ROY, A., L.S. LASDON AND J. LORDEMAN. 1986. Extending Planning Languages to Include Optimization Capabilities. *Management Science* **32,** 360–373.

SCHRAGE, L. 1989. *User's Manual for Linear, Integer and Quadratic Programming with LINDO,* fourth edition. Scientific Press, Redwood City, CA.

SCHRAGE, L. AND K. CUNNINGHAM. 1988. Demo LINGO/PC: Language for INteractive General Optimization, version 1.04a. LINDO Systems Inc., Chicago.

SIMONS, R.V. 1987. Mathematical Programming Modeling Using MGG. *IMA Journal of Mathematics in Management* **1,** 267–276.

STROUSTRUP, B. 1986. *The C++ Programming Language.* Addison-Wesley, Reading, MA.

References

Baker, T.W. and D.J. Biddle. 1986. A Hierarchical/Relational Approach to Modeling. Chesapeake Decision Sciences, Inc.; presented at the ORSA/TIMS Joint National Meeting, Miami Beach.

Beale, E.M.L. 1970. Matrix Generators and Output Analyzers. In H.W. Kuhn (ed.), *Proceedings of the Princeton Symposium on Mathematical Programming,* Princeton University Press, Princeton, NJ, pp. 25–36.

Bisschop, J. and A. Meeraus. 1982. On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study* **20,** 1–29.

Brooke, A., D. Kendrick and A. Meeraus. 1988. *GAMS: A User's Guide.* Scientific Press, Redwood City, CA.

Choksi, A.M., A. Meeraus and A.J. Stoutjesdijk. 1980. *The Planning of Investment Programs in the Fertilizer Industry.* Johns Hopkins University Press, Baltimore.

Creegan, J.B. 1985. DATAFORM, a Model Management System. Ketron, Inc., Arlington, VA.

Dolk, D.R. 1986. A Generalized Model Management System for Mathematical Programming. *ACM Transactions on Mathematical Software* **12,** 92–126.

Fourer, R. 1983. Modeling Languages versus Matrix Generators for Linear Programming. *ACM Transactions on Mathematical Software* **9,** 143–183.

Fourer, R., J.B. Gertler and H.J. Simkowitz. 1977. Models of Railroad Passenger-Car Requirements in the Northeast Corridor. *Annals of Economic and Social Measurement* **6,** 367–398.

Fourer, R., J.B. Gertler and H.J. Simkowitz. 1978. Optimal Fleet Sizing and Allocation for Improved Rail Service in the Northeast Corridor. *Transportation Research Record* **656,** 40–45.

General Optimization, Inc. 1986. *What's Best!* Holden-Day, Oakland, CA.

Geoffrion, A.M. 1988. SML: A Model Definition Language for Structured Modeling. Working Paper No. 360, Western Management Science Institute, University of California, Los Angeles.

Haverly Systems, Inc. 1977. MaGen. Denville, NJ.

Kendrick, D., A. Meeraus and J.S. Suh. 1981. Oil Refinery Modeling with the GAMS Language. Research Report No. 14, Center for Energy Studies, University of Texas, Austin.

Kernighan, B.W. and R. Pike. 1984. *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs, NJ.

Ketron, Inc. 1986. PAM: a Practitioner's Approach to Modeling. Arlington, VA.

Management Science Systems. 1970. DATAFORM Mathematical Programming Data Management System: User Manual. Ketron, Inc., Arlington, VA.

Murtagh, B.A. 1981. *Advanced Linear Programming: Computation and Practice.* McGraw-Hill, New York.

Murtagh, B.A. and M.A. Saunders. 1987. MINOS 5.1 User's Guide. Technical Report SOL 83–20R, Systems Optimization Laboratory, Department of Operations Research, Stanford University.

```
             30 32   5.40     31 33   8.24     32 34   4.84
             33 35   7.44     34 36   4.44     35 37   6.80
             36 38   4.11     37 39   6.25     38 40   3.69
             39 41   5.55     40 42   3.29     41 43   4.77
             42 44   2.91     43 45   4.19     44 46   2.53
             45 47   4.00     47 1    1.65

[NY,*,BO,*]  10 16   1.23     12 18   3.84     14 20   4.08
             15 21   1.47     16 22   2.96     17 23   1.60
             18 24   2.95     19 25   1.71     20 26   2.81
             21 27   1.77     22 28   2.87     23 29   1.84
             24 30   2.95     25 31   1.91     26 32   3.12
             27 33   1.93     28 34   3.31     29 35   2.00
             30 36   3.40     31 37   2.08     32 38   3.41
             33 39   2.69     34 40   4.45     35 41   2.32
             36 42   3.40     37 43   1.80     38 44   2.63
             39 45   1.52     40 46   2.23     41 47   1.25
             42 48   1.79     43  1    .97     44  2   1.28
             45  3    .48     46  4    .68     48  6    .08

[BO,*,NY,*]   7 13    .03      9 15   1.29     11 17   4.59
             12 18   2.56     13 19   3.92     14 20   2.37
             15 21   3.81     16 22   2.24     17 23   3.51
             18 24   2.13     19 25   3.28     20 26   2.05
             21 27   3.15     22 28   1.99     23 29   3.09
             24 30   1.93     25 31   3.19     26 32   1.91
             27 33   3.21     28 34   1.85     29 35   3.21
             30 36   1.71     31 37   3.04     32 38   2.08
             33 39   3.13     34 40   1.96     35 41   2.53
             36 42   1.43     37 43   2.04     38 44   1.12
             39 45   1.71     40 46    .91     41 47   1.32
             43  1   1.80     45  3   1.13     47  5    .23

[NY,*,PH,*]   1  3    .04     12 14   4.68     13 15   5.61
             14 16   3.56     15 17   5.81     16 18   3.81
             17 19   6.31     18 20   4.07     19 21   7.33
             20 22   4.55     21 23   7.37     22 24   4.73
             23 25   7.61     24 26   4.92     25 27   7.91
             26 28   5.19     27 29   8.40     28 30   5.53
             29 31   9.32     30 32   5.51     31 33  10.33
             32 34   9.21     33 35  18.95     34 36  11.23
             35 37  16.85     36 38   7.29     37 39  10.89
             38 40   5.41     39 41   8.21     40 42   4.52
             41 43   6.99     42 44   3.92     43 45   6.21
             44 46   3.44     45 47   5.17     46 48   2.55
             47  1   1.24

[PH,*,WA,*]   1  4    .20     14 17   4.49     15 18   3.53
             16 19   2.67     17 20   3.83     18 21   3.01
             19 22   4.12     20 23   3.15     21 24   4.67
             22 25   3.20     23 26   4.23     24 27   2.87
             25 28   3.84     26 29   2.60     27 30   3.80
             28 31   2.77     29 32   4.31     30 33   3.16
             31 34   4.88     32 35   3.45     33 36   5.55
             34 37   3.52     35 38   6.11     36 39   3.32
             37 40   5.53     38 41   3.03     39 42   4.51
             40 43   2.53     41 44   3.39     42 45   1.93
             43 46   2.52     44 47   1.20     45 48   1.75
             46  1    .88     47  2    .87     ;

end;
```

```
                37 39    38 40    39 41    40 42
                41 43    42 44    43 45    44 46
                45 47    47  1
   (NY,*,BO,*)  10 16    12 18    14 20    15 21
                16 22    17 23    18 24    19 25
                20 26    21 27    22 28    23 29
                24 30    25 31    26 32    27 33
                28 34    29 35    30 36    31 37
                32 38    33 39    34 40    35 41
                36 42    37 43    38 44    39 45
                40 46    41 47    42 48    43  1
                44  2    45  3    46  4    48  6
   (BO,*,NY,*)   7 13     9 15    11 17    12 18
                13 19    14 20    15 21    16 22
                17 23    18 24    19 25    20 26
                21 27    22 28    23 29    24 30
                25 31    26 32    27 33    28 34
                29 35    30 36    31 37    32 38
                33 39    34 40    35 41    36 42
                37 43    38 44    39 45    40 46
                41 47    43  1    45  3    47  5
   (NY,*,PH,*)   1  3    12 14    13 15    14 16
                15 17    16 18    17 19    18 20
                19 21    20 22    21 23    22 24
                23 25    24 26    25 27    26 28
                27 29    28 30    29 31    30 32
                31 33    32 34    33 35    34 36
                35 37    36 38    37 39    38 40
                39 41    40 42    41 43    42 44
                43 45    44 46    45 47    46 48
                47  1
   (PH,*,WA,*)   1  4    14 17    15 18    16 19
                17 20    18 21    19 22    20 23
                21 24    22 25    23 26    24 27
                25 28    26 29    27 30    28 31
                29 32    30 33    31 34    32 35
                33 36    34 37    35 38    36 39
                37 40    38 41    39 42    40 43
                41 44    42 45    43 46    44 47
                45 48    46  1    47  2    ;

param demand :=
 [WA,*,PH,*]   2  5    .55      6  9    .01      8 11    .01
              10 13    .13     12 15   1.59     13 16   1.69
              14 17   5.19     15 18   3.55     16 19   6.29
              17 20   4.00     18 21   5.80     19 22   3.40
              20 23   4.88     21 24   2.92     22 25   4.37
              23 26   2.80     24 27   4.23     25 28   2.88
              26 29   4.33     27 30   3.11     28 31   4.64
              29 32   3.44     30 33   4.95     31 34   3.73
              32 35   5.27     33 36   3.77     34 37   4.80
              35 38   3.31     36 39   3.89     37 40   2.65
              38 41   3.01     39 42   2.04     40 43   2.31
              41 44   1.52     42 45   1.75     44 47   1.88
              46  1   1.05
 [PH,*,NY,*]   1  3   1.05      5  7    .43      9 11    .20
              11 13    .21     13 15    .40     14 16   6.49
              15 17  16.40     16 18   9.48     17 19  17.15
              18 20   9.31     19 21  15.20     20 22   8.21
              21 23  13.32     22 24   7.35     23 25  11.83
              24 26   6.61     25 27  10.61     26 28   6.05
              27 29   9.65     28 30   5.61     29 31   9.25
```

```
### OBJECTIVES ###

minimize cars:
       sum {c in cities} U[c,last] +
       sum {(c1,t1,c2,t2) in schedule: t2 < t1} X[c1,t1,c2,t2];
                        # Number of cars in the system:
                        # sum of unused cars and cars in trains during
                        # the last time interval of the day
minimize miles:
       sum {(c1,t1,c2,t2) in schedule} distance[c1,c2] * X[c1,t1,c2,t2];
                        # Total car-miles run by all scheduled trains in a day

### CONSTRAINTS ###

account {c in cities, t in times}:
  U[c,t] = U[c, if t > 1 then t-1 else last] +
      sum {(c1,t1,c,t) in schedule} X[c1,t1,c,t] -
      sum {(c,t,c2,t2) in schedule} X[c,t,c2,t2];
                        # For every city and time:
                        # unused cars in the present interval must equal
                        # unused cars in the previous interval,
                        # plus cars just arriving in trains,
                        # minus cars just leaving in trains

satisfy {(c1,t1,c2,t2) in schedule}:
       low[c1,t1,c2,t2] <= X[c1,t1,c2,t2] <= high[c1,t1,c2,t2];
                        # For each scheduled train:
                        # number of cars must meet demand,
                        # but must not be so great that unnecessary
                        # sections are run

### DATA ###

data;

set cities := BO NY PH WA ;

set links := (BO,NY) (NY,PH) (PH,WA)
             (NY,BO) (PH,NY) (WA,PH) ;

param dist_table := [*,*]  BO NY  232
                           NY PH   90
                           PH WA  135 ;

param last := 48 ;

param section := 14 ;

set schedule :=
   (WA,*,PH,*)   2  5     6  9     8 11    10 13
                12 15    13 16    14 17    15 18
                16 19    17 20    18 21    19 22
                20 23    21 24    22 25    23 26
                24 27    25 28    26 29    27 30
                28 31    29 32    30 33    31 34
                32 35    33 36    34 37    35 38
                36 39    37 40    38 41    39 42
                40 43    41 44    42 45    44 47
                46  1
   (PH,*,NY,*)   1  3     5  7     9 11    11 13
                13 15    14 16    15 17    16 18
                17 19    18 20    19 21    20 22
                21 23    22 24    23 25    24 26
                25 27    26 28    27 29    28 30
                29 31    30 32    31 33    32 34
                33 35    34 36    35 37    36 38
```

Appendix D. TRAIN, a model of railroad passenger car allocation

Given a day's schedule, this model allocates passenger cars to trains so as to minimize either the number of cars required or the number of car-miles run (Fourer, Gertler and Simkowitz 1977, 1978). The data represent a hypothetical schedule and demands for service between Washington, Philadelphia, New York and Boston.

```
###  SCHEDULE SETS AND PARAMETERS  ###

set cities;

set links within {c1 in cities, c2 in cities: c1 <> c2};

                        # Set of cities, and set of intercity links

param last > 0 integer; # Number of time intervals in a day

set times := 1..last;   # Set of time intervals in a day

set schedule within
      {c1 in cities, t1 in times,
       c2 in cities, t2 in times: (c1,c2) in links};

                        # Member (c1,t1,c2,t2) of this set represents
                        # a train that leaves city c1 at time t1
                        # and arrives in city c2 at time t2

###  DEMAND PARAMETERS  ###

param section > 0 integer;

                        # Maximum number of cars in one section of a train

param demand {schedule} > 0;

                        # For each scheduled train:
                        # the smallest number of cars that
                        # can meet demand for the train

param low {(c1,t1,c2,t2) in schedule} := ceil(demand[c1,t1,c2,t2]);

                        # Minimum number of cars needed to meet demand

param high {(c1,t1,c2,t2) in schedule}

   := max (2, min (ceil(2*demand[c1,t1,c2,t2]),
                   section*ceil(demand[c1,t1,c2,t2]/section) ));

                        # Maximum number of cars allowed on a train:
                        # 2 if demand is for less than one car;
                        # otherwise, lesser of
                        # number of cars needed to hold twice the demand, and
                        # number of cars in minimum number of sections needed

###  DISTANCE PARAMETERS  ###

param dist_table {links} >= 0 default 0.0;

param distance {(c1,c2) in links} > 0
   := if dist_table[c1,c2] > 0 then dist_table[c1,c2] else dist_table[c2,c1];

                        # Inter-city distances: distance[c1,c2] is miles
                        # between city c1 and city c2

###  VARIABLES  ###

var U 'cars stored' {cities,times} >= 0;

                        # u[c,t] is the number of unused cars stored
                        # at city c in the interval beginning at time t

var X 'cars in train' {schedule} >= 0;

                        # x[c1,t1,c2,t2] is the number of cars assigned to
                        # the scheduled train that leaves c1 at t1 and
                        # arrives in c2 at t2
```

```
param pro (tr) :
         18REG     24REG     24PRO   :=
  1          0         1         0
  2          0         0         0
  3          0         0         0
  4          1         0         0
  5          0         0         0
  6          0         0         0
  7          0         1         0
  8          0         0         0
  9          0         0         0
 10          1         0         1
 11          0         0         0
 12          0         0         0
 13          0         1         0
 14          0         1         0   ;

end;
```

```
ilim1 'new-inventory limits' {p in prd, t in time}:
    Inv[p,t,1] <= Rprd[p,t] + Oprd[p,t];
                        # New inventory cannot exceed
                        # production in the most recent period

ilim 'inventory limits' {p in prd, t in first+1..last, a in 2..life}:
    Inv[p,t,a] <= Inv[p,t-1,a-1];
                        # Inventory left from period (t+1)-p
                        # can only decrease as time goes on

###  DATA  ###

data;

set prd := 18REG 24REG 24PRO ;

param first :=  1 ;
param last  := 13 ;
param life  :=  2 ;

param cs := 18 ;
param sl :=  8 ;
param iw :=  8 ;

param rtr := 16.00 ;
param otr := 43.85 ;
param rir :=  0.75 ;
param pir :=  0.80 ;

param :         pt       pc        cri       crs      iinv   :=

  18REG      1.194    2304.      0.015     1.100      82.0
  24REG      1.509    2920.      0.015     1.100     792.2
  24PRO      1.509    2910.      0.015     1.100       0.0   ;

param :     dpp        ol      cmin      cmax        hc        lc   :=

  1        19.5      96.0       0.0       8.0      7500      7500
  2        19.0      96.0       0.0       8.0      7500      7500
  3        20.0      96.0       0.0       8.0      7500      7500
  4        19.0      96.0       0.0       8.0      7500      7500
  5        19.5      96.0       0.0       8.0     15000     15000
  6        19.0      96.0       0.0       8.0     15000     15000
  7        19.0      96.0       0.0       8.0     15000     15000
  8        20.0      96.0       0.0       8.0     15000     15000
  9        19.0      96.0       0.0       8.0     15000     15000
 10        20.0      96.0       0.0       8.0     15000     15000
 11        20.0      96.0       0.0       8.0      7500      7500
 12        18.0      96.0       0.0       8.0      7500      7500
 13        18.0      96.0       0.0       8.0      7500      7500   ;

param dem (tr) :
          18REG     24REG     24PRO    :=

  1        63.8    1212.0       0.0
  2        76.0     306.2       0.0
  3        88.4     319.0       0.0
  4       913.8     208.4       0.0
  5       115.0     298.0       0.0
  6       133.8     328.2       0.0
  7        79.6     959.6       0.0
  8       111.0     257.6       0.0
  9       121.6     335.6       0.0
 10       470.0     118.0    1102.0
 11        78.4     284.8       0.0
 12        99.4     970.0       0.0
 13       140.4     343.8       0.0
 14        63.8    1212.0       0.0   ;
```

```
###  OBJECTIVE  ###

minimize cost:
    sum {t in time} rtr * sl * dpp[t] * cs * Crews[t] +
    sum {t in time} hc[t] * Hire[t] +
    sum {t in time} lc[t] * Layoff[t] +
    sum {t in time, p in prd} otr * cs * pt[p] * Oprd[p,t] +
    sum {t in time, p in prd, a in 1..life} cri[p] * pc[p] * Inv[p,t,a] +
    sum {t in time, p in prd} crs[p] * pc[p] * Short[p,t];
                        # Full regular wages for all crews employed, plus
                        # penalties for hiring and layoffs, plus
                        # wages for any overtime worked, plus
                        # inventory and shortage costs
                        # (All other production costs are assumed
                        # to depend on initial inventory and on demands,
                        # and so are not included explicitly.)

###  CONSTRAINTS  ###

rlim 'regular-time limit' {t in time}:
    sum {p in prd} pt[p] * Rprd[p,t] <= sl * dpp[t] * Crews[t];
                        # Hours needed to accomplish all regular-time
                        # production in a period must not exceed
                        # hours available on all shifts

olim 'overtime limit' {t in time}:
    sum {p in prd} pt[p] * Oprd[p,t] <= ol[t];
                        # Hours needed to accomplish all overtime
                        # production in a period must not exceed
                        # the specified overtime limit

empl0 'initial crew level':  Crews[first-1] = iw;
                        # Use given initial workforce

empl 'crew levels' {t in time}:  Crews[t] = Crews[t-1] + Hire[t] - Layoff[t];
                        # Workforce changes by hiring or layoffs

emplbnd 'crew limits' {t in time}:  cmin[t] <= Crews[t] <= cmax[t];
                        # Workforce must remain within specified bounds

dreq1 'first demand requirement' {p in prd}:
    Rprd[p,first] + Oprd[p,first] + Short[p,first]
                             - Inv[p,first,1] = dem[p,first] less iinv[p];

dreq 'demand requirements' {p in prd, t in first+1..last}:
    Rprd[p,t] + Oprd[p,t] + Short[p,t] - Short[p,t-1]
                          + sum {a in 1..life} (Inv[p,t-1,a] - Inv[p,t,a])
                                                  = dem[p,t] less iil[p,t-1];
                        # Production plus increase in shortage plus
                        # decrease in inventory must equal demand

ireq 'inventory requirements' {p in prd, t in time}:
    sum {a in 1..life} Inv[p,t,a] + iil[p,t] >= minv[p,t];
                        # Inventory in storage at end of period t
                        # must meet specified minimum

izero 'impossible inventories' {p in prd, v in 1..life-1, a in v+1..life}:
    Inv[p,first+v-1,a] = 0;
                        # In the vth period (starting from first)
                        # no inventory may be more than v periods old
                        # (initial inventories are handled separately)
```

```
###  DEMAND PARAMETERS  ###

param dem 'demand' {prd,first..last+1} >= 0;
                        # Requirements (in 1000s)
                        # to be met from current production and inventory

param pro 'promoted' {prd,first..last+1} logical;
                        # true if product will be the subject
                        # of a special promotion in the period

###  INVENTORY AND SHORTAGE PARAMETERS  ###

param rir 'regular inventory ratio' >= 0;
                        # Proportion of non-promoted demand
                        # that must be in inventory the previous period

param pir 'promotional inventory ratio' >= 0;
                        # Proportion of promoted demand
                        # that must be in inventory the previous period

param life 'inventory lifetime' > 0 integer;
                        # Upper limit on number of periods that
                        # any product may sit in inventory

param cri 'inventory cost ratio' {prd} > 0;
                        # Inventory cost per 1000 units is
                        # cri times nominal production cost

param crs 'shortage cost ratio' {prd} > 0;
                        # Shortage cost per 1000 units is
                        # crs times nominal production cost

param iinv 'initial inventory' {prd} >= 0;
                        # Inventory at start of first period; age unknown

param iil 'initial inventory left' {p in prd, t in time}
              := iinv[p] less sum {v in first..t} dem[p,v];
                        # Initial inventory still available for allocation
                        # at end of period t

param minv 'minimum inventory' {p in prd, t in time}
              := dem[p,t+1] * (if pro[p,t+1] then pir else rir);
                        # Lower limit on inventory at end of period t

###  VARIABLES  ###

var Crews{first-1..last} >= 0;
                        # Average number of crews employed in each period

var Hire{time} >= 0;    # Crews hired from previous to current period

var Layoff{time} >= 0;  # Crews laid off from previous to current period

var Rprd 'regular production' {prd,time} >= 0;
                        # Production using regular-time labor, in 1000s

var Oprd 'overtime production' {prd,time} >= 0;
                        # Production using overtime labor, in 1000s

var Inv 'inventory' {prd,time,1..life} >= 0;
                        # Inv[p,t,a] is the amount of product p that is
                        # a periods old -- produced in period (t+1)-a --
                        # and still in storage at the end of period t

var Short 'shortage' {prd,time} >= 0;
                        # Accumulated unsatisfied demand at the end of period t
```

Appendix C. PROD, a multiperiod production model

This model determines a series of workforce levels that will most economically meet demands and inventory requirements over time. The formulation is motivated by the experiences of a large producer in the United States. The data are for three products and 13 periods.

```
###  PRODUCTION SETS AND PARAMETERS  ###

set prd 'products';    # Members of the product group

param pt 'production time' {prd} > 0;
                        # Crew-hours to produce 1000 units

param pc 'production cost' {prd} > 0;
                        # Nominal production cost per 1000, used
                        # to compute inventory and shortage costs

###  TIME PERIOD SETS AND PARAMETERS  ###

param first > 0 integer;
                        # Index of first production period to be modeled

param last > first integer;
                        # Index of last production period to be modeled

set time 'planning horizon' := first..last;

###  EMPLOYMENT PARAMETERS  ###

param cs 'crew size' > 0 integer;
                        # Workers per crew

param sl 'shift length' > 0;
                        # Regular-time hours per shift

param rtr 'regular time rate' > 0;
                        # Wage per hour for regular-time labor

param otr 'overtime rate' > rtr;
                        # Wage per hour for overtime labor

param iw 'initial workforce' >= 0 integer;
                        # Crews employed at start of first period

param dpp 'days per period' {time} > 0;
                        # Regular working days in a production period

param ol 'overtime limit' {time} >= 0;
                        # Maximum crew-hours of overtime in a period

param cmin 'crew minimum' {time} >= 0;
                        # Lower limit on average employment in a period

param cmax 'crew maximum' {t in time} >= cmin[t];
                        # Upper limit on average employment in a period

param hc 'hiring cost' {time} >= 0;
                        # Penalty cost of hiring a crew

param lc 'layoff cost' {time} >= 0;
                        # Penalty cost of laying off a crew
```

```
param dcap  default 0.0  :=
   [ABU_ZAABAL,*]   SSP          600
                    SULF_A_P     227
                    SULF_A_S     242
   [ASSIOUT,*]      SSP          600
                    SULF_A_S     250
   [ASWAN,*]        AMM_ELEC     450
                    C_AMM_NITR  1100
                    NITR_ACID    800
   [HELWAN,*]       AMM_C_GAS    172
                    AMM_SULF      24
                    C_AMM_NITR   364
                    NITR_ACID    282
   [KAFR_EL_ZT,*]   SSP          600
                    SULF_A_P      50
                    SULF_A_S     200  ;

end;
```

```
   [*,CAN_335]    AMMONIA        -.21
                  BAGS         -23.
                  CAN_335        1.0
                  LIMESTONE      -.04
                  NITR_ACID      -.76
                  STEAM          -.4
                  WATER        -49.
   [*,NITR_ACID]  AMMONIA        -.292
                  ELECTRIC    -231.
                  NITR_ACID      1.0
                  WATER          -.6
   [*,SSP_155]    BAGS         -22.
                  ELECTRIC     -14.
                  PHOS_ROCK      -.62
                  SSP_155        1.0
                  SULF_ACID      -.41
                  WATER         -6.
   [*,SULF_A_P]   ELECTRIC     -75.
                  PYRITES        -.826
                  SULF_ACID      1.0
                  WATER        -60.
   [*,SULF_A_S]   ELECTRIC     -50.
                  EL_SULFUR      -.334
                  SULF_ACID      1.0
                  WATER        -20. ;

param util  default 0  :=

   [*,*]   SULF_A_S SULF_A_S    1      SULF_A_P SULF_A_P   1
           NITR_ACID NITR_ACID  1      AMM_ELEC AMM_ELEC   1
           AMM_C_GAS AMM_C_GAS  1      SSP SSP_155         1
           C_AMM_NITR CAN_310   1      C_AMM_NITR CAN_335  1
           AMM_SULF AMM_SULF    1 ;

param p_imp  default 0.0  :=

     PYRITES       17.5           AMM_SULF      75.
     EL_SULFUR     55.            DAP          175.
     UREA         150.            SSP_155       80.
     CAN_260       75.            C_250_55     100.
     CAN_310       90.            C_300_100    130.
     CAN_335      100.   ;

param p_r  default 0.0  :=

     ELECTRIC     .007
     BF_GAS       .007
     WATER        .031
     STEAM       1.25
     BAGS         .28   ;

param p_pr  default 0.0  :=

 [HELWAN,COKE_GAS]              16.0
 [ASWAN,EL_ASWAN]                1.0

 [*,LIMESTONE]      ASWAN        1.2
                    HELWAN       1.2

 [*,PHOS_ROCK]      ABU_ZAABAL   4.0
                    ASSIOUT      3.5
                    KAFR_EL_ZT   5.0   ;
```

```
param road  default 0.0  :
            ABU_KIR ABU_ZAABAL ASSIOUT ASWAN HELWAN KAFR_EL_ZT SUEZ TALKHA :=
ALEXANDRIA      16     210       607    1135   244      119     362   187
ASSIOUT        616     420         .     518   362      504     527   518
ASWAN         1134     938       518       .   880     1022    1045  1036
BEHERA          76      50       547    1065   184       42     288   120
BENI_SUEF      359     163       257     775   105      248     270   261
DAKAHLIA       208     138       515    1033   152       58     219     3
DAMIETTA       267     216       596    1114   233      131     286    66
FAYOUM         341     145       308     826    88      230     252   243
GHARBIA        150      65       485    1003   122       20     226    55
GIZA           287      48       372     890    .9      133     169   146
ISMAILIA       365     142       536    1054   173      241      89   146
KAFR_EL_SH     145     105       525    1043   162       20     266    35
KALUBIA        190      97       439     957    76       66     180    81
MENOUFIA       157     154       472     990   109       33     213    90
MINIA          384     288       132     650   230      372     394   386
NEW_VALLEY     815     619       199     519   561      703     726   717
QUENA          858     662       242     276   604      746     769   760
SHARKIA        240      60       473     991   110       78     214    58
SOHAG          715     519        99     419   461      603     626   617
SUEZ           370     224       541    1059   178      246       .   298  ;

param rail_half  default 0  :
              KAFR_EL_ZT   ABU_ZAABAL    HELWAN     ASSIOUT   :=
ABU_ZAABAL        85            .            .          .
HELWAN           142           57            .          .
ASSIOUT          504          420          362          .
ASWAN           1022          938          880        518    ;

param :            impd_barg   impd_road :=
ABU_ZAABAL           210          .1
ASSIOUT              583         0
ASWAN               1087        10
HELWAN               183         0
KAFR_EL_ZT           104         6 ;

param io  default 0.0  :=
   [*,AMM_C_GAS]  AMMONIA        1.0
                  BF_GAS      -609.
                  COKE_GAS      -2.0
                  ELECTRIC   -1960.
                  STEAM         -4.
                  WATER       -700.

   [*,AMM_ELEC]   AMMONIA        1.0
                  EL_ASWAN     -12.0

   [*,AMM_SULF]   AMMONIA        -.26
                  AMM_SULF       1.0
                  BAGS         -22.
                  ELECTRIC     -19.
                  SULF_ACID      -.76
                  WATER        -17.

   [*,CAN_310]    AMMONIA        -.20
                  BAGS         -23.
                  CAN_310        1.0
                  LIMESTONE      -.12
                  NITR_ACID      -.71
                  STEAM          -.4
                  WATER        -49.
```

```
param cf75  default 0.0  :
               CAN_260    CAN_310    CAN_335   AMM_SULF     UREA   :=
ALEXANDRIA        .           .         5.0        3.0       1.0
ASSIOUT        1.0        20.0       26.0        1.0      27.0
ASWAN             .       40.0          .          .         .
BEHERA         1.0          .        25.0       90.0      35.0
BENI_SUEF      1.0          .        15.0        1.0      20.0
DAKAHLIA       1.0          .        26.0       60.0      20.0
DAMIETTA          .           .         2.0       15.0       8.0
FAYOUM         1.0          .        20.0        6.0      20.0
GHARBIA           .           .        17.0       60.0      28.0
GIZA              .           .        40.0        6.0       2.0
ISMAILIA          .           .         4.0        6.0       2.0
KAFR_EL_SH     1.0          .        10.0       45.0      22.0
KALUBIA           .           .        25.0       16.0       7.0
MENOUFIA       1.0          .        24.0       21.0      30.0
MINIA          2.0        15.0       35.0        1.0      41.0
NEW_VALLEY        .           .          .          .        1.0
QUENA             .       95.0        2.0          .        3.0
SHARKIA        1.0          .        31.0       50.0      28.0
SOHAG             .       65.0        3.0          .        7.0
SUEZ              .           .         1.0          .         .

   :          SSP_155    C_250_55   C_300_100    DAP   :=
ALEXANDRIA       8.0          .          .         .
ASSIOUT         35.0        5.0         .1         .
ASWAN            8.0          .          .         .
BEHERA          64.0        1.0         .1        .1
BENI_SUEF       13.0        3.0          .         .
DAKAHLIA        52.0        1.0          .         .
DAMIETTA         5.0          .          .         .
FAYOUM          17.0        1.0          .         .
GHARBIA         57.0        1.0         .2        .1
GIZA            14.0        1.0         .1         .
ISMAILIA         4.0          .          .         .
KAFR_EL_SH      25.0        2.0         .1         .
KALUBIA         22.0        1.0          .        .1
MENOUFIA        33.0        2.0         .1        .1
MINIA           50.0        3.0         .2        .1
NEW_VALLEY       1.0          .          .         .
QUENA            8.0          .          .         .
SHARKIA         43.0        1.0         .1         .
SOHAG           20.0        1.0          .         .
SUEZ             1.0          .          .         .       ;

param fn  default 0.0 :      N     P205    :=
            AMM_SULF       .206     .
            CAN_260        .26      .
            CAN_310        .31      .
            CAN_335        .335     .
            C_250_55       .25      .055
            C_300_100      .30      .10
            DAP            .18      .46
            SSP_155        .        .15
            UREA           .46      .      ;
```

```
subject to al:
    Psil  =  sum {c in c_final} (
               sum {pl in cp_pos[c], r in region}
                                              tran_final[pl,r] * Xf[c,pl,r]
             + sum {po in port, r in region} tran_import[r,po] * Vf[c,r,po] )
           + sum {c in c_ship, p1 in cp_pos[c], p2 in cc_pos[c]}
                                               tran_inter[p1,p2] * Xi[c,p1,p2]
           + sum {c in c_raw, pl in cc_pos[c]: p_imp[c] > 0}
                                                    tran_raw[pl] * Vr[c,pl];
                           # Total transport cost is sum of shipping costs for
                           #   (1) all final products from all plants,
                           #   (2) all imports of final products,
                           #   (3) all intermediates shipped between plants,
                           #   (4) all imports of raw materials
subject to ai:
    Psii / exch  =  sum {c in c_final, r in region, po in port}
                                                      p_imp[c] * Vf[c,r,po]
                  + sum {c in c_raw, pl in cc_pos[c]} p_imp[c] * Vr[c,pl];
                           # Total import cost -- at exchange rate --
                           # is sum of import costs for final products
                           # in each region and raw materials at each plant

###  DATA  ###

data;

set center := ASWAN HELWAN ASSIOUT KAFR_EL_ZT ABU_ZAABAL ABU_KIR TALKHA SUEZ ;

set port := ABU_KIR ;

set plant := ASWAN HELWAN ASSIOUT KAFR_EL_ZT ABU_ZAABAL ;

set region := ALEXANDRIA BEHERA GHARBIA KAFR_EL_SH DAKAHLIA DAMIETTA
              SHARKIA ISMAILIA SUEZ MENOUFIA KALUBIA GIZA BENI_SUEF FAYOUM
              MINIA ASSIOUT NEW_VALLEY SOHAG QUENA ASWAN ;

set unit := SULF_A_S SULF_A_P NITR_ACID AMM_ELEC AMM_C_GAS C_AMM_NITR
            AMM_SULF SSP ;

set proc := SULF_A_S SULF_A_P NITR_ACID AMM_ELEC AMM_C_GAS CAN_310 CAN_335
            AMM_SULF SSP_155 ;

set nutr := N P205 ;

set c_final := UREA CAN_260 CAN_310 CAN_335 AMM_SULF DAP SSP_155 C_250_55
               C_300_100 ;

set c_inter := AMMONIA NITR_ACID SULF_ACID ;

set c_ship := AMMONIA SULF_ACID ;

set c_raw := EL_ASWAN COKE_GAS PHOS_ROCK LIMESTONE EL_SULFUR PYRITES
             ELECTRIC BF_GAS WATER STEAM BAGS ;

set p_except[ASWAN] := CAN_335 ;
set p_except[HELWAN] := CAN_310 ;
set p_except[ASSIOUT] := ;
set p_except[KAFR_EL_ZT] := ;
set p_except[ABU_ZAABAL] := ;

param exch := 0.4;

param util_pct := 0.85;
```

```
var U {c in c_raw, cc_pos[c]} >= 0;
                           # U[c,pl] is amount of raw material c
                           # purchased domestically for use at plant pl

var Psip;                  # Domestic recurrent cost
var Psil;                  # Transport cost
var Psii;                  # Import cost

###  OBJECTIVE  ###

minimize Psi:  Psip + Psil + Psii;

###  CONSTRAINTS  ###

subject to mbd {n in nutr, r in region}:
    sum {c in c_final} fn[c,n] *
                (sum {po in port} Vf[c,r,po] +
                 sum {pl in cp_pos[c]} Xf[c,pl,r])  >=  cn75[r,n];
                           # Total nutrients supplied to a region by all
                           # final products (sum of imports plus internal
                           # shipments from plants) must meet requirements

subject to mbdb {c in c_final, r in region: cf75[r,c] > 0}:
    sum {po in port} Vf[c,r,po] +
    sum {pl in cp_pos[c]} Xf[c,pl,r]  >=  cf75[r,c];
                           # Total of each final product supplied to each
                           # region (as in previous constraint) must meet
                           # requirements

subject to mb {c in commod, pl in plant}:
    sum {pr in p_pos[pl]} io[c,pr] * Z[pl,pr]
   + ( if c in c_ship then
                ( if pl in cp_pos[c] then sum {p2 in cc_pos[c]} Xi[c,pl,p2] )
              + ( if pl in cc_pos[c] then sum {p2 in cp_pos[c]} Xi[c,p2,pl] ))
   + ( if (c in c_raw and pl in cc_pos[c]) then
                 (( if p_imp[c] > 0 then Vr[c,pl] )
                + ( if p_dom[pl,c] > 0 then U[c,pl] )))
  >= if (c in c_final and pl in cp_pos[c]) then sum {r in region} Xf[c,pl,r];
                           # For each commodity at each plant:  sum of
                           #   (1) production or consumption at plant,
                           #   (2) inter-plant shipments in or out,
                           #   (3) import and domestic purchases (raw only)
                           # is >= 0 for raw materials and intermediates;
                           # is >= the total shipped for final products

subject to cc {pl in plant, u in m_pos[pl]}:
    sum {pr in p_pos[pl]} util[u,pr] * Z[pl,pr]  <=  util_pct * icap[u,pl];
                           # For each productive unit at each plant,
                           # total utilization by all processes
                           # may not exceed the unit's capacity
subject to ap:
    Psip  =  sum {c in c_raw, pl in cc_pos[c]} p_dom[pl,c] * U[c,pl];
                           # Psip is the cost of domestic raw materials,
                           # summed over all plants that consume them
```

```
param p_imp {commod} >= 0; # Import Price (US$ per ton 1975)

param p_r {c_raw} >= 0;
param p_pr {plant,c_raw} >= 0;

param p_dom {pl in plant, c in c_raw} :=
              if p_r[c] > 0 then p_r[c] else p_pr[pl,c];
                           # Domestic raw material prices

param dcap {plant,unit} >= 0;
                           # Design capacity of plants (t/day)

param icap {u in unit, pl in plant} := 0.33 * dcap[pl,u];
                           # Initial capacity of plants (t/day)

param exch;                # Exchange rate

param util_pct;            # Utilization percent for initial capacity

###  DERIVED SETS OF "POSSIBILITIES"  ###

set m_pos {pl in plant} := {u in unit: icap[u,pl] > 0};
                           # At each plant, set of units for which there is
                           # initial capacity

set p_cap {pl in plant} :=
             {pr in proc: forall {u in unit: util[u,pr] > 0} u in m_pos[pl] };
                           # At each plant, set of processes for which
                           # all necessary units have some initial capacity

set p_except {plant} within proc;
                           # At each plant, list of processes that are
                           # arbitrarily ruled out

set p_pos {pl in plant} := p_cap[pl] diff p_except[pl];
                           # At each plant, set of possible processes

set cp_pos {c in commod} := {pl in plant: sum {pr in p_pos[pl]} io[c,pr] > 0};

set cc_pos {c in commod} := {pl in plant: sum {pr in p_pos[pl]} io[c,pr] < 0};

set c_pos {c in commod} := cp_pos[c] union cc_pos[c];
                           # For each commodity, set of plants that can
                           # produce it (cp_pos) or consume it (cc_pos),
                           # and their union (c_pos)

###  VARIABLES  ###

var Z {pl in plant, p_pos[pl]} >= 0;
                           # Z[pl,pr] is level of process pr at plant pl

var Xf {c in c_final, cp_pos[c], region} >= 0;
                           # Xf[c,pl,r] is amount of final product c
                           # shipped from plant pl to region r

var Xi {c in c_ship, cp_pos[c], cc_pos[c]} >= 0;
                           # Xi[c,p1,p2] is amount of intermediate c
                           # shipped from plant p1 to plant p2

var Vf {c_final,region,port} >= 0;
                           # Vf[c,r,po] is amount of final product c
                           # imported by region r from port po

var Vr {c in c_raw, cc_pos[c]} >= 0;
                           # Vr[c,pl] is amount of raw material c
                           # imported for use at plant pl
```

Appendix B. EGYPT, a static model of fertilizer production

This static production model, originally stated in the GAMS language (Bisschop and Meeraus 1982), is based on a World Bank study of the Egyptian fertilizer industry (Choksi, Meeraus and Stoutjesdijk 1980).

```
###  SETS  ###

set center;                # Locations from which final product may be shipped
set port within center;    # Locations at which imports can be received
set plant within center;   # Locations of plants

set region;                # Demand regions

set unit;                  # Productive units
set proc;                  # Processes

set nutr;                  # Nutrients

set c_final;               # Final products (fertilizers)
set c_inter;               # Intermediate products
set c_ship within c_inter; # Intermediates for shipment
set c_raw;                 # Domestic raw materials and miscellaneous inputs

set commod := c_final union c_inter union c_raw;
                           # All commodities

###  PARAMETERS  ###

param cf75 {region,c_final} >= 0;
                           # Consumption of fertilizer 1974-75 (1000 tons/year)

param fn {c_final,nutr} >= 0;
                           # Nutrient content of fertilizers

param cn75 {r in region, n in nutr} := sum {c in c_final} cf75[r,c] * fn[c,n];
                           # Consumption of nutrients 1974-75 (1000 tons/year)

param road {region,center} >= 0;
                           # Road distances

param rail_half {plant,plant} >= 0;
param rail {p1 in plant, p2 in plant} :=
    if rail_half[p1,p2] > 0 then rail_half[p1,p2] else rail_half[p2,p1];
                           # Interplant rail distances (kms)

param impd_barg {plant} >= 0;
param impd_road {plant} >= 0;
                           # Import distances (kms) by barge and road

param tran_final {pl in plant, r in region} :=
              if road[r,pl] > 0 then .5 + .0144 * road[r,pl] else 0;

param tran_import {r in region, po in port} :=
              if road[r,po] > 0 then .5 + .0144 * road[r,po] else 0;

param tran_inter {p1 in plant, p2 in plant} :=
              if rail[p1,p2] > 0 then 3.5 + .03 * rail[p1,p2] else 0;

param tran_raw {pl in plant} :=
            (if impd_barg[pl] > 0 then 1.0 + .0030 * impd_barg[pl] else 0)
          + (if impd_road[pl] > 0 then 0.5 + .0144 * impd_road[pl] else 0);
                           # Transport cost (le per ton) for:
                           #   final products, imported final products,
                           #   interplant shipment, imported raw materials

param io {commod,proc};    # Input-output coefficients

param util {unit,proc} >= 0;
                           # Capacity utilization coefficients
```

```
w38      0.015     0.009     0.012
w39      0.007     0.017     0.022
w40      0.009     0.014     0.020
w41      0.003     0.014     0.011
w42      0.017     0.011     0.012
w43      0.009     0.013     0.011
w44      0.002     0.012     0.012
w45      0.016     0.025     0.028
w46      0.038     0.062     0.040
w47      0.007     0.010     0.010
w48      0.003     0.015     0.016
w49      0.005     0.016     0.017
w50      0.011     0.008     0.007
w51      0.010     0.022     0.021
w53      0.004     0.026     0.020
w54      0.020     0.017     0.025
w55      0.004     0.019     0.028
w56      0.004     0.010     0.008
w57      0.014     0.020     0.018
w59      0.012     0.006     0.007
w60      0.019     0.010     0.009
w61      0.028     0.010     0.012
w62      0.000     0.000     0.000
w63      0.070     0.027     0.037
w64      0.009     0.004     0.005
w65      0.022     0.015     0.016
w66      0.046     0.017     0.020
w68      0.005     0.012     0.016
w69      0.085     0.036     0.039
w71      0.011     0.013     0.010
w72      0.089     0.031     0.034
w75      0.026     0.012     0.010
w77      0.001     0.004     0.002
w78      0.002     0.004     0.002
w79      0.001     0.004     0.002
w80      0.001     0.001     0.002
w81      0.001     0.003     0.002
w83      0.009     0.010     0.008
w84      0.001     0.002     0.002
w85      0.001     0.004     0.005
w86      0.001     0.002     0.002
w87      0.002     0.003     0.000
w89      0.001     0.001     0.002
w90      0.006     0.017     0.013
w91      0.002     0.010     0.013
w92      0.000     0.003     0.002
w93      0.002     0.006     0.007
w95      0.001     0.007     0.007
w96      0.000     0.000     0.000
w98      0.006     0.005     0.002    ;

end;
```

```
w66        0       1       1       1       0       1       1       1
w68        0       1       1       1       0       1       1       1
w69        0       1       1       1       0       1       1       1
w71        0       1       1       1       0       1       1       1
w72        0       1       1       1       0       1       1       1
w73        0       1       1       1       0       1       1       0
w74        0       1       1       1       0       0       0       1
w75        0       1       1       1       0       1       1       1
w76        0       0       0       0       0       0       0       0
w77        1       0       1       1       1       0       0       1
w78        1       0       1       1       1       0       0       1
w79        1       0       1       1       1       0       0       1
w80        1       0       1       1       1       0       0       1
w81        1       0       1       1       1       0       0       1
w82        1       0       1       1       1       1       1       0
w83        1       0       1       1       1       0       1       1
w84        1       0       1       1       1       0       0       1
w85        1       1       1       1       1       0       0       1
w86        1       0       1       1       1       0       0       1
w87        1       0       1       1       1       1       1       0
w89        1       0       1       1       1       1       0       1
w90        0       1       1       1       1       0       0       1
w91        1       0       1       1       1       0       0       1
w92        1       0       1       1       1       0       0       1
w93        1       1       1       0       1       0       0       1
w94        0       0       1       1       1       0       1       1
w95        1       0       1       1       1       0       0       1
w96        0       0       0       0       0       0       0       0
w98        1       0       1       1       1       1       0       1
x22        1       1       1       1       0       0       1       0
x23        1       1       1       1       0       0       1       0    ;

param ds default 0.000 (tr) :
         18REG     24REG     24PRO    :=
w01      0.000     0.000     0.008
w02      0.004     0.000     0.000
w03      0.000     0.000     0.000
w04      0.010     0.002     0.000
w05      0.000     0.000     0.000
w06      0.010     0.008     0.008
w08      0.030     0.024     0.024
w09      0.014     0.018     0.020
w12      0.014     0.012     0.010
w14      0.007     0.007     0.012
w15      0.010     0.019     0.018
w17      0.013     0.010     0.011
w19      0.015     0.012     0.009
w20      0.012     0.021     0.022
w21      0.000     0.000     0.000
w24      0.012     0.022     0.018
w25      0.019     0.025     0.020
w26      0.006     0.015     0.021
w27      0.008     0.010     0.015
w28      0.011     0.016     0.019
w29      0.008     0.020     0.013
w30      0.011     0.013     0.015
w31      0.011     0.013     0.017
w32      0.006     0.000     0.000
w33      0.000     0.015     0.014
w34      0.008     0.007     0.005
w35      0.002     0.006     0.014
w36      0.015     0.013     0.005
w37      0.017     0.016     0.015
```

```
w95     5.82     3.29     6.55     7.06    11.47      .        .      7.83
w96     1.77     5.20     2.72     0.59     3.47     2.48      .        .
w98     3.04     1.92     3.64     3.70     4.90     3.05      .      3.88
x22     4.08     6.25     4.15     4.30     1.77      .      1.77      .
x23     3.39     5.74     3.55     4.08     1.69      .      1.47      .      ;

param msr (tr) :
         w01      w02      w03      w04      w05      w62      w76      w96     :=
w01        0        0        0        0        0        0        1        0
w02        0        0        0        0        0        0        1        0
w03        0        0        0        0        0        0        1        0
w04        0        0        0        0        0        0        1        0
w05        0        0        0        0        0        0        0        0
w06        0        1        1        1        1        1        1        1
w08        0        1        1        1        1        1        1        1
w09        0        1        1        1        1        1        0        1
w12        0        1        1        1        1        0        1        1
w14        1        1        1        1        1        0        0        1
w15        0        1        1        1        1        1        0        1
w17        0        1        1        1        1        1        0        1
w18        0        1        1        1        1        0        0        1
w19        0        1        1        1        1        0        0        1
w20        1        1        1        1        1        0        0        1
w24        0        1        1        1        1        0        0        1
w25        0        1        1        1        1        0        1        0
w26        1        1        1        0        1        1        0        1
w27        1        1        1        0        1        1        0        1
w28        1        1        1        0        1        0        0        1
w29        0        1        1        1        1        0        0        1
w30        1        1        1        0        1        1        0        1
w31        1        1        1        0        1        0        0        1
w32        0        0        0        0        0        0        0        0
w33        1        0        1        1        1        1        0        1
w34        1        1        1        0        1        0        0        1
w35        1        1        1        1        1        0        0        1
w36        0        1        1        1        0        1        1        1
w37        1        1        1        0        1        1        0        1
w38        1        1        1        0        1        0        0        1
w39        0        1        1        1        1        1        0        1
w40        1        1        1        0        1        0        0        1
w41        1        0        1        1        1        0        0        1
w42        1        1        1        0        1        0        0        1
w43        1        1        1        0        1        0        0        1
w44        1        1        1        1        1        0        0        1
w45        0        1        1        1        1        1        0        1
w46        0        1        1        1        1        0        1        1
w47        0        1        1        1        1        1        0        1
w48        0        1        1        1        1        0        0        1
w49        1        1        1        1        1        0        0        1
w50        0        1        1        1        1        1        0        1
w51        0        1        1        1        1        0        1        1
w53        1        1        1        1        1        0        0        1
w54        0        1        1        1        1        1        1        1
w55        0        1        1        1        1        0        0        1
w56        0        1        1        1        1        1        0        1
w57        0        1        1        1        1        1        0        1
w59        0        1        1        1        0        1        1        1
w60        0        1        1        1        1        0        1        1
w61        0        1        1        1        0        0        1        1
w62        0        0        0        0        0        0        1        0
w63        0        1        1        1        0        1        1        1
w64        0        1        1        1        1        1        0        1
w65        0        1        1        1        0        1        1        1
```

w27	2.48	6.87	3.17	1.59	2.08	3.45	.	0.99
w28	2.05	6.83	2.97	1.13	2.91	.	.	1.26
w29	4.03	3.68	4.46	3.20	5.50	.	.	3.20
w30	2.48	5.78	2.99	2.24	1.79	3.10	.	1.39
w31	2.34	5.41	2.87	1.67	1.66	.	.	1.39
w32	14.36	.	.	.	.	.	.	.
w33	3.87	4.27	5.11	3.48	5.66	4.03	.	3.05
w34	3.26	4.80	3.21	2.70	4.14	.	.	1.77
w35	2.34	2.84	2.89	3.35	3.78	2.68	.	2.52
w36	2.43	5.69	2.96	2.95	1.02	2.61	1.07	2.54
w37	2.23	4.64	2.41	1.99	4.30	2.61	.	1.44
w38	4.66	4.36	5.23	3.04	4.46	.	.	3.82
w39	1.11	3.51	1.10	2.53	3.07	1.12	.	2.23
w40	2.99	4.78	4.23	1.57	3.92	.	.	1.80
w41	4.93	4.00	5.43	4.45	6.31	.	.	3.81
w42	3.86	6.55	5.03	2.11	4.41	.	.	2.63
w43	4.61	4.45	3.77	1.22	4.31	.	.	2.35
w44	2.05	4.48	1.06	3.70	3.46	1.10	.	3.21
w45	0.92	3.42	1.58	3.04	1.82	1.94	.	2.52
w46	1.36	2.44	0.95	3.08	2.78	0.39	2.16	2.37
w47	1.30	3.39	1.60	2.49	4.29	2.04	.	1.68
w48	1.65	3.78	1.03	2.97	2.21	1.31	.	2.74
w49	1.96	3.00	1.50	3.24	3.68	1.00	.	2.99
w50	0.90	4.14	1.60	1.95	3.61	1.61	.	1.52
w51	1.59	3.95	0.25	2.96	2.58	1.00	2.41	2.71
w53	1.59	3.79	1.28	3.12	3.10	0.89	.	2.98
w54	1.72	4.36	1.61	2.92	2.34	1.91	1.97	3.05
w55	2.45	2.73	2.21	4.47	4.30	2.57	.	4.48
w56	1.10	3.73	1.59	2.74	2.33	1.45	.	2.44
w57	0.95	3.39	1.37	2.30	2.47	1.15	.	1.95
w59	3.29	5.35	3.32	3.81	1.52	3.38	1.34	4.08
w60	2.41	6.12	2.46	3.65	2.35	.	1.37	4.06
w61	3.32	5.50	3.41	3.38	1.23	.	0.99	4.28
w62	1.12	3.00	0.82	3.22	2.95	.	3.33	2.53
w63	3.59	6.36	3.25	4.12	1.84	3.59	1.46	4.03
w64	1.85	4.45	2.17	3.43	2.13	2.03	.	4.02
w65	2.78	4.79	2.81	2.94	1.54	2.90	1.07	2.94
w66	3.90	5.79	3.05	3.65	1.36	3.39	1.22	3.57
w68	2.61	5.20	2.90	2.34	1.68	3.19	1.48	2.31
w69	2.94	5.21	2.78	3.43	0.21	3.26	0.68	2.54
w71	2.06	4.98	2.38	2.44	1.59	2.97	1.05	2.55
w72	2.61	5.50	2.83	3.12	1.35	3.23	0.88	2.99
w73	8.52	6.16	8.03	8.83	10.44	7.38	10.26	.
w74	6.11	5.46	9.07	9.38	10.80	.	.	8.25
w75	2.66	4.94	2.87	3.69	1.52	3.15	1.24	4.00
w76	1.99	5.26	2.23	3.36	0.58	3.17	.	2.50
w77	4.32	3.07	5.05	3.88	6.04	.	.	4.15
w78	5.60	2.59	5.78	5.56	7.10	.	.	5.60
w79	4.25	2.32	4.93	4.57	6.04	.	.	4.58
w80	5.94	4.00	5.60	7.02	9.46	.	.	7.51
w81	5.39	2.21	5.10	6.22	6.46	.	.	6.58
w82	8.80	5.69	9.29	9.88	11.69	8.63	11.52	.
w83	4.40	.	5.24	5.21	5.81	3.91	7.04	5.33
w84	5.87	5.43	6.17	5.70	7.63	.	.	5.70
w85	3.90	3.65	3.38	4.57	5.64	3.05	.	5.04
w86	5.48	2.10	5.70	6.37	7.33	.	.	6.19
w87	8.88	5.54	9.50	9.71	11.64	8.85	11.68	.
w89	4.62	4.01	4.03	6.30	6.30	3.81	.	7.77
w90	4.35	2.72	4.61	4.01	5.60	.	.	3.20
w91	7.61	4.42	7.83	6.85	8.79	.	.	7.66
w92	7.15	2.69	6.91	7.20	.	.	.	7.06
w93	3.17	3.95	4.37	3.74	5.05	.	.	2.40
w94	1.21	3.07	0.90	2.74	3.17	.	2.63	2.39

```
set fact := w01 w05 w96 ;

param huge := 99. ;

param rtmin := 0.0 ;
param rtmax := 8.0 ;

param otmin :=  0.0 ;
param otmax := 96.0 ;

param rmin  :=  w01 0.00   w05 0.00   w96 0.00 ;
param rmax  :=  w01 3.00   w05 2.00   w96 3.00 ;

param omin  :=  w01  0.0   w05  0.0   w96  0.0 ;
param omax  :=  w01 48.0   w05  0.0   w96 48.0 ;

param hd    :=  w01  8.0   w05  8.0   w96  8.0 ;

param dp    :=  w01 19.0   w05 19.0   w96 19.0 ;

param wt  :=  18REG  47.3    24REG  63.0    24PRO  63.0 ;

param tc  :=  18REG  40.00   24REG  45.00   24PRO  45.00 ;

param dt  :=  18REG 376.0    24REG 172.4    24PRO 316.3 ;

param cpp :=  18REG 102.     24REG  91.     24PRO  91. ;

param dsr := w01 96.  w02 96.  w03 96.  w04 96.  w05 96.
             w62 96.  w76 96.  w96 96. ;

param pt (tr) :
       18REG     24REG     24PRO    :=
w01    1.194     1.429     1.429
w05    1.194     1.509     1.509
w96    0.000     1.600     1.600    ;

param rpc (tr) :
       18REG     24REG     24PRO    :=
w01    2119.      2653.     2617.
w05    2489.      3182.     3176.
w96       0.      2925.     2918.   ;

param opc (tr) :
       18REG     24REG     24PRO    :=
w01    2903.     3585.     3579.
w05       0.        0.        0.
w96       0.     3629.     3622.    ;

param sc  default 99.99 (tr) :
         w01     w02     w03     w04     w05     w62     w76     w96  :=
w01      .      2.97    1.14    2.08    2.37    1.26    2.42    1.43
w02     4.74     .      4.17    6.12    7.41    3.78    7.04    5.21
w03     2.45    4.74     .      3.67    2.84    0.90    2.41    2.55
w04     1.74    5.03    2.43     .      3.19    2.45    2.69    0.58
w05     2.70    5.16    2.84    2.85     .      3.26    3.34    2.71
w06     1.99    4.17    2.13    2.19    2.52    2.06    2.00    1.51
w08     0.21    2.92    1.24    2.07    2.29    1.25    2.32    1.55
w09     0.66    3.76    1.41    2.47    1.82    1.66     .      1.87
w12     1.38    3.83    1.68    2.53    2.39     .      1.96    1.94
w14     2.47    1.58    2.40    3.59    3.85    2.25     .      3.05
w15     1.06    4.95    2.48    1.39    3.41    1.96     .      1.02
w17     0.88    3.39    1.46    2.00    2.67    1.45     .      1.46
w18     7.90    6.57    7.79    9.59   10.81     .       .      6.70
w19     1.42    4.12    1.96    1.99    3.52    1.88     .      1.26
w20     3.03    1.59    2.34    4.76    3.98    1.88     .      3.73
w24     1.58    2.80    2.27    2.87    3.19    1.31     .      2.05
w25     1.51    5.05    2.74    0.57    2.98     .      2.95    0.27
w26     1.75    3.61    2.70    1.54    4.07    3.52     .      1.03
```

```
###  OBJECTIVE  ###

minimize cost:  sum {p in prd, f in fact} rpc[p,f] * Rprd[p,f] +
                sum {p in prd, f in fact} opc[p,f] * Oprd[p,f] +
                sum {p in prd, (d,w) in rt} sc[d,w] * wt[p] * Ship[p,d,w] +
                sum {p in prd, d in dctr} tc[p] * Trans[p,d];
                        # Total cost:  regular production, overtime
                        # production, shipping, and transshipment

###  CONSTRAINTS  ###

rtlim 'regular-time total limits':

    rtmin <= sum {p in prd, f in fact}
                        (pt[p,f] * Rprd[p,f]) / (dp[f] * hd[f]) <= rtmax;

                        # Total crews must lie between limits

otlim 'overtime total limits':

    otmin <= sum {p in prd, f in fact} pt[p,f] * Oprd[p,f] <= otmax;

                        # Total overtime must lie between limits

rlim 'regular-time limits' {f in fact}:

    rmin[f] <= sum {p in prd}
                        (pt[p,f] * Rprd[p,f]) / (dp[f] * hd[f]) <= rmax[f];

                        # Crews at each factory must lie between limits

olim 'overtime limits' {f in fact}:

    omin[f] <= sum {p in prd} pt[p,f] * Oprd[p,f] <= omax[f];

                        # Overtime at each factory must lie between limits

noRprd 'no regular production' {p in prd, f in fact: rpc[p,f] = 0}:

    Rprd[p,f] = 0;

noOprd 'no overtime production' {p in prd, f in fact: opc[p,f] = 0}:

    Oprd[p,f] = 0;      # Do not produce where specified cost is zero

bal 'material balance' {p in prd, w in whse}:

    sum {(v,w) in rt}
       Ship [p,v,w] + (if w in fact then Rprd[p,w] + Oprd[p,w]) =

    dem[p,w] + (if w in dctr then sum {(w,v) in rt} Ship[p,w,v]);

                        # Demand is satisfied by shipment into warehouse
                        # plus production (if it is a factory)
                        # minus shipment out (if it is a distn. center)

trdef 'transshipment definition' {p in prd, d in dctr}:

    Trans[p,d] >= sum {(d,w) in rt} Ship [p,d,w] -
                  (if d in fact then Rprd[p,d] + Oprd[p,d]);

                        # Transshipment at a distribution center is
                        # shipments out less production (if any)

###  DATA -- 3 PRODUCTS  ###

data;

set prd := 18REG 24REG 24PRO ;

set whse := w01 w02 w03 w04 w05 w06 w08 w09 w12 w14 w15 w17
            w18 w19 w20 w21 w24 w25 w26 w27 w28 w29 w30 w31
            w32 w33 w34 w35 w36 w37 w38 w39 w40 w41 w42 w43
            w44 w45 w46 w47 w48 w49 w50 w51 w53 w54 w55 w56
            w57 w59 w60 w61 w62 w63 w64 w65 w66 w68 w69 w71
            w72 w73 w74 w75 w76 w77 w78 w79 w80 w81 w82 w83
            w84 w85 w86 w87 w89 w90 w91 w92 w93 w94 w95 w96
            w98 x22 x23 ;

set dctr := w01 w02 w03 w04 w05 w62 w76 w96 ;
```

```
###  PRODUCT SETS AND PARAMETERS  ###

set prd 'products';     # Elements of the product group

param wt 'weight' {prd} > 0;
                        # Weight in 100 lb / 1000 cases

param cpp 'cases per pallet' {prd} > 0;
                        # Cases of product per shipping pallet

param tc 'transshipment cost' {prd} >= 0;
                        # Transshipment cost in $ / 1000 cases

param pt 'production time' {prd,fact} >= 0;
                        # Crew-hours to produce 1000 cases

param rpc 'regular-time production cost' {prd,fact} >= 0;
                        # Cost of production on regular time,
                        # in $ / 1000 cases

param opc 'overtime production cost' {prd,fact} >= 0;
                        # Cost of production on overtime, in $ / 1000 cases

###  DEMAND SETS AND PARAMETERS  ###

param dt 'total demand' {prd} >= 0;
                        # Total demands for products, in 1000s

param ds 'demand shares' {prd,whse} >= 0.0, <= 1.0;
                        # Historical demand data, from which each
                        # warehouse's share of total demand is deduced

param dstot {p in prd} := sum {w in whse} ds[p,w];
                        # Total of demand shares; should be 1, but often isn't

param dem 'demand' {p in prd, w in whse} := dt[p] * ds[p,w] / dstot[p];
                        # Projected demands to be satisfied, in 1000s

set rt 'shipping routes available' :=
 {d in dctr, w in whse:
         d <> w  and  sc[d,w] < huge  and
         (w in dctr or sum {p in prd} dem[p,w] > 0)  and
         not (msr[d,w] and sum {p in prd} 1000*dem[p,w]/cpp[p] < dsr[d]) };
                        # List of ordered pairs that represent routes
                        # on which shipments are allowed

###  VARIABLES  ###

var Rprd 'regular-time production' {prd,fact} >= 0;
                        # Regular-time production of each product
                        # at each factory, in 1000s of cases

var Oprd 'overtime production' {prd,fact} >= 0;
                        # Overtime production of each product
                        # at each factory, in 1000s of cases

var Ship 'shipments' {prd,rt} >= 0;
                        # Shipments of each product on each allowed route,
                        # in 1000s of cases

var Trans 'transshipments' {prd,dctr} >= 0;
                        # Transshipments of each product at each
                        # distribution center, in 1000s of cases
```

Appendix A. DIST, a product distribution model

This model determines a production and distribution plan to meet given demands for a set of goods. The formulation is motivated by the experiences of a large producer in the United States. The data are for a set of three products.

```
###  SHIPPING SETS AND PARAMETERS  ###

set whse 'warehouses';  # Locations from which demand is satisfied

set dctr 'distribution centers' within whse;

                        # Locations from which product may be shipped

param sc 'shipping cost' {dctr,whse} >= 0;

                        # Shipping costs, to whse from dctr, in $ / 100 lb

param huge 'largest shipping cost' > 0;

                        # Largest cost allowed for a usable shipping route

param msr 'minimum size restriction' {dctr,whse} logical;

                        # True indicates a minimum-size restriction on
                        # direct shipments using this dctr --> whse route

param dsr 'direct shipment requirement' {dctr} >= 0;

                        # Minimum total demand, in pallets, needed to
                        # allow shipment on routes subject to the
                        # minimum size restriction

###  PLANT SETS AND PARAMETERS  ###

set fact 'factories' within dctr;

                        # Locations where product is manufactured

param rtmin 'regular-time total minimum' >= 0;

                        # Lower limit on (average) total regular-time
                        # crews employed at all factories

param rtmax 'regular-time total maximum' >= rtmin;

                        # Upper limit on (average) total regular-time
                        # crews employed at all factories

param otmin 'overtime total minimum' >= 0;

                        # Lower limit on total overtime hours at all factories

param otmax 'overtime total maximum' >= otmin;

                        # Upper limit on total overtime hours at all factories

param rmin 'regular-time minimums' {fact} >= 0;

                        # Lower limits on (average) regular-time crews

param rmax 'regular-time maximums' {f in fact} >= rmin[f];

                        # Upper limits on (average) regular-time crews

param omin 'overtime minimums' {fact} >= 0;

                        # Lower limits on overtime hours

param omax 'overtime maximums' {f in fact} >= omin[f];

                        # Upper limits on overtime hours

param hd 'hours per day' {fact} >= 0;

                        # Regular-time hours per working day

param dp 'days in period' {fact} > 0;

                        # Working days in the current planning period
```

be callable from the system that manages the data. Then AMPL models might be defined, solved and analyzed entirely within a database or spreadsheet environment. Possibly database software could serve to maintain multiple versions of a model that use similar data, as well as multiple versions of data for similar models (see also Dolk 1986).

Algorithm control is another activity with which the language could be more closely integrated. Control software could allow for links between models, as in current systems like GAMS, so that variables from one model become parameters to another. As another example, a postoptimal parametric analysis might be conducted by simply giving the names of AMPL parameters to be varied (and their relative rates of variation, if different).

Finally, we are actively developing extensions to the AMPL language itself. For the very common case of network structures in linear programs, we have provided `arc` and `node` declarations that correspond to familiar descriptions of network-flow variables and balance-of-flow constraints, respectively. For the convex piecewise linear terms that often appear in the objective of an otherwise linear program, we have developed a syntax that permits specification of successive slopes and breakpoints.

We can also easily extend the language to more general kinds of mathematical programs. To accommodate nonlinear programs, virtually no change in the AMPL syntax has been necessary; only a moderate increase in the translator's complexity has been required, to provide for processing of nonlinear expressions and for production of an output that can be read by nonlinear optimizers. We expect no greater difficulty in accommodating certain common constructs of discrete optimization, such as integrality constraints and fixed costs, that have natural representations within an algebraic model.

Acknowledgements

We are grateful to Gary Cramer, Stan Hendryx, Adrian Kester, Doug McIlroy, and Hai-Ping Wu, who have provided us with valuable suggestions and test problems.

- Dummies are widely used by modelers in writing algebraic expressions.
- Complex expressions required by large-scale models remain easy to write using dummy indices, but are often quite awkward to write without them.

Examples of common complexities include dummy indices used in set and arithmetic expressions,

```
izero {p in prd, v in 1..life-1, a in v+1..life}: Inv[p,first+v-1,a] = 0;
```

and in logical expressions to qualify an indexing set:

```
noRprd {p in prd, f in fact: rpc[p,f] = 0}: Rprd[p,f] = 0;
```

Dummy indices are also essential to effective use of the variety of compound sets discussed in Section 2. We might have chosen to make dummies optional in AMPL, so that subscripts could be either the names of sets or the names of indices declared to run over sets; but we felt that the convenience of avoiding dummy indices in some expressions would be greatly outweighed by the confusion of allowing two different conventions.

AMPL also differs from widely used modeling languages in the relational, logical and conditional operators that it provides, and in the set and arithmetic expressions that it allows. Many of these differences are intended to make AMPL more familiar in its resemblance to algebraic notation and more powerful in its ability to describe complex constraints. The success of particular design elements will ultimately have to be judged, however, by the reactions of individual users.

Newer languages under development may be closer to AMPL in some aspects of design. For example, SML (Geoffrion 1988) also enforces the independence of model and data.

7.2 Further development

We have confined the above comparisons to the *languages* that are used by computer systems for linear programming. Yet all of the successful languages are supplied as part of integrated systems, which provide other facilities—for solving linear programs, for manipulating data, and for reporting solutions—that can be as important to users as the convenience of any modeling language. Indeed, there is considerable interest in software that provides an interface between linear programming and existing modeling systems whose languages were developed for other purposes. Examples are What's Best (General Optimization 1986), which works with 1–2–3 spreadsheets, and IFPS/OPTIMUM (Roy, Lasdon and Lordeman 1986) within the IFPS planning system.

In developing AMPL, we have initially concentrated on the language's design and implementation. Thus we have chosen to employ the simplest practicable interfaces to other systems. Our translator accepts tabular data input that can be prepared by any text editor, and its output provides the information needed by most LP optimizers; no special provision is made for maintaining different versions of data and models, or for examination and display of the solutions.

We envision, however, that further development of AMPL software will provide a closer integration of the language with other parts of the modeling environment. We have already mentioned in Section 5 that, as a start, any database or spreadsheet could be used to generate data in our standard format. As a modest further step, the AMPL translator could be modified to read directly from database or spreadsheet files, and a supplementary processor could be designed to write the optimal primal and dual values back to the same files. Standard software could then serve as a tool for examining and reporting solutions as well as for managing data.

A more ambitious plan would modify both the AMPL translator and an LP solver to to give its nonzero values. In contrast, AMPL is designed so that the parameter's name, indexing and other characteristics are declared in the model,

```
param rail_half {plant,plant} >= 0;
```

while the literal numerical values must be given separately as part of the data:

```
param rail_half  default 0  :
           KAFR_EL_ZT  ABU_ZAABAL  HELWAN  ASSIOUT  :=
ABU_ZAABAL      85          .          .        .
HELWAN         142         57          .        .
ASSIOUT        504        420        362        .
ASWAN         1022        938        880      518   ;
```

If users are concerned to be able to state the model and data together as compactly as possible, then it makes sense for a language to employ the same statements in declaring data characteristics as in specifying data values. By comparison, the AMPL user must cite an indexed set or parameter twice—once in the model and once in the data—when providing literal data values as in the example above. We see significant compensating advantages, however, to such an arrangement:

- The statement of the symbolic model can be made compact and understandable, while the bulky and less readable data tables are provided elsewhere.
- The independence of the symbolic model can encourage modelers to specify precise conditions for the validity of the associated data, and can help to prevent accidental changes to the model when only the data are supposed to be changed.
- The separate data specification can be regarded as just a supplementary part of the modeling language. Alternative data formats, tailored to various computing environments, may then be substituted for the standard format (of Section 5) without any change to the fundamental syntax of the modeling language.

The independence of model and data can also make AMPL easier to use in extended modeling exercises, where one typically wants to either run the same model on different data sets, or apply different models to the same data.

Another concise notation in many languages permits the name of a set to also indicate indexing over the set. The overtime limit constraints from our PROD example could be written in GAMS, for instance, as

```
OLIM(TIME).. SUM (PRD, PT(PRD) * OPRD(PRD,TIME)) =L= OL(TIME);
```

These constraints are indexed over the set `TIME`, and the summation is indexed over `PRD`; but the names of these sets are also used as "subscripts" in the parameter expressions `PT(PRD) * OPRD(PRD,TIME)` and `OL(TIME)`. The same thing could be said in less space by giving the sets shorter names:

```
OLIM(T).. SUM (P, PT(P) * OPRD(P,T)) =L= OL(T);
```

By contrast, the equivalent AMPL expression,

```
olim {t in time}: sum {p in prd} pt[p] * Oprd[p,t] <= ol[t];
```

requires that the subscripts be "dummy" indices `t` and `p` that are explicitly declared to run over differently named sets `time` and `prd`. For simple constraint expressions like this, the form *without* dummy indices has advantages of brevity and simplicity. Nevertheless, we have chosen to require the dummy indices in AMPL, for two reasons:

7. Conclusion

We complete our description of AMPL by briefly comparing it, in §7.1, to the languages employed by some of the more widely used linear programming systems. Finally, in §7.2 we indicate several current and likely directions of further development.

7.1 Comparisons

As the introduction to this paper has indicated, AMPL is quite unlike the matrix generation languages that have long been employed in linear programming. AMPL is a *declarative* language that specifies a linear program by directly describing each of its components. Systems such as MaGen (Haverly Systems 1977) and DATAFORM (Management Science Systems 1970, Creegan 1985) are built instead around specialized *programming* languages that describe how to generate a coefficient matrix and other parts of a linear program. Further differences are discussed in detail by Fourer (1983).

AMPL also differs significantly from the matrix-oriented declarative languages provided by recent systems such as PAM (Ketron 1986) and MIMI/LP (Baker and Biddle 1986). These languages specify a linear program by declaring the position and content of nonzero blocks of coefficients in the constraint matrix. Such a specification can be concise and convenient, particularly for models whose block structure is fairly regular.

We have chosen instead to make AMPL a declarative language based on an algebraic description of constraints and objectives. One advantage of the algebraic approach is its familiarity; almost anyone engaged in large-scale linear programming is acquainted with the algebraic form, and hence is also familiar with most of AMPL's conventions. For many people who formulate linear programs by first writing the equations out on paper, the conversion to AMPL may be mostly a matter of transcription. A further advantage lies in the generality of algebraic notation. It extends gracefully to handle complicated kinds of indexing expressions, *ad hoc* logical conditions, and even nonlinear constraints that have no natural representation as a matrix of coefficients.

Numerous modeling systems have been designed to incorporate algebraic languages comparable to AMPL in purpose and design. Some of these systems, of which LINDO (Schrage 1989) is a popular example, do not provide symbolic indexing; all of the numerical parameter values appear directly in the constraints, and each individual constraint must be written out explicitly. Languages without indexing can be simple and convenient, but only for small linear programs.

Closest to AMPL are the algebraic modeling languages that do offer symbolic indexing. Such languages are provided in several commercially distributed modeling systems, including LINGO (Schrage and Cunningham 1988), GAMS (Brooke, Kendrick and Meeraus 1988) and MGG (Simons 1987). AMPL differs most fundamentally from these languages in two respects: it very strongly encourages the independence of model and data, and it distinguishes dummy indices from the sets over which they run.

For purposes of illustration we consider the GAMS language, which is one of the most popular and advanced. In a GAMS model, the same statements may serve both to describe an indexed table of data and to specify the data values. For example, one may write

```
TABLE RAIL-HALF(I,I)
                 KAFR-EL-ZT  ABU-ZAABAL  HELWAN  ASSIOUT
    ABU-ZAABAL         85
    HELWAN            142          57
    ASSIOUT           504         420      362
    ASWAN            1022         938      880      518
```

to simultaneously state that a parameter `RAIL-HALF` exists, to specify its indexing, and running SunOS Release 4.0 and equipped with 12 megabytes of memory and a Motorola 68881 mathematics co-processor. Columns in the first group indicate the sizes of the linear programs: the numbers of rows (constraints), columns (variables, excluding any added by MINOS) and nonzero elements. The middle columns then give the total AMPL model processing times—through *generate,* plus *presolve* if used—and the times in the concluding *output* phase.

In the final group of columns, the first two show the times and numbers of iterations required by MINOS to solve the linear programs. Faster execution could be achieved in many cases by selectively resetting certain parameters of the simplex method, but the timings shown are sufficiently realistic for purposes of comparison with the AMPL times. The MINOS figures do not include reading the input files and writing the solutions; additional times for these tasks appear in the two rightmost columns.

These results vary considerably from one model structure to another. On the whole, however, our timings suggest that the computing cost of converting an AMPL model and data into a linear program is at worst comparable to the cost of solving the linear program, and is often significantly less. As a result, we can conclude that the computer time required for an AMPL application to LP modeling should seldom be more than twice the time required for any alternative approach. Since AMPL is intended to save people's time by giving as much translation work as possible to the computer, we interpret this conclusion as evidence that the use of AMPL is likely to be less costly overall.

expressions. GIT1, GIT2 and GIT3 are shipping models that sum over various slices from a set of quintuples (using the concise notation discussed in §4.2); the first two differ only in that GIT1 expresses certain simple bounds as separate constraints, whereas GIT2 specifies all relevant bounds within the variable declarations. Finally, HPROD1 and HPROD2 are based on a production planning model that makes extensive use of sets generated through set expressions; they differ only in the size of the data.

Table 6–1 presents summary statistics for a pair of runs on each model, the first using the *presolve* phase, and the second without it. All timings are in seconds on a Sun-3/160,

Problem	rows	cols	nonzeros	AMPL	*output*	MINOS	iter	read	write
CMS	1682	24277	142054	232.6	294.3	38249.8	18041	611.1	255.8
CMS	2522	24277	142894	204.1	294.5	39989.0	17292	623.1	264.6
DIST03	185	1090	4325	9.9	8.7	32.1	209	19.8	14.3
DIST03	299	1179	4682	8.9	9.5	44.5	226	21.8	16.0
DIST08	449	2451	9703	19.7	19.6	144.7	473	44.3	31.8
DIST08	790	2728	10792	17.2	21.9	200.6	416	50.2	37.5
DIST13	448	1563	6073	23.0	13.3	102.3	429	28.4	21.8
DIST13	1265	2262	8868	19.6	19.0	236.0	367	43.5	37.1
EGYPT1	146	351	1268	4.8	2.7	20.5	247	6.5	5.1
EGYPT1	285	351	1336	4.5	2.9	32.8	261	7.1	6.3
EGYPT2	146	351	1268	4.4	2.8	20.9	250	6.5	5.2
EGYPT2	285	351	1336	4.1	2.9	35.0	284	7.2	6.3
GIT1	381	1089	4303	14.2	10.2	65.6	397	23.0	15.7
GIT1	1300	1089	5192	12.8	12.0	177.7	334	29.9	24.0
GIT2	377	1089	4293	13.6	9.7	61.7	374	23.0	15.7
GIT2	411	1089	4303	12.4	9.8	70.3	407	23.1	15.9
GIT3	1331	12745	54854	320.3	101.6	3230.3	3966	244.1	138.4
GIT3	1331	12745	54854	309.3	101.8	3230.3	3966	244.1	138.4
HPROD1	128	140	1383	10.2	3.5	3.6	17	6.6	2.8
HPROD1	129	140	1384	9.9	3.5	3.7	17	6.5	2.7
HPROD2	231	367	3445	30.6	7.4	19.5	90	16.1	5.9
HPROD2	232	367	3446	30.0	7.4	19.6	90	15.9	5.8
OIL	43	59	232	1.4	0.4	1.6	41	1.2	1.1
OIL	47	60	236	1.3	0.4	1.7	49	1.2	1.1
PROD03	179	231	870	1.9	1.9	12.2	127	4.5	4.4
PROD03	210	235	922	1.8	2.0	12.2	109	4.8	4.7
PROD08	418	551	2163	3.5	4.6	36.4	178	11.1	10.5
PROD08	470	560	2262	3.1	4.8	40.8	189	11.6	10.8
PROD13	648	871	3438	5.3	7.2	84.7	289	17.6	16.4
PROD13	730	885	3602	4.5	7.6	84.7	260	18.3	16.9
TRAIN1	194	411	1058	3.1	2.9	4.8	31	6.3	6.4
TRAIN1	413	411	1277	2.8	3.4	7.7	31	8.4	8.2
TRAIN2	194	410	1055	7.6	2.7	5.8	48	6.3	6.2
TRAIN2	413	411	1277	7.2	3.2	9.9	48	8.2	8.2

Table 6–1. *Timings in Sun-3/160 seconds, with and without* presolve.

since the removal of variables may result in additional equality constraints that involve just one variable.

- Converting to bounds any inequality constraints that involve just one variable, as discussed in Section 4. Such bounds are combined with any that have been specified in a variable's declaration. Additional constraints of this kind may also be found in multiple passes as variables are fixed.
- Converting to ranges or equalities any double-inequality constraints, as explained in Section 4.
- Removing certain inequality constraints that must be slack in any feasible solution. Such constraints are identified by substituting upper or lower bounds (as appropriate) for the variables. (This test may also identify certain constraints that can never be satisfied, in which case the user is warned of a mis-specified model.)

Except for the dropping of unused variables, these transformations are optional. Our computational experience indicates that the resulting simplified linear program is seldom any harder to solve, and is sometimes much easier.

The final *output* phase makes the translated model available to an optimizer. Our intent is to let different applications link appropriate output routines with the AMPL translator. Initially we have provided an output routine suitable for use with optimizers that read linear programs in the standard MPS form (as described by Murtagh 1981 and by the reference manuals of many linear programming systems). Our routine generates arbitrary 8-character row (constraint) and column (variable) names to conform to the MPS format. It writes one file containing the MPS output, and supplementary files providing a variety of information:

- The "true" row and column names from the AMPL model.
- The constant term from each objective (not needed for purposes of optimization).
- Names of constraints eliminated by *presolve.*
- Names of unused variables dropped by *presolve.*
- Names and values of fixed variables removed by *presolve.*

The output also includes a "SPECS" file suitable for use by the MINOS optimizer (Murtagh and Saunders 1987).

6.2 Tests

We modified version 5.3 of the MINOS optimization system to read the MPS, name and SPECS files described above, and to cite AMPL variable and constraint names in its solution report. We then tested our implementation by translating and solving some linear programs ranging up to over a thousand constraints, and over ten thousand variables. Statistics for these tests are collected in Table 6–1.

The model called EGYPT2 is the one reproduced in the Appendix; EGYPT1 is an alternative version using ordered pairs instead of sets of sets, but the same data. Among the other models quoted in this paper (and listed in Fourer, Gay and Kernighan 1987), the sequences DIST03, DIST08, DIST13 and PROD03, PROD08, PROD13 are our DIST and PROD examples with progressively larger data sets (based on 3, 8 and 13 products). TRAIN1 and TRAIN2 are versions of our TRAIN example.

Among the other test problems, OIL is a small refinery model (Kendrick, Meeraus and Suh 1981). CMS is a cash management model that has an interesting variety of indexing

6. Implementation

In addition to designing the AMPL language, we have implemented a complete AMPL translator. This section briefly summarizes important aspects of our linear programming implementation. We describe the processing steps of the translator in §6.1 and give some statistics on its efficiency in §6.2.

6.1 Processing

A modeler initiates an application of AMPL by working out a representation of the model as described in Sections 2–4, and a representation of the data such as that presented in Section 5. These representations are stored in model and data files, which can be created by use of any text editor.

The primary job of the AMPL translator is to read the model and data files, and to write a representation of a linear program suitable for use by optimization algorithms. The translator must also store enough model information to allow for an understandable listing of the optimal solution. Our translator implementation carries out this work in seven logical phases: *parse, read data, compile, generate, collect, presolve,* and *output.*

The *parse* phase reads the model file and parses it into a corresponding expression tree. It uses a LEX program for lexical analysis, and a parser produced by the YACC parser generator. LEX and YACC substantially simplify the job of language development, particularly in the early stages when the syntax undergoes frequent changes (Kernighan and Pike 1984).

Next, the *read data* phase carries out the initial input and checking of the data. Since the parameters and sets are allowed to appear in any order within the data file, the data cannot be thoroughly checked until all have been read. Hence this phase merely checks that nothing is defined twice, that no set contains duplicate elements, that set members have the right dimensions, and that the values supplied for numerical parameters are numbers.

The *compile* phase transforms the expression tree (created by *parse*) to permit efficient computation of sets, parameters, variables, constraints and other derived entities. It moves invariants out of loops, collects common subexpressions, and combines the work of retrieving parameter values indexed over the same set (as in the case of `rmin[f]`, `dp[f]`, `hd[f]` and `rmax[f]` from the `rlim` constraint of PROD). Since the content of index sets depends on the data file, *compile* must follow *read data.*

In the subsequent *generate* phase, the data are thoroughly checked and all derived entities are computed. The checking portion of this phase verifies all conditions imposed by the model (such as integrality and nonnegativity restrictions on parameters) and ensures that indices are valid for the model components that they are indexing.

At the conclusion of *generate,* the translator has created a list of all linear terms in the model. A variable may still appear, however, in two or more terms within the same objective or constraint. The *collect* phase merges such multiple appearances into one, and sorts each variable's coefficients to match the ordering given to the constraints.

After completing the first five phases, the translator has determined the linear program that was implied by the model and data files. The *presolve* phase next performs various transformations that may make this LP smaller or otherwise easier to solve:

- Dropping variables that appear in no constraint or objective.
- Removing variables that are fixed at a single value, either by upper and lower bounds that are equal, or by equality constraints that involve just one variable. In the latter case, the constraints are also dropped. This activity may proceed for several passes,

parentheses; each slice is through two cities, and consists of pairs of times. Following each pair of times, however, the parameter specification also gives a demand value.

Many other arrangements of slices are possible. Slicing can even be valuable in specifying sparse collections of parameters indexed over only two sets. In the EGYPT model, the plant-specific prices for raw materials could have been given in a table like this:

```
param p_pr default 0.0 :
             LIMESTONE  COKE_GAS  EL_ASWAN  PHOS_ROCK  :=
KAFR_EL_ZT       .          .         .        5.0
ABU_ZAABAL       .          .         .        4.0
HELWAN          1.2       16.0        .         .
ASSIOUT          .          .         .        3.5
ASWAN           1.2         .        1.0        .      ;
```

Instead, they are condensed into four slices, one for each raw material:

```
param p_pr  default 0.0  :=
 [HELWAN,COKE_GAS]              16.0
 [ASWAN,EL_ASWAN]                1.0
 [*,LIMESTONE]      ASWAN        1.2
                    HELWAN       1.2
 [*,PHOS_ROCK]      ABU_ZAABAL   4.0
                    ASSIOUT      3.5
                    KAFR_EL_ZT   5.0 ;
```

The choice between these representations is a matter of convenience or readability. Two larger examples also appear in the EGYPT data, for parameters `io` and `dcap`.

eter values are zero for many combinations of the indices. We then find it convenient to specify a *default* value of zero, and to give only the nonzero values explicitly. As an example, the EGYPT model declares interplant rail distances by

```
param rail_half {plant,plant} >= 0;
param rail {p1 in plant, p2 in plant} :=
   if rail_half[p1,p2] > 0 then rail_half[p1,p2] else rail_half[p2,p1];
```

and specifies the data as follows:

```
set plant := ASWAN HELWAN ASSIOUT KAFR_EL_ZT ABU_ZAABAL ;

param rail_half default 0 :
            KAFR_EL_ZT  ABU_ZAABAL  HELWAN  ASSIOUT  :=
ABU_ZAABAL      85          .          .        .
HELWAN         142         57          .        .
ASSIOUT        504        420        362        .
ASWAN         1022        938        880      518   ;
```

The phrase `default 0` specifies that the default value is zero. Parameters are assigned this value in two ways. First, each appearance of `.` in the table indicates the default value; `rail_half[ABU_ZAABAL,ABU_ZAABAL]` and five other parameters are set to zero by this device. Second, parameters that fail to appear in the table are automatically given the value zero; these include `rail_half[ASWAN,ASWAN]` and eight others.

Several other examples of default zeros can be seen in the EGYPT model. Parameter `sc` in DIST uses a default of 99.99 to represent "huge" shipping costs on routes where shipment is not allowed. In some of these examples, the use of the default symbol rather than 0.0 or 99.99 serves mainly to make the tables more readable; it can be changed from `.` to any preferred symbol.

For parameters subscripted by three or more indices, the data values must be specified in a series of slices. Each slice, in turn, is represented as a list or table. Both the ideas and the syntax of slicing are much the same as given for sets in §5.1 above. In TRAIN, for instance, demands are declared by

```
param demand {schedule} > 0;
```

and the specification of the demand data begins as follows:

```
param demand :=
 [WA,*,PH,*]   2  5    .55      6  9    .01      8 11    .01
              10 13    .13     12 15   1.59     13 16   1.69
              14 17   5.19     15 18   3.55     16 19   6.29
              17 20   4.00     18 21   5.80     19 22   3.40
              20 23   4.88     21 24   2.92     22 25   4.37
              23 26   2.80     24 27   4.23     25 28   2.88
              26 29   4.33     27 30   3.11     28 31   4.64
              29 32   3.44     30 33   4.95     31 34   3.73
              32 35   5.27     33 36   3.77     34 37   4.80
              35 38   3.31     36 39   3.89     37 40   2.65
              38 41   3.01     39 42   2.04     40 43   2.31
              41 44   1.52     42 45   1.75     44 47   1.88
              46  1   1.05

 [PH,*,NY,*]   1  3   1.05      5  7    .43      9 11    .20
              11 13    .21     13 15    .40     14 16   6.49 ...
```

This listing has much in common with the listing of the members of `schedule` in §5.1. The slice-defining templates are the same, except that they are enclosed in brackets rather than

In this special form of the `param` statement, each column gives values for a different parameter; the parameter names appear as labels at the top of the columns.

For parameters indexed over two sets, the data are naturally presented in a table like the following from Figure 1–4:

```
param units :    nuts    bolts   washers  :=
         iron     .79     .83      .92
         nickel   .21     .17      .08    ;
```

The row labels indicate the first index, and the column labels the second index. Thus `units[iron,bolts]` is .83.

Several further options can be useful in the generation or display of certain tables, particularly when the rows would be very long. First, a table may be *transposed;* in PROD, as an example, the demands are declared as

```
param dem {prd,first..last+1} >= 0;
```

and the data specification is as follows:

```
param dem (tr) :
          18REG     24REG      24PRO   :=
  1        63.8    1212.0        0.0
  2        76.0     306.2        0.0
  3        88.4     319.0        0.0
  4       913.8     208.4        0.0
  5       115.0     298.0        0.0
  6       133.8     328.2        0.0
  7        79.6     959.6        0.0
  8       111.0     257.6        0.0
  9       121.6     335.6        0.0
  10      470.0     118.0     1102.0
  11       78.4     284.8        0.0
  12       99.4     970.0        0.0
  13      140.4     343.8        0.0
  14       63.8    1212.0        0.0   ;
```

The qualifier `(tr)` says that the column labels indicate the first index and the rows labels the second, just the opposite of the arrangement in the preceding example. The same data could be given in an untransposed form, but then the table would have 3 rows and 14 columns, and would not be nearly so easy to display or edit. If the transposition option were not available, we could instead change the model to declare `param dem {first..last+1,prd}`. We prefer, however, to let the algebraic model be declared in the most natural way, regardless of how the data will be represented. (In PROD we maintain the convention that, where a model component is indexed over both products and times, the set of products comes first.)

Sometimes even transposing a table would leave its rows uncomfortably long. Then the table may be divided column-wise into several smaller ones, as can be seen in the data for `cf75` from the EGYPT model (Appendix B). Or, since line breaks are not significant, each of the rows may be divided across several lines of the data file.

When reading data, our AMPL translator makes no assumptions about the values of any parameters. If a parameter is not assigned a value in the model or in the data file, then any attempt to use its value in the model will be rejected as an error. In this way some kinds of discrepancies in the data, such as missing rows or columns of a table, can be caught during the execution of the translator.

In some applications certain indexed collections of parameters are “sparse”: the param-

```
set schedule :=
   (WA,*,PH,*)   2  5      6  9      8 11     10 13
                12 15     13 16     14 17     15 18
                16 19     17 20     18 21     19 22
                20 23     21 24     22 25     23 26
                24 27     25 28     26 29     27 30
                28 31     29 32     30 33     31 34
                32 35     33 36     34 37     35 38
                36 39     37 40     38 41     39 42
                40 43     41 44     42 45     44 47
                46  1
   (PH,*,NY,*)   1  3      5  7      9 11     11 13
                13 15     14 16     15 17     16 18 ...
```

The *template* `(WA,*,PH,*)` indicates a slice through `WA` in the first coordinate and `PH` in the third; various pairs of values for the second and fourth coordinates are then supplied. A list specification for the same set would begin as

```
set schedule := (WA,2,PH,5) (WA,6,PH,9) (WA,8,PH,11)
     (WA,10,PH,13) (WA,12,PH,15) (WA,13,PH,16) (WA,14,PH,17) ...
```

The sliced representation is clearly easier to read, and is perhaps also easier to create. Our data format offers a variety of slicing options in addition to those shown here; slices having two coordinates free may be described in tables as well as in lists.

The entire set `schedule` is specified as a union of six different slices (of which only two are seen above). Generally, any series of slices may be used to define a set, so long as no member is given twice; different slicing options may even be used in the specification of the same set. Thus a large set of triples or quadruples can be specified in many ways. The choice is determined by the modeler's convenience; as an example, if the set members are maintained in a database then it may be easier to slice along coordinates that correspond to sort keys.

5.2 Parameters

Simple, unindexed parameters are assigned values in an obvious way, as shown by these examples from Figure 1–4:

```
param T := 4;
param max_prd := 123.7;
```

Most of a typical model's parameters are indexed over sets, however, and their values are specified in a variety of one-dimensional lists and two-dimensional tables.

The most elementary case is a parameter indexed over a single set, such as

```
param init_stock :=
        iron    35.8
        nickel   7.32 ;
```

Line breaks are disregarded, so this statement could be put all on one line:

```
param init_stock := nickel 35.8 iron 7.32 ;
```

In Figure 1–4, `init_stock` is indexed over the same set as the parameters `cost` and `value`. Thus a single table can conveniently give the data for all:

```
param :         init_stock    cost    value  :=
        iron       35.8       .03      .02
        nickel     7.32       .025    -.01   ;
```

5. Data

As we have emphasized in earlier sections, specific set and parameter data must be combined with an AMPL model to describe one particular linear program. Data values have varied sources, but usually a computer is used to help collect and organize them. Database software is often employed for this purpose, and spreadsheet programs are also proving to be convenient. In an ideally integrated system, a modeling language translator would have some direct connection to the database or spreadsheet software; either the translator would read their files, or it would be invoked by them as a subroutine.

Even if interfaces to specialized software were available, however, it would be desirable that a modeling language also support some simple, standard format for data files. The availability of such a format has several benefits: permitting the translator to run in the greatest variety of environments, encouraging exchange of models for educational purposes, and facilitating the collection of standard models for the testing of algorithms.

As part of our initial implementation, therefore, we have designed a standard AMPL data file format. Our format supports several natural ways of specifying set and parameter values, using one-dimensional lists and two-dimensional tables. Wherever possible, similar syntax and concepts are used for both the set and parameter statements. Files in our format can be created by any text editor; they are also fairly easy to generate as the output of database and spreadsheet programs.

Examples of data in standard AMPL format appear in Figure 1–4 and in the Appendices. We comment on the specification of set members in §5.1 below, and on the specification of parameter values in §5.2.

5.1 Sets

The members of a simple set are specified straightforwardly. In data for the EGYPT model, as an example, the sets of nutrients and processes are given by

```
set nutr := N P205 ;

set proc := SULF_A_S SULF_A_P NITR_ACID AMM_ELEC AMM_C_GAS
     CAN_310 CAN_335 AMM_SULF SSP_155 ;
```

The same approach serves to specify the members of each set belonging to an indexed collection of sets:

```
set p_except[HELWAN] := CAN_310 ;
set p_except[ASWAN] := CAN_335 ;
```

The data for a set of pairs can be organized as either a one-dimensional *list* or a two-dimensional *table*. For example, if `p_except` were defined as a set of pairs (as suggested in §2.4) then the above data could be listed as

```
set p_except := (ASWAN,CAN_335) (HELWAN,CAN_310) ;
```

or written in a table as

```
set p_except :  CAN_335  CAN_310  :=
         ASWAN     +        -
         HELWAN    -        +     ;
```

In the table, a `+` indicates a pair that is in the set, and a `-` indicates a pair that is not in the set.

Sets of triples and longer compound members are usually most conveniently presented in several "slices" along certain coordinates. In the TRAIN model, the `schedule` parameter is a set of quadruples, and the specification of its members in the data file begins as follows:

Sets of sets, rather than ordered pairs, could have been used in formulating DIST. Suppose that `rt[d]` were declared as the set of warehouses to which products may be shipped directly from distribution center `d`. Then the constraint `bal` in DIST would be

```
bal {p in prd, w in whse}:
     sum {v in dctr: w in rt[v]} Ship[p,v,w] +
     (if w in fact then Rprd[p,w] + Oprd[p,w]) =
          dem[p,w] + (if w in dctr then sum {v in rt[w]} Ship[p,w,v]);
```

The slice `{v in rt[w]}` along the first coordinate is noticeably easier to specify than the slice `{v in dctr: w in rt[v]}` along the second coordinate. The symmetry between the two kinds of slices is lost.

The EGYPT model can also be formulated in terms of either ordered pairs or sets of sets, but its situation is different. The sets of pairs do not represent network flows, and can be arranged so that they are always sliced on the first coordinate when used in the constraints. The material balance constraints for commodities provide an extended example:

```
subject to mb {c in commod, pl in plant}:
   sum {pr in p_pos[pl]} io[c,pr] * Z[pl,pr]
     + ( if c in c_ship then
         ( if pl in cp_pos[c] then sum {p2 in cc_pos[c]} Xi[c,pl,p2] )
       + ( if pl in cc_pos[c] then sum {p2 in cp_pos[c]} Xi[c,p2,pl] ))
     + ( if (c in c_raw and pl in cc_pos[c]) then
         (( if p_imp[c] > 0 then Vr[c,pl] )
       + ( if p_dom[pl,c] > 0 then U[c,pl] )))
   >= if (c in c_final and pl in cp_pos[c]) then sum {r in region} Xf[c,pl,r];
```

Here sets of sets offer a natural and concise notation that may make them preferable to ordered pairs.

4.3 Objectives

A linear program's objective function has all the properties of a constraint, except that it lacks a relational operator. Thus the declaration of an objective has the same form as the declaration of a constraint, except for starting with the keyword `minimize` or `maximize` rather than `subject to`.

Although a linear program need only have a single objective, AMPL permits the declaration of any number of alternative objectives, either singly or in indexed collections. Thus TRAIN has both an objective to represent total cars in the fleet, and an objective to represent total car-miles traveled in the schedule; in the application for which it was developed, these objectives were traded off against each other by use of a parametric simplex algorithm.

By use of a conditional expression, these variants can be declared together:

```
trdef {p in prd, d in dctr}:
     Trans[p,d] >= sum {(d,w) in rt} Ship [p,d,w] -
                  (if d in fact then Rprd[p,d] + Oprd[p,d]);
```

For `d` not in `fact`, the value of the `if...then...` expression is zero—since it has no `else`—and so the production term `Rprd[p,d] + Oprd[p,d]` is omitted. An example with three variants is seen in the material balance constraints for each product at each warehouse:

```
bal {p in prd, w in whse}:
   sum {(v,w) in rt} Ship[p,v,w] +
   (if w in fact then Rprd[p,w] + Oprd[p,w]) =
      dem[p,w] + (if w in dctr then sum {(w,v) in rt} Ship[p,w,v]);
```

The first `if` represents production at `w`, which can occur only if `w` is a factory; the second `if` represents shipments from `w` to other warehouses, which can occur only if `w` is a distribution center.

The above examples also show how ordered pairs are commonly used in constraints. In `bal` there are two sums that involve the set `rt` of pairs:

```
sum {(v,w) in rt} Ship[p,v,w]
sum {(w,v) in rt} Ship[p,w,v]
```

These sums lie within the scope of the constraint's overall indexing expression, `{p in prd, w in whse}`, which has already defined the dummy index `w`. Hence, for a particular warehouse `w`, the first sum is over all `v` such that `(v,w)` is a pair in `rt`; in other words, it is a sum over all distribution centers that ship *to* `w`. The second sum is over all `v` such that `(w,v)` is a pair in `rt`, or equivalently over all warehouses that receive *from* `w`. In effect, the first sum's indexing expression takes a "slice" from `rt` in the second coordinate, while the second sum's indexing expression takes a slice in the first coordinate. This kind of arrangement, with slices in first one coordinate and then the other, is likely to be found in any network application that uses sets of pairs to specify the arcs.

AMPL's concise notation for slicing pairs does introduce a certain ambiguity, in expressions like

```
sum {(v,w) in rt} ...
```

If this phrase lies within the scope of another indexing expression that has already defined the dummy index `w` (as in the case of `bal`) then the summation is over a slice through `rt`. On the other hand, if the `sum` does not lie in any scope defining `w`, then the summation is over *all* ordered pairs in `rt`.

As an alternative, we have considered requiring a more explicit indexing expression in summations over slices, so that the `bal` constraint, for instance, would have to begin as

```
bal {p in prd, w in whse}:
     sum {v in dctr: (v,w) in rt} Ship[p,v,w] + ...
```

or

```
bal {p in prd, w in whse}:
     sum {(v,w1) in rt: w1 = w} Ship[p,v,w] + ...
```

The latter is rather awkward, however, while the former works only in models (such as DIST) that explicitly define the set of all `v` such that `(v,w)` is in the set `rt`. We have come to believe that the simplicity and generality of the more concise slice notation outweigh any disadvantages arising from its ambiguity. (The advantages are even more evident in the TRAIN model, which takes sums over slices through ordered quadruples.)

total hours of work required by production may not exceed the hours available from all crews employed:

```
rlim {t in time}:
     sum {p in prd} pt[p] * Rprd[p,t] <= sl * dpp[t] * Crews[t];
```

For each product in each period, all previously produced inventory, plus any initial inventory still unused, must total at least the required minimum inventory:

```
ireq {p in prd, t in time}:
     sum {a in 1..life} Inv[p,t,a] + iil[p,t] >= minv[p,t];
```

The amount of inventory that is `a` periods old at the end of period `t` cannot exceed the amount that was `a-1` periods old at the end of period `t-1`:

```
ilim {p in prd, t in first+1..last, a in 2..life}:
     Inv[p,t,a] <= Inv[p,t-1,a-1];
```

The AMPL translator determines which variables have nonzero coefficients in the constraints implied by these declarations; it then computes the coefficients of these variables and the value of the constant term. In our implementation, the coefficients are determined as if all variables had been moved to the left of the relational operator, and all constants to the right, but such transformations need not concern the modeler. Any expression in parameters and variables is acceptable as a constraint, so long as the expressions on each side of the relation can be interpreted as linear. For instance, the language allows a parameter to multiply a sum of variables, or a variable to be divided by a parameter.

Certain pairs of related constraints are most conveniently expressed as double inequalities. In PROD, the number of crews employed each period must lie between the minimum and maximum for that period:

```
emplbnd {t in time}: cmin[t] <= Crews[t] <= cmax[t];
```

If `cmin[t]` is less than `cmax[t]` then this constraint gives upper and lower bounds for variable `Crews[t]`; if `cmin[t]` equals `cmax[t]` then `Crews[t]` is effectively fixed at their common value. Two further possibilities occur in DIST, where the number of crews required to carry out all regular-time production in a period must lie within specified bounds at each factory:

```
rlim {f in fact}: rmin[f] <=
     sum {p in prd} (pt[p,f] * Rprd[p,f]) / (dp[f] * hd[f]) <= rmax[f];
```

Because only the middle expression contains variables, this double inequality can be treated as a single constraint. For each member `f` of `fact`, if `rmin[f]` is less than `rmax[f]` then the associated constraint is an inequality whose slack is bounded by `rmax[f] - rmin[f]` (that is, a *range* in traditional linear programming terminology); if `rmin[f]` equals `rmax[f]` then the constraint is just an equality.

Altogether, from the standpoint of an algorithm for solving linear programs, there are four kinds of double inequalities that may be treated in four different ways. From a modeler's standpoint, however, these are all just linear constraints. Hence the AMPL language provides no special syntax for distinguishing one kind from another; the distinction is made optionally by the model translator, as Section 6 will explain.

Some kinds of constraints have two or more closely related variants. In DIST, as an example, there are two varieties of transshipment constraints. At each ordinary distribution center, the amount of each product transshipped must equal at least the amount shipped out of the center; at each factory distribution center, the amount transshipped must equal at least the amount shipped out of the center *minus* the amount produced at the factory.

4. The Linear Program

The most complicated components of linear programs are the constraints. Thus, following a brief consideration of the variables in §4.1, most of this section is concerned with constraint declarations. In §4.2 we first examine some fairly straightforward indexed collections of algebraic constraints and their transcriptions into AMPL. We then investigate the issues that must be resolved in handling three common but more difficult cases: double inequalities, optional linear terms, and "slices" over compound sets.

To conclude this section, a few comments on the specification of the objective are collected in §4.3.

4.1 Variables

The variables of an algebraic linear programming model are described in much the same way as the numerical data. Thus an AMPL declaration for a variable consists of the keyword `var` followed by an identifier, followed by the same kinds of optional indexing and qualification expressions that might appear in a `param` declaration.

A few variables in the EGYPT model are neither indexed nor qualified:

```
var Psip;
```

However, most of the variables in a large linear program are defined as indexed collections, and are nonnegative. Thus PROD, for example, contains the following declarations:

```
var Crews {first-1..last} >= 0;
var Hire {time} >= 0;
var Rprd {prd,time} >= 0;
var Inv {prd,time,1..life} >= 0;
```

The qualification `>= 0` represents a simple constraint on these variables. Because nonnegativity is so common, and because it is handled implicitly by nearly all algorithms for linear programming, it is almost always specified as part of a variable's declaration rather than as an explicit constraint.

More generally, the qualification expression may specify any lower or upper bound on each variable. Two such expressions may be given to specify both a lower and an upper bound. Alternatively, one or both one or both bounds may be described by explicit constraints, as discussed further below.

4.2 Constraints

A linear constraint says that one linear arithmetic expression is equal to, greater than or equal to, or less than or equal to another linear arithmetic expression. Typical linear programs have few distinct *kinds* of constraints; 10 kinds or fewer is normal, and 20 is large. Each kind of constraint can be represented by a single symbolic equality or inequality in the parameters, the variables and one or more dummy indices. As the dummy indices run over certain sets, a symbolic constraint gives rise to many explicit ones.

The AMPL transcription of a constraint declaration thus has two major parts: an indexing expression, identical in form to those found elsewhere in an AMPL model, and a comparison expression using `=`, `<=` or `>=`. Preceding these are the optional keywords `subject to`, and a constraint identifier. (In the declaration of the model, the constraint identifier serves only as a syntactic place-holder, and perhaps as a suggestive name for documentary purposes. The identifier could also be useful, however, as a name for the constraint in reports of slack and dual values following optimization.)

Several typical uncomplicated constraints are found in PROD. For each planning period, member `t` from `time`, and the result assigned to `minv[p,t]`. The amount of initial inventory available for allocation after period `t` is also a computed parameter; it is the initial inventory (if any) that remains after deducting the demand for the first `t` periods:

```
param iil {p in prd, t in time}
     := iinv[p] less sum {v in first..t} dem[p,v];
```

Both `minv` and `iil` subsequently appear in the inventory requirement constraints, where the constant term is `minv[p,t] - iil[p,t]`.

Any attempt to provide explicit values for `minv` or `iil`, in the specification of the model's data, will be rejected as an error. However, if the keyword `default` is employed in place of the `:=` operator, then values for some or all of the parameters `minv[p,t]` and `iil[p,t]` may be given along with the other data, and will override the computed values.

As another alternative, a separate preprocessing program could be used to compute all the values for parameters such as `minv` and `iil`, in which case they could be treated like any other data parameters in the AMPL model. Such an approach is unavoidable when the computations involve something more complicated than the evaluation of an arithmetic expression (such as the application of an algorithm). We prefer, however, to represent the "raw" data as parameters in the model whenever possible; then any arithmetic processing of these data must appear explicitly in the model's declarations, which are easy to read and check for validity. The power and variety of AMPL's arithmetic expressions should tend to encourage this practice.

important for one of our larger test problems that manipulates a set of quintuples.)

Finally, there are instances in which a parameter's value must depend on some logical condition. As an example, `road[r,pl]` is the distance from plant `pl` to region `r`, or zero if plant `pl` is in region `r`. Transportation from a plant to a region incurs a fixed cost plus a cost proportional to distance, but only if the plant is outside the region. The arithmetic expression for transportation cost is thus as follows:

```
if road[r,pl] > 0 then .5 + .0144 * road[r,pl] else 0
```

If the condition between `if` and `then` is true, then the entire expression takes the value between `then` and `else`; if instead the condition is false, then the expression takes the value after `else`. The entire `if...then...else...` construct may itself serve as an operand wherever appropriate; EGYPT uses a sum of conditionals,

```
  (if impd_barg[pl] > 0 then 1.0 + .0030 * impd_barg[pl] else 0)
+ (if impd_road[pl] > 0 then 0.5 + .0144 * impd_road[pl] else 0)
```

to combine the costs for barge and road transportation of imported raw materials.

The `else` part of a conditional expression may be omitted, in which case `else 0` is assumed. This default is particularly convenient in specifying optional terms of constraints, as will be seen in Section 4. The scope of the expression following `else` (or `then`, if there is no `else`) is to the end of the expression that follows. Thus `if...then...` and `if...then...else...` constructs are normally parenthesized to make their scope clear.

3.3 Computed parameters

It is seldom possible to arrange that the data available to a model are precisely the coefficient values required in the objective and constraints. Thus the coefficients are often specified by expressions in the parameters. For example, PROD gives the total regular wages for crews in period `t` as

```
rtr * sl * dpp[t] * cs * Crews[t]
```

where `rtr` is the wage rate per worker in dollars per hour, `sl` is the number of hours in a daily shift, `dpp[t]` is the number of days in the period, `cs` is the number of workers in a crew, and `Crews[t]` is a variable that stands for the number of crews in the period. Expressions also appear as constant terms (right-hand sides, in LP terminology) for the constraints. In the first-period demand requirement constraint for product `p`, the term

```
dem[p,first] less iinv[p]
```

is evaluated as demand minus initial inventory if demand exceeds initial inventory, or zero otherwise.

Although any parameter expression may be used in the objective and constraints, the expressions are best kept simple as in the examples above. 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.

Declarations for computed parameters are much like the declarations for computed sets that were seen in the previous section. In PROD, the minimum inventory for product `p` in period `t` is defined to be its demand in the following period times either `pir` or `rir`, depending on whether it will be promoted:

```
param minv {p in prd, t in time}
     := dem[p,t+1] * (if pro[p,t+1] then pir else rir);
```

The expression following `:=` is evaluated for each combination of a member `p` from `prd` and a

Iterated operator	Underlying binary operator
`sum`	`+`
`prod`	`*`
`min`	
`max`	
`exists`	`or`
`forall`	`and`

Table 3–2. *Iterated operators.*

Expressions can also be built by iterating certain operations over sets. Most common is the iterated summation, represented by a Σ in algebraic notation and by `sum` in AMPL:

```
sum {p in prd} dem[p,w]
```

Any indexing expression may follow `sum`. The subsequent arithmetic expression is evaluated once for each member of the index set, and all the resulting values are added. Thus the above sum from DIST represents the total demand for all products at warehouse `w`. In precedence the `sum` operator lies between binary `+` and `*`, so that the expression following `sum` includes everything up to the next $+$ or $-$ not within a parenthesized subexpression.

By allowing any indexing expression after `sum`, AMPL provides a general and flexible notation for summations. Even the complicated sums in linear constraints can be transcribed straightforwardly, as examples in Section 4 will show. Moreover, the generality of this notation actually makes the language simpler, in that the rules for indexing a `sum` are no different from the rules for indexing a `param` declaration. The AMPL user needs to learn only one syntax for index sets.

Other associative, commutative operators can be iterated just like `sum`. Table 3–2 shows those available in AMPL for arithmetic and logical operations. An example of `forall`, an iterated operator that returns a logical result, is found in the EGYPT model:

```
forall {u in unit: util[u,pr] > 0} u in m_pos[pl]
```

Given a process `pr` and a plant `pl`, this expression is true if and only if, for every member `u` of `unit` such that `util[u,pr]` is positive, `u` is also a member of the set `m_pos[pl]`. In other words, since `m_pos[pl]` is the set of units for which initial capacity exists at plant `pl`, while `util[u,pr]` is positive exactly when process `pr` requires unit `u`, the expression is true if and only if there is initial capacity at the plant for every unit required by the process.

(There are also iterated union and intersection operations on sets, as well as an iterated `setof` operator that builds sets from arbitrarily specified members. As an illustration, the TRAIN model defines a set `links` of all city pairs that may appear in the schedule; we have specified this set as part of the data, but it could instead be computed as

```
set links := setof {(c1,t1,c2,t2) in schedule} (c1,c2);
```

The same set could be defined by use of the iterated `exists` operator:

```
set links := {c1 in cities, c2 in cities:
     exists {t1 in times, t2 in times} (c1,t1,c2,t2) in schedule};
```

The `setof` expression is easier to read, however, and permits the AMPL translator to carry out the computation of `links` more efficiently. These concerns have proved particularly the first:

```
param first > 0 integer;
param last > first integer;
```

Similarly, in every period, the maximum crew size must be no less than the minimum:

```
param cmin {time} >= 0;
param cmax {t in time} >= cmin[t];
```

In a few cases, simple inequalities are insufficient to express the desired restrictions. To accommodate all possibilities, we have included in our design a separate `check` statement that may accompany a parameter declaration; as an example, the following could be used to require that the minimum crew sizes be nondecreasing:

```
param cmin {first..last};
     check {t in first..last-1} cmin[t] <= cmin[t+1];
```

In general, a `check` statement can appear at any convenient place in an AMPL model, and can specify any logical condition on the sets and parameters; the translator tests all such conditions and reports any violations.

The parameters of an AMPL model most commonly represent numerical values. However, parameters may be declared `symbolic` to specify that their values are arbitrary objects such as might be the members of any set.

3.2 Arithmetic and logical expressions

Arithmetic expressions in AMPL evaluate to floating-point numbers. Any parameter reference or numerical literal (`17`, `2.71828`, `1.0e+30`) is an arithmetic expression by itself. Common arithmetic functions of one variable (`abs`, `ceil`, `floor`) and of two or more variables (`min`, `max`) are also expressions, as seen in TRAIN. Longer arithmetic expressions are built up by use of the familiar operators such as `+` and `*`.

AMPL's logical expressions evaluate to true or false. They are most often created through the use of standard comparison operators like `=` and `<=`. AMPL also provides a set membership operator, `in`, which produces a true result if and only if its left operand in a member of its right operand. Finally, logical expressions can be combined and extended by logic operators like `or` and `not`. Table 3–1 provides a summary of operators and operations, listed in order of decreasing precedence.

Operator	Operation
`^`	exponentiation
`+  -  not`	unary plus, minus, logical negation
`*  /  mod`	multiplication, division, remainder
`sum`, etc.	iterated addition, etc. (see Table 3–2)
`+  -  less`	addition, subtraction, non-negative subtraction
`in`	set membership
`< <= = >= > <>`	comparison
`and`	logical conjunction
`or`	logical disjunction
`if...then...else...`	conditional evaluation

Table 3–1. *Arithmetic and logical operators, in order of decreasing precedence.*

3. Numerical Values

An effective large-scale modeling language must be able to describe "vectors" and "matrices" and similar collections of numerical values indexed over sets. As the preceding sections have explained, only a symbolic description of these values need appear in the model, while the actual data can be given separately in some convenient way (such as in the format described by Section 5).

In AMPL a single symbolic numerical value is called a *parameter*. Since parameters are most often indexed over sets, we will loosely refer to an indexed collection of parameters as "a parameter" when the meaning is clear. To begin this section, §3.1 describes AMPL's rules for declaring indexed parameters and for specifying simple conditions on them.

Representations of numerical values are combined by arithmetic and logical operations to produce the expressions in a model's objective and constraints. Along with the familiar unary and binary operators, conventional algebraic notation provides *iterated* operators such as $\sum_{i=1}^{m}$ for addition and $\prod_{j=1}^{n}$ for multiplication. AMPL's versions of these operators are surveyed in §3.2, with particular attention to the use of AMPL sets in indexing the iterated summations that are essential to linear programming. We also introduce a conditional (if-then-else) construction that is frequently useful *within* the arithmetic expressions of complex models.

An AMPL numerical expression may be used almost anywhere in a model that a number is appropriate. However, an algebraic model is easiest to read and to verify if the expressions in its objective and constraints are kept fairly simple. Thus AMPL provides for using arithmetic expressions to define new parameters in terms of previously-defined parameters and sets, as explained in §3.3.

3.1 Parameters

An AMPL parameter declaration describes certain data required by a model, and indicates how the model will refer to those data in symbolic expressions. Syntactically, a parameter declaration consists of the keyword `param` followed by an identifier and by optional phrases for indexing, checking and other purposes.

The formation of indexing expressions and the declaration of indexed parameters have been introduced in Sections 1 and 2. A straightforward example is

```
param io {commod,proc};
```

which defines the input-output coefficients for the EGYPT model. Given members `c` and `p` of the sets `commod` and `proc`, respectively, the corresponding parameter value is denoted `io[c,p]`.

Unlike `io`, most parameters cannot meaningfully assume arbitrary positive and negative values. Thus typical declarations contain a qualifying expression, as in the following examples from PROD:

```
param iinv {prd} >= 0;
param cri {prd} > 0;
param life > 0 integer;
```

All values for `iinv` must be nonnegative, and all for `cri` must be strictly positive; `life` must be a positive integer. These restrictions are essential to the validity of the model, and are enforced by the translator when it gets to the point of inspecting the data values. Any violation is treated as an error.

Although simple restrictions like nonnegativity and integrality are most common, others are sometimes appropriate. In PROD, the last period of the planning horizon must be after be represented instead as sets of ordered pairs within `plant cross proc`, in which case the process possibilities would be given by `set p_pos := p_cap diff p_except` and the process level variables would be declared as `var Z {p_pos} >= 0`. The unit-plant pairs for which there is positive capacity could likewise be declared by

```
set m_pos := {u in unit, pl in plant: icap[u,pl] > 0};
```

All of the other sets of sets in EGYPT can be similarly converted.

Conversely, the set of pairs `rt` in the DIST model could be represented instead as a set of sets:

```
set rt {d in dctr} := {w in whse: d <> w and sc[d,w] < huge};
```

For each distribution center `d`, this would define `rt[d]` as the subset of warehouses to which the center can ship.

Almost any model that uses sets of sets can be made to use ordered pairs instead, and vice versa. The choice depends on which notation the modeler finds more appropriate and convenient. The examples above suggest that ordered pairs sometimes offer more concise but less descriptive expressions. The most important differences, however, are likely to arise in the formulation of the constraints, discussed further in Section 4.

A variable `X` representing the number of cars in each train is declared by indexing it over the set of quadruples:

```
var X {schedule} >= 0;
```

In the data for this model (Appendix D) `cities` has 4 members and `last` is 48. There are $4 \times 48 \times 4 \times 48 = 36864$ potential quadruples, yet `schedule` has only about 200 members. The ordered quadruples are essential to a clear and efficient model description.

2.4 Sets of sets

Just as parameters, variables and constraints can be indexed over sets, it sometimes makes sense to define a collection of sets indexed over some other set. The EGYPT model, for example, postulates a set of fertilizer *plants* and a set of production *processes.* At each plant, however, a certain subset of processes is prohibited. The collection of all these prohibited subsets is a "set of sets" indexed over the set of plants.

In AMPL, the collection of subsets of prohibited processes is declared straightforwardly:

```
set plant;
set proc;
set p_except {plant} within proc;
```

One set `p_except[pl]` is here defined for each member `pl` in `plant`. All of these sets must be subsets of `proc`; their actual membership is specified along with the rest of the set data, as seen in the Appendix B listing.

AMPL's set operators can be used to define new collections of sets from ones that are similarly indexed. For instance, a second set of sets, `p_cap`, represents the subset of processes for which capacity is available at each plant. Then the subset of all *possible processes* at each plant is declared as

```
set p_pos {pl in plant} := p_cap[pl] diff p_except[pl];
```

For each `pl` in `plant`, this declaration defines a separate set `p_pos[pl]` equal to the "difference" of `p_cap[pl]` and `p_except[pl]`. Thus, for each plant, the subset of possible processes consists of the ones that have capacity available and that are not prohibited.

AMPL's indexing expressions can also be used to define indexed collections of sets. Consider the set `unit` of production units that may be found in fertilizer plants; for each combination of unit `u` and plant `pl`, there is an initial capacity `icap[u,pl]`. It is desirable to define `m_pos[pl]` as the subset of units that have positive initial capacity at plant `pl`:

```
set m_pos {pl in plant} := {u in unit: icap[u,pl] > 0};
```

More complicated expressions of this kind define several other convenient sets of sets in the EGYPT model.

Sets of sets are typically employed in compound indexing expressions for parameters, variables and constraints. As an example, the variables `Z[pl,pr]` represent the levels of processes `pr` at plants `pl`. They are declared by

```
var Z {pl in plant, p_pos[pl]} >= 0;
```

The simpler indexing expression `{plant,proc}` might have been used, but then a variable would have been defined for every combination of plant and process. The above declaration creates a variable only for every combination of a plant and a possible process at that plant.

Sets of sets have much in common with ordered pairs. Both allow a model to specify a sparse subset of the cross product of two sets. The sets `p_cap` and `p_except` above could of the right operand. Thus the above statements say that the data for `rt` must consist of ordered pairs whose first components come from `dctr` and whose second components come from `whse`.

An indexing expression for `rt` might be either of

```
{rt}
{(d,w) in rt}
```

depending on whether the dummy indices are needed. Indexing expressions may also combine `rt` with other sets, as in

```
var Ship {prd,rt} >= 0;
```

In the objective and constraints of DIST, references to these product shipment variables have the form `Ship[p,d,w]`, where `p` is a member of `prd` and `(d,w)` is a member of `rt`.

If the shipment routes are to be read directly as data, then a human modeler must compile the list of route pairs and enter them into the data file. For the DIST application, however, the permitted routes are always a certain function of the shipping costs and other numerical data. Thus a more convenient and more reliable AMPL model defines `rt` in terms of logical conditions on the parameters, by use of an indexing expression. First of all, the shipping cost rates are specified by

```
param sc {dctr,whse} >= 0;
param huge > 0;
```

Only the routes from `d` to `w` for which `sc[d,w]` is less than `huge` are to be permitted. Thus `rt` can be defined by

```
set rt := {d in dctr, w in whse: sc[d,w] < huge};
```

Since each distribution center is also a warehouse, however, this set contains a route from each distribution center to itself. Such routes have a shipping rate of zero in the data, so they need to be ruled out separately:

```
set rt := {d in dctr, w in whse: d <> w and sc[d,w] < huge};
```

In the full DIST example, a route may also be disallowed if it runs to a warehouse where there is no demand (unless the warehouse is also a distribution center), or if it is subject to a "minimum size restriction" and the total demand at the warehouse fills less than a certain number of shipping pallets. The entire definition is

```
set rt := {d in dctr, w in whse:
     d <> w  and  sc[d,w] < huge  and
     (w in dctr or sum {p in prd} dem[p,w] > 0)  and
     not (msr[d,w] and sum {p in prd} 1000*dem[p,w]/cpp[p] < dsr[d]) };
```

The lengthy logical condition in the indexing expression is built up mainly from arithmetic comparisons connected by `and`, `or` and `not`. It also uses the logical expression `w in dctr`, which is true if and only if `w` is a member of the set `dctr`.

Sets of longer ordered lists are handled in an analogous fashion. As an example, in TRAIN each line of the railroad schedule is an ordered quadruple: city of departure, time of departure, city of arrival, time of arrival. The AMPL declarations are

```
set cities;
param last > 0 integer;
set times := 1..last;
set schedule dimen 4;
```

The sets are evaluated from left to right. Thus, in the last example above, `p` runs through all members of `prd` and `v` runs from `1` to `life-1`; for each such combination, `a` runs from `v+1` to `life`. This is a natural way to define a "triangular" array of index values. AMPL also permits "square" arrays, as in EGYPT:

```
{p1 in plant, p2 in plant}
```

This expression specifies indexing over all possible pairs of members from `plant`, in the definition of a table of interplant distances.

Realistic models often require more complicated indexing, in which the membership of the indexing set is somehow restricted. AMPL provides for this possibility by allowing a logical condition to be specified after the set or sets in the indexing expression. For example, DIST indexes a collection of constraints over all product-factory pairs for which the production cost is specified as zero:

```
{p in prd, f in fact: rpc[p,f] = 0}
```

The qualification in an indexing expression may also compare the dummy indices directly. Thus we could write

```
{p1 in plant, p2 in plant: p1 <> p2}
```

to index over all possible pairs of different members from `plant`.

2.3 Compound sets

It is often most natural to think of a set as comprising not individual items, but ordered pairs of items, or possibly ordered triples, quadruples or longer lists. As an example, in DIST the allowed shipment routes comprise a set of ordered pairs (d, w) such that d is a member of the set of distribution centers and w is a member of the set of warehouses. Variables representing shipment amounts are indexed over these pairs.

If every distribution center could ship to every warehouse, then AMPL could handle (d, w) pairs by means of the expressions introduced in §2.2 above. Indeed, the indexing expression `{dctr,whse}` or `{d in dctr, w in whse}` specifies precisely the set of all ordered pairs of centers and warehouses. In the application that gives rise to the DIST model, however, shipments from a distribution center are permitted only to certain related warehouses. Thus the set of shipment routes is a "sparse" subset of the center-warehouse pairs, and the variables representing shipments are defined only for the pairs in this subset. Such a situation is common in distribution and network models. To handle it naturally, a modeling language must be able to index over "all (d, w) in the set of routes" and similar kinds of sets.

In AMPL the set of shipment routes could be declared most simply as follows:

```
set rt dimen 2;
```

This would say that `rt` is a set whose members must be "2-dimensional": ordered pairs of objects. The routes in DIST, however, cannot be just any pairs of objects; each member must be a distribution center paired with a warehouse. A more appropriate AMPL declaration has the following form:

```
set whse;
set dctr within whse;
set rt within (dctr cross whse);
```

The set operator `cross` is a cross, or Cartesian, product; applied to two simple sets, its result is the set of all ordered pairs comprising a member of the left operand followed by a member

Simple sets may also be defined in terms of other sets or parameters, rather than directly from the data. AMPL provides operators for the union, intersection and difference of sets, as used in EGYPT to build the set of commodities from the sets of final, intermediate and raw materials:

```
set commod := c_final union c_inter union c_raw;
```

There is also the `..` operator that constructs sets of consecutive integers between two limits. In PROD, the name `time` is given to the set of integers beginning with the value of parameter `first` and ending with the value of parameter `last`:

```
param first > 0 integer;
param last > first integer;
set time := first .. last;
```

Set expressions built from these operators can also be used anywhere else that a set value is required in an AMPL model. For example, a declaration such as

```
set c_prod within c_final union c_inter;
```

would declare a set `c_prod` whose members must lie within either `c_final` or `c_inter`.

2.2 Indexing expressions

A more general syntax is required to specify the sets over which parameters, variables and constraints are indexed. The same syntax can then be used to describe the sets over which summations and other operations are iterated, as explained in Section 3 below.

In algebraic notation, indexing is indicated informally by a phrase such as "for all $i \in \mathcal{P}$" or "for $t = 1, \ldots, T$". AMPL formalizes these phrases as *indexing expressions* delimited by braces. As seen in PROD, the simplest kind of indexing expression is just the name of a set, optionally preceded by a named dummy index:

```
{prd}
{t in time}
```

The set in an indexing expression may also be specified by use of set operators:

```
{first-1..last}
{a in 1..life}
```

In keeping with the conventions of algebraic notation, we do not require that the name of a dummy index bear any particular relation to the name of any set. Sometimes it is convenient to use the same index name with different sets (as `t` is used with several sets representing times in PROD) or different index names with the same set (as `pl`, `p1`, `p2` are used with `plant` in EGYPT). The dummy index may be dropped entirely in parameter and variable declarations where it is not needed.

Large models cannot be adequately described by indexing over the members of individual sets. Many parameters, variables and constraints are most naturally indexed over all combinations of members from two or more sets. AMPL denotes such indexing by listing the sets sequentially within the braces of an indexing expression, as in the following examples from PROD:

```
{prd,time}
{prd,first..last+1}
{prd,time,1..life}
{p in prd, t in time}
{p in prd, v in 1..life-1, a in v+1..life}
```

2. Sets and Indexing

Index sets are the fundamental building blocks of any large linear programming model. Most of a model's components are indexed over some combination of these sets; for models of practical interest, moreover, the sets are seldom as easy to describe as $\mathcal{P}$, $\mathcal{R}$ and $1, \ldots, T$ in Figure 1–1. Thus the design of a modeling language cannot afford to place too many restrictions on the variety of set and indexing expressions. Any restriction is likely to reduce the range of models that can be conveniently represented.

In light of these observations, we have sought to offer a particularly broad variety of set types in our modeling language. AMPL provides for "sparse" subsets, for sets of ordered pairs, triples and longer "tuples", and for indexed collections of sets. Sets can be defined by applying operations like union and intersection to other sets, or by specifying arbitrary logical conditions for membership. Parameters, variables and constraints can be indexed over any set; common iterated operations, such as summation, are indexed over sets in the same way.

We begin this section by introducing the simpler kinds of AMPL sets in §2.1, and the concept of an indexing expression in §2.2. We then take a longer look at sets of ordered pairs and other compound sets in §2.3, and at indexed collections of sets in §2.4.

Most examples in this and subsequent sections are taken from four extensive AMPL models that are collected in the appendices. PROD and DIST are adapted from a multiperiod production model and a multicommodity distribution model that were developed for a large manufacturer. EGYPT is a static model of the Egyptian fertilizer industry (Choksi, Meeraus and Stoutjesdijk 1980). TRAIN is an adaptation of a model of railroad passenger-car requirements (Fourer, Gertler and Simkowitz 1977, 1978).

2.1 Simple sets

An unordered set of arbitrary objects can be defined by giving its name in an AMPL set declaration:

```
set prd;
set center;
set whse;
```

AMPL also provides for the declaration of arbitrary subsets. As an example, DIST is formulated on the assumption that distribution centers are located at a subset of the warehouses, and that factories are located at a subset of the distribution centers:

```
set dctr within whse;
set fact within dctr;
```

These subset declarations serve both as an aid to anyone reading the model, and as a check on the data. If the data include, say, a member of set `fact` that is not a member of set `dctr`, then the translator will reject the data for `fact` and report an error.

The actual set declarations in DIST are a little longer than those above, because they include a quoted *alias* following the set identifier:

```
set whse 'warehouses';
set dctr 'distribution centers' within whse;
set fact 'factories' within dctr;
```

An alias can be regarded either as a brief comment, or as a long identifier to be passed along by the AMPL translator for eventual use in reports. A similar syntax is used in appending aliases to other identifiers in an AMPL model. To save space, however, we will henceforth omit aliases in quotations from our examples.

implementation of an AMPL translator may accept data in whatever formats its creators deem appropriate. As a practical matter, however, we wish to have a standard data format that all versions of the translator will accept. Figure 1–4 shows a small data set for the sample LP in our standard format; it is a largely self-explanatory transcription of Figure 1–2a. Section 5 considers the data format in more detail.

Once the data values have been read successfully, the members of all sets and the values of all parameters are known. The AMPL translator can then identify the variables that will appear in the resulting linear program, determine the coefficients and constants in the objective and constraints, and write the output suitable for an algorithm. Section 6 describes our implementation of an AMPL translator.

In writing `1..T+1` above, we use an arithmetic expression to help define a set. For the most part, expressions may be used in AMPL anywhere that a numeric value is needed.

1.5 Objective

An objective function can be any linear expression in the parameters and variables. The AMPL representation of the objective in Figure 1–3 is transcribed from the algebraic objective expression in Figure 1–1.

AMPL representations of indexed sums appear for the first time in our example's objective. The sum of the estimated profits for period `t` is typical:

```
sum {j in prd} profit[j,t] * Make[j,t]
```

The identifier `j` is a dummy index that has exactly the same purpose and meaning as its counterpart in the algebraic notation. It is defined for the scope of the `sum`, which extends to the end of the following term.

1.6 Constraints

A constraint may be any linear equality or inequality in the parameters and variables. Thus a model's constraints use all the same kinds of expressions as its objective. Whereas the objective in Figure 1–1 is a single expression, however, the constraints come in collections indexed over sets. There is one production-limit constraint, for example, in each period.

The AMPL representation for a collection of constraints must specify two things: the set over which the constraints are indexed, and the expression for the constraints. Thus the production limits look like this:

```
subject to limit {t in 1..T}: sum {j in prd} Make[j,t] <= max_prd;
```

Following the keywords `subject to` and the identifier `limit`, the expression in braces gives `1..T` as the indexing set. The identifier `t` is another dummy index, playing the same role as its counterpart in the algebraic form; its scope is the entire inequality following the colon. Thus the declaration specifies a different inequality constraint for each choice of a member `t` from `1..T`.

AMPL constraint expressions need not have all the variables on the left of the relational operator, or all the constant terms on the right. The `balance` constraints are an example of a more general form:

```
subject to balance {i in raw, t in 1..T}:
     Store[i,t+1] = Store[i,t] - sum {j in prd} units[i,j] * Make[j,t];
```

The reference to `Store[i,t+1]` shows how a member of the set `1..T` is conveniently used in an arithmetic expression.

The `start` constraints can be regarded as simple upper bounds on the `Store` variables for period 1. Most LP optimizers work more efficiently by handling such bounds as implicit restrictions on the variables, rather than as an explicit part of the constraint matrix. Nevertheless, bounds may be specified in AMPL just like any other algebraic constraint; detection and treatment of bounds are left for the computer system to carry out as part of its processing of the model.

Most of Section 4 is devoted to issues that arise in representing AMPL constraints.

1.7 Data

Once the AMPL translator has read and processed the contents of Figure 1–3, it is ready to read the data. Strictly speaking, the rules for the data are not a part of AMPL; each comments when declarations are presented as examples in the following text.

Another kind of set is a sequence of integers. In the sample model, the sequence of all periods from 1 to T is such a set. It is represented in AMPL by the expression `1..T`, where `T` is a parameter whose value is the number of periods. The members of a sequence set are obviously ordered, and may appear in arithmetic expressions.

Section 2 considers the requirements of sets and indexing in detail.

1.3 Parameters

A parameter is any numerical value pertinent to a model. The simplest kind of parameter is a single, independent value, such as the number of periods or the maximum total production in our example.

Most AMPL statements that declare parameters also specify certain restrictions on them. For instance, the number of periods is declared by

```
param T > 0 integer;
```

which says that the value of `T` must be a positive integer. The integer restriction is understood implicitly by the human reader of the algebraic model, but it must be made explicit for the computer system that translates AMPL. Data inconsistent with this restriction are rejected by the translator.

Most of a model's parameters are not individual values, but are instead grouped into arrays indexed over sets. The initial stocks are a typical case; there is one stock level, b_i, that must be specified for each raw material i in the set $\mathcal{R}$. AMPL expresses this relationship by

```
param init_stock {raw} >= 0;
```

which declares one nonnegative parameter corresponding to each member of the set `raw`. Later, in describing the constraints, an index `i` is defined and the parameter corresponding to member `i` of `raw` is denoted `init_stock[i]`. (AMPL expressions in braces always represent entire sets, while expressions in brackets represent specific members of sets.)

The parameters a_{ij} and c_{jt} are arrays indexed over two sets. The AMPL declaration corresponding to c_{jt} is

```
param profit {prd,1..T};
```

This defines one parameter for each combination of a member from `prd` and a member from `1..T`. Naturally, the parameter corresponding to a particular `j` in `prd` and `t` in `1..T` is later denoted `profit[j,t]`. (AMPL respects the case of letters, so the index `t` is not confused with the set `T`.)

Section 3 takes a closer look at the handling of parameters and expressions.

1.4 Variables

A linear program's variables are declared much like its parameters. The only substantial difference is that the values of the variables are to be determined through optimization, whereas the values of the parameters are data given in advance.

A typical declaration of variables is the one for raw material in storage:

```
var Store {raw,1..T+1} >= 0;
```

One nonnegative variable is here defined for each combination of a member from `raw` and a member from `1..T+1`. By analogy with the notation for a parameter, the variable corresponding to a particular `i` in `raw` and `t` in `1..T+1` is denoted `Store[i,t]`.

```
data;
set prd := nuts bolts washers;
set raw := iron nickel;
param T := 4;
param max_prd := 123.7;

param units :     nuts    bolts   washers  :=
        iron      .79      .83      .92
        nickel    .21      .17      .08    ;

param profit :      1       2       3       4    :=
        nuts       1.73    1.8     1.6     2.2
        bolts      1.82    1.9     1.7     2.5
        washers    1.05    1.1      .95    1.33  ;

param :          init_stock    cost    value  :=
        iron       35.8        .03      .02
        nickel     7.32        .025    -.01   ;

end;
```

Figure 1–4. *Data from Figure 1–2a in standard AMPL format.*

When AMPL is used in this way, the major work left to people is the formulation of the model and the collection of the data. In addition, if a model is initially formulated by use of the traditional algebraic notation, then a person must convert it to AMPL statements before the translator can be applied. AMPL is designed, however, so that such a conversion is more *transcription* than translation. Almost every expression in Figure 1–3, for example, can be determined in a straightforward way from some analogous expression in Figure 1–1.

Because AMPL is read by a computer, it does differ from algebraic notation in several obvious ways. The syntax is more regular; every declaration of parameters begins with `param`, for instance, and ends with a semicolon. Traditional mathematical notations like $a_{ij}x_{jt}$, $i \in \mathcal{R}$ and $\sum_{t=1}^{T}$ are replaced by unambiguous expressions that use only ASCII characters. The AMPL syntax does permit multicharacter names, however, in place of the single letters that are more appropriate to algebraic expressions.

The five major parts of an algebraic model—sets, parameters, variables, objectives and constraints—are also the five kinds of components in an AMPL model. The remainder of this section briefly introduces each component and its AMPL representation; at the end, we also remark on the data format.

1.2 Sets

A set can be any unordered collection of objects pertinent to a model. Two unordered sets in our example are the set $\mathcal{P}$ of final products and the set $\mathcal{R}$ of raw materials. They are declared by the AMPL statements

```
set prd;      # products
set raw;      # raw materials
```

The membership of these sets is specified as part of the LP data (in Figures 1–2a and 1–4).

Comments accompanying the declarations of `prd` and `raw` begin with the symbol `#` and extend to the end of the line. Models are almost always easier to understand if they contain appropriate comments, such as those throughout Figure 1–3. For brevity, however, we omit

```
###  SETS  ###

set prd;                      # products
set raw;                      # raw materials

###  PARAMETERS  ###

param T > 0 integer;          # number of production periods

param max_prd > 0;            # maximum units of production per period

param units {raw,prd} >= 0;   # units[i,j] is the quantity of raw material i
                              # needed to manufacture one unit of product j

param init_stock {raw} >= 0;  # init_stock[i] is the maximum initial stock
                              # of raw material i

param profit {prd,1..T};      # profit[j,t] is the estimated value (if >= 0)
                              # or disposal cost (if <= 0) of
                              # a unit of product j in period t

param cost {raw} >= 0;        # cost[i] is the storage cost
                              # per unit per period of raw material i

param value {raw};            # value[i] is the estimated residual value
                              # (if >= 0) or disposal cost (if <= 0)
                              # of raw material i after the last period

###  VARIABLES  ###

var Make {prd,1..T} >= 0;     # Make[j,t] is the number of units of product j
                              # manufactured in period t

var Store {raw,1..T+1} >= 0;  # Store[i,t] is the number of units of raw material i
                              # in storage at the beginning of period t

###  OBJECTIVE  ###

maximize total_profit:

      sum {t in 1..T} ( sum {j in prd} profit[j,t] * Make[j,t] -
                       sum {i in raw} cost[i] * Store[i,t] )

    + sum {i in raw} value[i] * Store[i,T+1];

                   # Total over all periods of estimated profit,
                   # minus total over all periods of storage cost,
                   # plus value of remaining raw materials after last period

###  CONSTRAINTS  ###

subject to limit {t in 1..T}:  sum {j in prd} Make[j,t] <= max_prd;

                   # Total production in each period must not exceed maximum

subject to start {i in raw}:  Store[i,1] <= init_stock[i];

                   # Units of each raw material in storage at beginning
                   # of period 1 must not exceed initial stock

subject to balance {i in raw, t in 1..T}:
     Store[i,t+1] = Store[i,t] - sum {j in prd} units[i,j] * Make[j,t];

                   # Units of each raw material in storage
                   # at the beginning of any period t+1 must equal
                   # units in storage at the beginning of period t,
                   # less units used for production in period t
```

Figure 1–3. *The model of Figure 1–1 transcribed into AMPL.*

$\mathcal{P} = \{\textit{nuts}, \textit{bolts}, \textit{washers}\}$
$\mathcal{R} = \{\textit{iron}, \textit{nickel}\}$
$T = 4,\ M = 123.7$

a_{ij}	*nuts*	*bolts*	*washers*
iron	.79	.83	.92
nickel	.21	.17	.08

c_{jt}	1	2	3	4
nuts	1.73	1.8	1.6	2.2
bolts	1.82	1.9	1.7	2.5
washers	1.05	1.1	.95	1.33

i	b_i	d_i	f_i
iron	35.8	.03	.02
nickel	7.32	.025	−.01

Figure 1–2a. *Data for Figure 1–1 with two raw materials, three products, four periods.*

$$\begin{array}{ll}
\text{Maximize} & 1.73x_{11} + 1.82x_{21} + 1.05x_{31} - .03s_{11} - .025s_{21} \\
& +1.8x_{12} + 1.9x_{22} + 1.1x_{32} - .03s_{12} - .025s_{22} \\
& +1.6x_{13} + 1.7x_{23} + .95x_{33} - .03s_{13} - .025s_{23} \\
& +2.2x_{14} + 2.5x_{24} + 1.33x_{34} - .03s_{14} - .025s_{24} \\
& +.02s_{15} - .01s_{25}, \\
& \\
\text{subject to} & x_{11} + x_{21} + x_{31} \le 123.7, \\
& x_{12} + x_{22} + x_{32} \le 123.7, \\
& x_{13} + x_{23} + x_{33} \le 123.7, \\
& x_{14} + x_{24} + x_{34} \le 123.7, \\
& \\
& s_{11} \le 35.8, \\
& s_{21} \le 7.32, \\
& \\
& s_{12} = s_{11} - .79x_{11} - .83x_{21} - .92x_{31}, \\
& s_{13} = s_{12} - .79x_{12} - .83x_{22} - .92x_{32}, \\
& s_{14} = s_{13} - .79x_{13} - .83x_{23} - .92x_{33}, \\
& s_{15} = s_{14} - .79x_{14} - .83x_{24} - .92x_{34}, \\
& s_{22} = s_{21} - .21x_{11} - .17x_{21} - .08x_{31}, \\
& s_{23} = s_{22} - .21x_{12} - .17x_{22} - .08x_{32}, \\
& s_{24} = s_{23} - .21x_{13} - .17x_{23} - .08x_{33}, \\
& s_{25} = s_{24} - .21x_{14} - .17x_{24} - .08x_{34}.
\end{array}$$

Figure 1–2b. *Linear program defined by Figures 1–1 and 1–2a.*

Given	$\mathcal{P}$	a set of products,
	$\mathcal{R}$	a set of raw materials,
	$T > 0$	the number of production periods,
	$M > 0$	maximum total production per period;
and	$a_{ij} \geq 0$	$i \in \mathcal{R}$, $j \in \mathcal{P}$: units of raw material i needed to manufacture one unit of product j;
	$b_i \geq 0$	$i \in \mathcal{R}$: maximum initial stock of raw material i;
	c_{jt}	$j \in \mathcal{P}$, $t = 1, \ldots, T$: estimated profit (if ≥ 0) or disposal cost (if ≤ 0) of product j in period t;
	$d_i \geq 0$	$i \in \mathcal{R}$: storage cost per period per unit of raw material i;
	f_i	$i \in \mathcal{R}$: estimated residual value (if ≥ 0) or disposal cost (if ≤ 0) of raw material i after the last period.
Define	$x_{jt} \geq 0$	$j \in \mathcal{P}$, $t = 1, \ldots, T$: units of product j manufactured in period t;
	$s_{it} \geq 0$	$i \in \mathcal{R}$, $t = 1, \ldots, T+1$: units of raw material i in storage at the beginning of period t.
Maximize	$\sum_{t=1}^{T} \left(\sum_{j \in \mathcal{P}} c_{jt} x_{jt} - \sum_{i \in \mathcal{R}} d_i s_{it} \right) + \sum_{i \in \mathcal{R}} f_i s_{i,T+1}$:	total over all periods of estimated profit less storage cost, plus value of remaining raw materials after the last period;
subject to	$\sum_{j \in \mathcal{P}} x_{jt} \leq M$,	$t = 1, \ldots, T$: total production in period t must not exceed the specified maximum;
	$s_{i1} \leq b_i$,	$i \in \mathcal{R}$: units of raw material i on hand at the beginning of period 1 must not exceed the specified maximum;
	$s_{i,t+1} = s_{it} - \sum_{j \in \mathcal{P}} a_{ij} x_{jt}$,	$i \in \mathcal{R}$, $t = 1, \ldots, T$: units of raw material i on hand at the beginning of period $t+1$ must equal units on hand at the beginning of period t less units used for production in period t.

Figure 1–1. *Algebraic formulation of a maximum-revenue production problem.*

interpret a model and its associated data. The translator's output is a representation of a mathematical program that is suitable as input for most algorithms. Timings for a variety of realistic problems, ranging to over a thousand constraints and ten thousand variables, suggest that the computing cost of translation is quite reasonable in comparison to the cost of optimization.

We intend AMPL to be able to express many kinds of mathematical programs. In the interest of keeping this paper to a reasonable length, however, we confine the discussion and examples to linear programming. Section 7 compares AMPL to the languages used by various linear programming systems, but also indicates how AMPL is being extended to other kinds of models and how it may be integrated with other modeling software. Appendices list the four AMPL linear programs from which the illustrations in the text are extracted.

1.1 An introductory example

Figure 1–1 displays the algebraic formulation of a simple linear programming model, as it might appear in a report or paper. The formulation begins with a description of the index *sets* and numerical *parameters* that the model requires. Next, the decision *variables* are defined. Finally the *objective* and *constraints* are specified as expressions in the sets, parameters and variables.

The algebraic formulation in Figure 1–1 does not define any particular optimization problem. The purpose of this formulation is to specify a general class of problems that share a certain structure and purpose: production over time to maximize revenues. If we want to define a specific problem, we must supplement this formulation with values for all of the sets and parameters. Each different specification of set and parameter values will yield one different problem. To distinguish between a general formulation and a particular problem, we call the former a *model* and the latter a *linear program* or *LP*.

The distinction between general models and specific LPs is essential in dealing with very large linear optimization problems. As an illustration, Figure 1–2a presents a collection of data for a small instance of the preceding formulation: 2 raw materials, 3 final products and 4 periods. In such a small example the objective and constraints are easy to write out explicitly, as shown in Figure 1–2b. Suppose now that there are 10 raw materials, 30 final products and 20 periods. The model in Figure 1–1 is unchanged, and the data tables in Figure 1–2a still fit on perhaps two pages; but the linear program expands to 230 constraints in 810 variables, and its explicit listing (in the manner of Figure 1–2b) is too big to be usefully readable. If the periods are further increased to 40, the model is again unchanged and only one data table (the expected profits c_{jt}) doubles in size, even though the numbers of variables and constraints in the linear program are both roughly doubled.

We will use the term *model translator* to describe a computer system that reads a model in the compact algebraic form of Figure 1–1 along with data in the form of Figure 1–2a, and that writes out a linear program in the verbose explicit form of Figure 1–2b. For a practical implementation, the data input can be given a more machine-readable arrangement, and the explicit output can be written in a format more suitable to an efficient algorithm. The major challenge of translation, however, is to devise a language that can clearly represent the compact algebraic model, yet that can be read and interpreted by a computer system.

AMPL is such a language. The AMPL representation of Figure 1–1's model is shown in Figure 1–3, and is used throughout this introduction to illustrate the language's features.

An AMPL translator starts by reading, parsing and interpreting a model like the one in Figure 1–3. The translator then reads some representation of particular data; Figure 1–4 displays one suitable format for the data of Figure 1–2a. The model and data are then processed to determine the linear program that they represent, and the linear program is written out in some appropriate form.

1. Introduction

Practical large-scale mathematical programming involves more than just the minimization or maximization of an objective function subject to constraint equations and inequalities. Before any optimizing algorithm can be applied, some effort must be expended to formulate the underlying model and to generate the requisite computational data structures.

If algorithms could deal with optimization problems as people do, then the formulation and generation phases of modeling might be relatively easy. In reality, however, there are many differences between the form in which human modelers understand a problem and the form in which algorithms solve it. Reliable *translation* from the "modeler's form" to the "algorithm's form" is often a considerable expense.

In the traditional approach to translation, the work is divided between human and computer. First, a person who understands the modeler's form writes a computer program whose output will represent the required data structures. Then a computer compiles and executes the program to create the algorithm's form. This arrangement is often costly and error-prone; most seriously, the program must be debugged by a human modeler even though its output—the algorithm's form—is not meant for people to read.

In the important special case of linear programming, the largest part of the algorithm's form is the representation of the constraint coefficient matrix. Typically this is a very sparse matrix whose rows and columns number in the hundreds or thousands, and whose nonzero elements appear in intricate patterns. A computer program that produces a compact representation of the coefficients is called a matrix generator. Several programming languages have been designed specifically for writing matrix generators (Haverly Systems 1977, Creegan 1985) and standard languages like Fortran are also often used (Beale 1970).

Many of the difficulties of translation from modeler's form to algorithm's form can be circumvented by the use of a computer *modeling language* for mathematical programming. A modeling language is designed to express the modeler's form in a way that can serve as direct input to a computer system. Then the translation to the algorithm's form can be performed entirely by computer, without the intermediate stage of programming. The advantages of modeling languages over matrix generators have been analyzed in detail by Fourer (1983). Implementations such as GAMS (Bisschop and Meeraus 1982; Brooke, Kendrick and Meeraus 1988) and MGG (Simons 1987) were under way in the 1970's, and the pace of development has increased in recent years.

We describe in this paper the design and implementation of AMPL, a new modeling language for mathematical programming. Compared to previous languages, AMPL is notable for the generality of its syntax, and for the similarity of its expressions to the algebraic notation customarily used in the modeler's form. It offers a variety of types and operations for the definition of indexing sets, as well as a range of logical expressions. AMPL draws considerable inspiration from the XML prototype language (Fourer 1983), but incorporates many changes and extensions.

AMPL is introduced below through the example of a simple maximum-revenue production problem. Sections 2, 3 and 4 then use more complex examples to examine major aspects of the language design in detail. We have attempted to touch upon most of the language's fundamental features, while avoiding the lengthy specifics that would be appropriate to a user's guide or reference manual. Our emphasis is on aspects of the language that represent particularly important or difficult design decisions.

By itself, AMPL can only be employed to *specify* classes of mathematical programming models. For the language to be useful, it must be incorporated into a system that manages data, models and solutions. Thus Section 5 discusses a standard representation of data for an AMPL model, and Section 6 describes our implementation of a translator that can

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 133

AMPL: A Mathematical Programming Language

*Robert Fourer**
David M. Gay
Brian W. Kernighan

January 1987
Revised June 1989

*Department of Industrial Engineering and Management Sciences
Northwestern University, Evanston, Illinois 60208-3119

This appears in shortened form as "A Modeling Language for Mathematical Programming", *Management Science* **36** ♯5 (1990), pp. 519–554.