1^- = 16$. Hence, we need to optimize goal G_1 without degrading the solution of G_2. This requires changing the upper bounds on s_2^- to the value specified by the solution of G_2—namely, zero. For goal G_1,

current $p = 1$ and $q = 0$ from G_2 remain unchanged because we are minimizing s_1^-. The following interactive AMPL commands achieve this result:

```
ampl: let um[2]:=0;
ampl: objective z[1]; solve; display z[1], x, sminus, splus;
```

The ouput is

```
z[1] =0
:       x       sminus splus  :=
1  0.0203636       0    0
2  0.0457143       0    0
3  0.0581818       0    0
4  0               2    0
```

The solution shows that all the remaining goals are satisfied. Hence, no further optimization is needed. The goal programming solution is $x_p = .0203636$, $x_f = .0457143$, $x_s = .0581818$, and $x_g = 0$.

Remarks.

1. We can replace let `um[2]:=0;` with either `fix sminus[2]:=0;` or `let sminus[2]:=0;` with equal end result.
2. The interactive session can be totally automated using a commands file that automatically selects the current goal to be optimized and imposes the proper restrictions before solving the next priority goal. The use of this file (named *amplCmds.txt*) requires making some modifications in the original model as shown in file *GolaProgA.txt*. To be completely versatile, the data of the model are stored in a separate file named *amplData.txt*. In this case, the execution of the model requires issuing two command lines:

   ```
   ampl: model amplEx8.1-1A.txt;
   ampl: data amplData.txt;
   ampl: commands amplCmds.txt;
   ```

See Section C.7 for more information about the use of commands.

Chapter 9

Set covering model (Example 9.1-2). Figure C.14 presents a general AMPL model for any set-covering problem (file *SetCovering.txt*). The formulation is straightforward, once the use of *indexed set* is understood (see Section C.4). The model defines `street` as a (regular) set whose elements are A through K. Next, the *indexed* set `corner{street}` defines the corners as a function of street. With these two sets, the constraints of the model can be formulated directly. The data of the model give the elements of the indexed sets that are specific to the situation in Example 9.1-2. Any other situation is handled by changing the data of the model.

```
#------------------ Example 9.1-2------------------
param n;     #maximum number of corners
set street;
set corner{street};
var x{1..n}binary;
minimize z: sum {j in 1..n} x[j];
subject to limit {i in street}:
      sum {j in corner[i]} x[j]>=1;
data;
param n:=8;
set street:=A B C D E F G H I J K;
set corner[A]:=1 2;
set corner[B]:=2 3;
set corner[C]:=4 5;
set corner[D]:=7 8;
set corner[E]:=6 7;
set corner[F]:=2 6;
set corner[G]:=1 6;
set corner[H]:=4 7;
set corner[I]:=2 4;
set corner[J]:=5 8;
set corner[K]:=3 5;

option solver cplex;
solve;
display z,x;
```

FIGURE C.14

General AMPL model for the set covering problem of Example 9.1-2 (file *SetCovering.txt*)

Job sequencing model (Example 9.1-4). File *JobSeq.txt* provides the AMPL model for the problem of Example 9.1-4. The model in Figure C.15 is self-explanatory because it is a direct translation of the general mathematical model. It can handle any number of jobs by changing the input data. Note that the model is a direct function of the raw data: processing time `p`, due date `d`, and delay penalty `perDayPenalty`.

Chapter 13

General multi-item EOQ (Example 13.3-3). The AMPL nonlinear model for the general multi-item EOQ with storage limitation is given in Figure C.16 (file *ConstrEOQ.txt*). The model follows the same rules used in solving linear programs. However, AMPL nonlinear models exhibit peculiarities that may impede reaching a solution. In particular, "judicious" initial values must be specified for the variables. In Figure C.16, the definition statement

```
var y{1..n}>=0, :=10;  #initial trial value = 10;
```

```
#------------------ Example 9.1-4------------------
param n;
set I={1..n};
set J={1..n};     #I is the same as J
param p{I};
param d{I};
param perDayPenalty{I};
param M=1000;
var x{J}>=0;              #continuous
var y{I,J} binary;        #0-1
var sMinus{J}>=0;         # s=sMinus-sPlus
var sPlus{J}>=0;
minimize penalty: sum {j in J}
           perDayPenalty[j]*sPlus[j];
subject to
eitherOr1{i in I,j in J:i<>j}:
         M*y[i,j]+x[i]-x[j]>=p[j];
eitherOr2{i in I,j in J:i<>j}:
         M*(1-y[i,j])+x[j]-x[i]>=p[i];
dueDate{j in J}:x[j]+sMinus[j]-sPlus[j]=d[j]-p[j];
data;
param n:=3;
param p:= 1 5  2 20  3 15;
param d:= 1 25  2 22  3 35;
param perDayPenalty := 1 19  2 12  3 34;
option solver cplex; solve;
display penalty,x;
```

FIGURE C.15

AMPL model of the job sequencing problem of Example 9.1-4 (file *JobSeq.txt*)

includes the code `:=10` that assigns the initial value 10 to all the variables. If you use an initial value of 1 in the present example, division by zero will result during the iterations. Thus, you may need to replace K_iD_i/y_i with $K_iD_i/(y_i + \Delta)$, $\Delta > 0$ and very small, to prevent division by zero during the iterative process. Indeed, Problems 1 and 4, Set 13.3c, could not be solved with AMPL without invoking this trick.

Chapter 21

Modeling nonlinear problems (Example 21.2-2). Figure C.17 gives the AMPL model (file *NLP.txt*). The only deviation from LP (other than the nonlinearity, of course) is that you may need to specify "appropriate" initial values for the variables to get the solution iteration to converge. In Figure C.17, the arbitrary initial values $x_1 = 10$ and $x_2 = 10$ are specified by appending `:=10` and `:=10` to the definition of the two variables. If you do not specify initial values at all, AMPL will not reach the optimum solution and will print the message "too many major iterations." Although a solution is given in this case, it

```
param n;
param K{1..n};
param D{1..n};
param h{1..n};
param a{1..n};
param A;
var y{1..n}>=0, :=10;  #initial trial value = 10
minimize z: sum{j in 1..n}(K[j]*D[j]/y[j]+h[j]*y[j]/2);
subject to storage:sum{j in 1..n}a[j]*y[j]<=A;
data;
param n:=3;
param K:= 1 10 2 5 3 15;
param D:=1 2 2 4 3 4;
param h:=1 .3 2 .1 3 .2;
param a:=1 1 2 1 3 1;
param A:=25;
solve;display z,y;
printf{"SOLUTION:\n"}>a.out;
printf{" Total cost = %4.2\n"},z>a.out;
for {i in 1..n}
       printf{" y%1i = %4.2f\n"},i,y[i]>a.out;
```

FIGURE C.16
AMPL model for Example 13.3-3 (file *ConstrEOQ.txt*)

```
var x1>=0 :=10; #inital value = 10
var x2>=0 :=10; #initial value = 10

maximize z: x1-x2;
subject to c1: 3*x1^4+x2<=243;
subject to c2: x1+2*x2^2<=32;
subject to c3: x1>=2.1;
subject to c4: x2>=3.5;

solve;
display z,x1,x2;
```

FIGURE C.17
AMPL model of Example 21.2-2 (file *NLP.txt*)

usually is not correct. In essence, the most logical way to deal with a nonlinear problem is to specify different initial values for the variables and then decide if a consensus can be reached regarding the optimum solution.

BIBLIOGRAPHY

Fourer, R., D. Gay, and B. Kernighan, *AMPL, A Modeling Language for Mathematical Programming*, 2nd ed., Brooks/Cole-Thomson, Pacific Grove, CA, 2003.