5

The Sets View of the World

In Normal form, each attribute/field of an entity/record should depend on
the entity key, the whole key, and nothing but the key, so help me Codd.
-anonymous

5.1 Introduction

The most powerful feature of LINGO is its ability to model large systems. The key concept that provides this power is the idea of a set of similar objects. When you are modeling situations in real life, there will typically be one or more groups of similar objects. Examples of such groups might be factories, products, time periods, customers, vehicles, employees, etc. LINGO allows you to group similar objects together into *sets*. Once the objects in your model are grouped into sets, you can make single statements in LINGO that apply to all members of a set.

A LINGO model of a large system will typically have three sections: 1) a SETS section, 2) a DATA section, and 3) a model equations section. The SETS section describes the data structures to be used for solving a certain class of problems. The DATA section provides the data to "populate" the data structures. The model equations section describes the relationships between the various pieces of data and our decisions.

5.1.1 Why Use Sets?

In most large models, you will need to express a group of several very similar calculations or constraints. LINGO's ability to handle sets allows you to express such formulae or constraints efficiently.

For example, preparing a warehouse-shipping model for 100 warehouses would be tedious if you had to write each constraint explicitly (e.g., "Warehouse 1 can ship no more than its present inventory, Warehouse 2 can ship no more than its present inventory, Warehouse 3 can ship no more than its present inventory…" and so on). You would prefer to make a single general statement of the form: "Each warehouse can ship no more than its present inventory".

5.1.2 What Are Sets?

A set is a group of similar objects. A set might be a list of products, trucks, employees, etc. Each member in the set may have one or more characteristics associated with it (e.g., weight, price/unit, or income). We call these characteristics *attributes*. All members of the same set have the same set of attribute types. Attribute values can be known in advance or unknowns for which LINGO solves. For example, each product in a set of products might have an attribute listing its price. Each truck in a set of trucks might have a hauling capacity attribute. In addition, each employee in a set of employees might have an attribute specifying salary as well as an attribute listing birth date.

5.1.3 Types of Sets

LINGO recognizes two kinds of sets: *primitive* and *derived*. A primitive set is a set composed only of objects that can't be further reduced.

A derived set is defined from one or more other sets using two operations: a) selection (of a subset), and/or b) Cartesian product (sometimes called a "cross" or a "join") of two or more other sets. The key concept is that a derived set *derives* its members from other pre-existing sets. For example, we might have the two primitive sets: *WAREHOUSE* and *CUSTOMER*. We might have the derived set called *SHIPLINK*, which consists of every possible combination of a warehouse and a customer. Although the set *SHIPLINK* is derived solely from primitive sets, it is also possible to build derived sets from other derived sets as well.

5.2 The SETS Section of a Model

In a set-based LINGO model, the first section in the model is usually the *SETS section*. A SETS section begins with the keyword `SETS:` (including the colon) and ends with the keyword `ENDSETS`. A model may have no SETS section, a single SETS section, or multiple SETS sections. A SETS section may appear almost anywhere in a model. The major restriction is that you must define a set and its attributes before they are referenced in the model's constraints.

5.2.1 Defining Primitive Sets

To define a primitive set in a SETS section, you specify:

- the *name* of the set, and
- any *attributes* the members of the set may have.

A primitive set definition has the following syntax[1]:

setname:[*attribute_list*];

The *setname* is a name you choose. It should be a descriptive name that is easy to remember. The set name must conform to standard LINGO naming conventions: begin with an alphabetic character, followed by up to 31 alphanumeric characters or the underscore (_). LINGO does not distinguish between upper and lowercase characters in names.

An example sets declaration is:

```
SETS:
  WAREHOUSE: CAPACITY;
ENDSETS
```

This means that we will be working with one or more warehouses. Each one of them has an attribute called *CAPACITY*. Set members may have zero or more *attributes* specified in the *attribute_list* of the set definition. An *attribute* is some property each member of the set possesses. Attribute names must follow standard naming conventions and be separated by commas.

[1]The use of Square brackets indicates that a particular item is optional. In this particular case, a primitive set's *member_list* and *attribute_list* are optional.

For illustration, suppose our warehouses had additional attributes related to their location and the number of loading docks. These additional attributes could be added to the attribute list of the set declaration as:

```
WAREHOUSE: CAPACITY, LOCATION, DOCKS;
```

5.2.2 Defining Derived Sets

To define a derived set, you specify:

- ♦ the *name* of the set,
- ♦ its *parent sets*,
- ♦ optionally, any *attributes* the set members may have.

A derived set definition has the following syntax:

set_name (parent*_set_list*) [*membership_filter*] [: *attribute_list*];

The *set_name* is a standard LINGO name you choose to name the set. The optional *membership_filter* may place a general condition on membership in the set.

The *parent_set_list* is a list of previously defined sets, separated by commas. LINGO constructs all the combinations of members from each of the parent sets to create the members of the derived set. As an example, consider the following SETS section:

```
SETS:
   PRODUCT ;
   MACHINE ;
   WEEK;
   ALLOWED( PRODUCT, MACHINE, WEEK): VOLUME;
ENDSETS
```

Sets *PRODUCT*, *MACHINE*, and *WEEK* are primitive sets, while *ALLOWED* is derived from parent sets *PRODUCT*, *MACHINE*, and *WEEK*. Unless specified otherwise, the set *ALLOWED* will have one member for every combination of *PRODUCT*, *MACHINE*, and *WEEK*. The attribute *VOLUME* might be used to specify how much of each product is produced on each machine in each week. A derived set that contains all possible combinations of members is referred to as being a *dense* set. When a set declaration includes a *membership_filter* or if the members of the derived set are given explicitly in a DATA section, then we say the set is *sparse*.

Summarizing, a derived set's members may be constructed by either:

- ♦ an explicit member list in a DATA section,
- ♦ a membership filter, or
- ♦ implicitly dense by saying nothing about the membership of the derived set.

Specification of an explicit membership list for a derived set in a DATA section will be illustrated in the next section of the text.

If you have a large, sparse set, explicitly listing all members can become cumbersome. Fortunately, in many sparse sets, the members all satisfy some condition that differentiates them from the non-members. If you can specify this condition, you can save yourself a lot of typing. This is exactly how the membership filter method works. Using the membership filter method of defining a derived set's *member_list* involves specifying a logical condition that each potential set member must satisfy for inclusion in the set. You can look at the logical condition as a *filter* that filters out potential members who don't measure up to some criteria.

As an example of a membership filter, suppose you have already defined a set called *TRUCKS* and each truck has an attribute called *CAPACITY*. You would like to derive a subset from *TRUCKS* that contains only those trucks capable of hauling big loads. You could use an explicit member list and explicitly enter each of the trucks that can carry heavy loads. However, why do all that work when you could use a membership filter as follows:

```
HEAVY_DUTY( TRUCKS) | CAPACITY( &1) #GT# 50000;
```

We have named the set *HEAVY_DUTY* and have derived it from the parent set *TRUCKS*. The vertical bar character (|) is used to mark the beginning of a membership filter. The membership filter allows only those trucks that have a hauling capacity (*CAPACITY*(&1)) greater than (#GT#) 50,000 into the *HEAVY_DUTY* set. The &1 symbol in the filter is known as a *set index placeholder*. When building a derived set that uses a membership filter, LINGO generates all the combinations of parent set members. Each combination is then "plugged" into the membership condition to see if it passes the test. The first parent set's value is plugged into &1, the second into &2, and so on. In this example, we have only one parent set (*TRUCKS*), so &2 would not have made sense. The symbol #GT# is a *logical operator* and means "greater than". Other logical operators recognized by LINGO include:

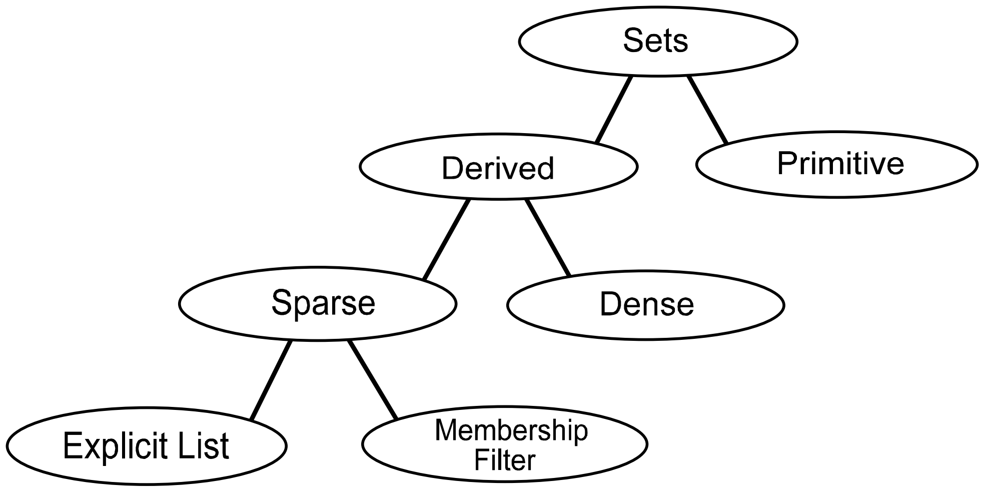
♦ #EQ#	equal
♦ #NE#	not equal
♦ #GE#	greater-than-or-equal-to
♦ #LT#	less than
♦ #LE#	less-than-or-equal-to

5.2.3 Summary

LINGO recognizes two types of sets - *primitive* and *derived*. Primitive sets are the fundamental objects in a model and can't be broken down into smaller components. Derived sets, on the other hand, are created from other component sets. These component sets are referred to as the parents of the derived set and may be either primitive or derived.

A derived set can be either *sparse* or *dense*. Dense sets contain all combinations of the parent set members (sometimes this is also referred to as the *Cartesian product* or *cross* of the parent sets). Sparse sets contain only a subset of the cross of the parent sets. These may be defined by two methods - *explicit listing* or *membership filter*. The explicit listing method involves listing the members of the sparse set in a DATA section. The membership filter method allows you to specify the sparse set members compactly using a logical condition, which all members must satisfy. The relationships amongst the various set types are illustrated in Figure 5.1 below.

Figure 5.1 Types of Sets



5.3 The DATA Section

A SETS section describes the structure of the data for a particular class of problems. A DATA section provides the data to create a specific instance of this class of problems. The DATA section allows you to isolate things that are likely to change from week to week. This is a useful practice in that it leads to easier model maintenance and makes a model easier to scale up or down in dimension.

We find it useful to partition a LINGO model of a large system into three distinct sections: a) the SETS section, b) the DATA section, and c) the model equations section. The developer of a model has to understand all three sections. However, if the developer has done a good job of partitioning the model into the aforementioned sections, the day-to-day user may only need to be familiar with the DATA section.

Similar to the SETS section, the DATA section begins with the keyword `DATA:` (including the colon) and ends with the keyword `ENDDATA`. In the DATA section, you place statements to initialize either the attributes of the member of a set you defined in a SETS section or even the set members. These expressions have the syntax:

attribute_list = *value_list*;

or

set_name = member_list;

The *attribute_list* contains the names of the attributes you want to initialize, optionally separated by commas. If there is more than one attribute name on the left-hand side of the statement, then all attributes must be associated with the same set. The *value_list* contains the values you want to assign to the attributes in the *attribute_list,* optionally separated by commas. Consider the following example:

```
SETS:
   SET1: X, Y;
ENDSETS
DATA:
  SET1 = M1, M2, M3;
     X =  1   2   3;
     Y =  4   5   6;
ENDDATA
```

We have two attributes, *X* and *Y*, defined on the set *SET1*. The three values of *X* are set to 1, 2, and 3, while *Y* is set to 4, 5, and 6. We could have also used the following compound data statement to the same end:

```
SETS:
   SET1: X, Y;
ENDSETS
DATA:
  SET1  X  Y =
   M1   1  4
   M2   2  5
   M3   3  6;
ENDDATA
```

Looking at this example, you might imagine *X* would be assigned the values 1, 4, and 2, since they are first in the values list, rather than the true values of 1, 2, and 3. When LINGO reads a data statement's value list, it assigns the first *n* values to the first position of each of the *n* attributes in the attribute list, the second *n* values to the second position of each of the *n* attributes, and so on. In other words, LINGO is expecting the input data in column form rather than row form.

The DATA section can also be used for specifying members of a derived set. The following illustrates both how to specify set membership in a DATA section and how to specify a sparse derived set. This example also specifies values for the *VOLUME* attribute, although that is not required:

```
SETS:
   PRODUCT ;
   MACHINE ;
   WEEK ;
   ALLOWED( PRODUCT, MACHINE, WEEK): VOLUME;
ENDSETS
DATA:
   PRODUCT = A  B;
   MACHINE = M  N;
   WEEK = 1..2;
   ALLOWED, VOLUME =
     A M 1   20.5
     A N 2   31.3
     B N 1   15.8;
ENDDATA
```

The *ALLOWED* set does not have the full complement of eight members. Instead, *ALLOWED* is just the three member sparse set:

```
(A,M,1), (A,N,2), and (B,N,1).
```

LINGO recognizes a number of standard sets. For example, if you declare in a DATA section:

PRODUCT = 1..5;

then the members of the *PRODUCT* set will in fact be 1, 2, 3, 4, and 5. If you declare:

PERIOD = Feb..May;

then the members of the *PERIOD* set will in fact be Feb, Mar, Apr, and May. Other examples of inferred sets include mon..sun and thing1..thing12.

If an attribute is not referenced in a DATA section, then it is by default a decision variable. LINGO may set such an attribute to whatever value is consistent with the statements in the model equations section.

This section gave you a brief introduction to the use of the DATA section. Data do not have to actually reside in the DATA section as shown in these examples. In fact, a DATA section can have OLE links to Excel, ODBC links to databases, and connections to other spreadsheet and text based data files. Examples are given later in this chapter.

Note, when LINGO constructs the derived set, it is the right-most parent set that is incremented the fastest.

5.4 Set Looping Functions

In the model equations section of a model, we state the relationships among various attributes. Any statements not in a SETS or DATA section are by default in the model equations section. The power of set based modeling comes from the ability to apply an operation to all members of a set using a single statement. The functions in LINGO that allow you to do this are called *set looping functions*. If your models do not make use of one or more set looping functions, you are missing out on the power of set based modeling and, even worse, you're probably working too hard!

Set looping functions allow you to iterate through all the members of a set to perform some operation. There are four set looping functions in LINGO. The names of the functions and their uses are:

Function	**Function's Use**
@FOR	Used to generate constraints over members of a set.
@SUM	Computes the sum of an expression over all members of a set.
@MIN	Computes the minimum of an expression over all members of a set.
@MAX	Computes the maximum of an expression over all members of a set.

The syntax for a set looping function is:

@loop_function (*setname* [(*set_index_list*)
[| *conditional_qualifier*]] : *expression_list*);

The *@loop_function* symbol corresponds to one of the four set looping functions listed in the table above. The *setname* is the name of the set over which you want to loop. The *set_index_list* is optional and is used to create a list of indices each of which correspond to one of the parent, primitive sets that form the set specified by *setname*. As LINGO loops through the members of the set *setname*, it will set the values of the indices in the *set_index_list* to correspond to the current member of the set *setname*. The *conditional_qualifier* is an optional filter and may be used to limit the scope of the set looping function. When LINGO is looping over each member of *setname*, it evaluates the *conditional_qualifier*. If the *conditional_qualifier* evaluates to true, then the *expression_list* of the *@loop_function* is performed for the set member. Otherwise, it is skipped. The *expression_list* is a list of expressions to be applied to each member of the set *setname*. When using the *@FOR* function, the expression list may contain multiple expressions that are separated by semicolons. These expressions will be added as constraints to the model. When using the remaining three set looping functions (*@SUM*, *@MAX*, and *@MIN*), the expression list must contain only one expression. If the *set_index_list* is omitted, all attributes referenced in the *expression_list* must be defined on the set *setname*.

5.4.1 @*SUM* Set Looping Function

In this example, we will construct several summation expressions using the *@SUM* function in order to illustrate the features of set looping functions in general and the *@SUM* function in particular.

Consider the model:

```
SETS:
   SET_A : X;
ENDSETS
DATA:
   SET_A = A1 A2 A3 A4 A5;
       X = 5  1  3  4  6;
ENDDATA
X_SUM = @SUM( SET_A( J): X( J));
```

LINGO evaluates the *@SUM* function by first initializing an internal accumulator to zero. LINGO then begins looping over the members in *SET_A*. You can think of *J* as a pronoun. The index variable *J* is first set to the first member of *SET_A* (i.e., *A1*) and *X*(*A1*) is then added to the accumulator. Then, *J* is set to the second element and this process continues until all values of *X* have been added to the accumulator. The value of the sum is then stored into the variable *X_SUM*.

Since all the attributes in our expression list (in this case, only *X* appears in the expression list) are defined on the index set (*SET_A*), we could have alternatively written our sum as:

```
X_SUM = @SUM( SET_A: X);
```

In this case, we have dropped the superfluous index set list and the index on *X*. When an expression uses this shorthand, we say the index list is *implied*. Implied index lists are not allowed when attributes in the expression list have different parent sets.

Next, suppose we want to sum the first three elements of the attribute *X*. We can use a conditional qualifier on the set index to accomplish this as follows:

```
X3_SUM = @SUM( SET_A( J) | J #LE# 3: X( J));
```

The `#LE#` symbol is called a *logical operator.* This operator compares the operand on the left (*J*) with the one on the right (3) and returns *true* if the left operand is less-than-or-equal-to the one on the right. Otherwise, it returns *false*. Therefore, this time, when LINGO computes the sum, it plugs the set index variable *J* into the conditional qualifier *J* `#LE#` 3. If the conditional qualifier evaluates to true, *X*(*J*) will be added to the sum. The end result is that LINGO sums up the first three terms in *X*, omitting the fourth and fifth terms, for a total sum of 9.

Before leaving this example, one subtle aspect to note in this last sum expression is the value that the set index *J* is returning. Note, we are comparing the set index variable to the quantity 3 in the conditional qualifier *J* `#LE#` 3. In order for this to be meaningful, *J* must represent a numeric value. Since a set index is used to loop over set members, one might imagine a set index is merely a placeholder for the current set member. In a sense, this is true. However, what set indexes *really* return is the *index* of the current set member in its parent primitive set. The index returned is *one-based*. In other words, the value 1 is returned when indexing the first set member, 2 when indexing the second, and so on. Given that set indices return a numeric value, they may be used in arithmetic expressions along with other variables in your model.

5.4.2 *@MIN* and *@MAX* Set Looping Functions

The *@MIN* and *@MAX* functions are used to find the minimum and maximum of an expression over members of a set. Again, consider the model:

```
SETS:
   SET_A : X;
ENDSETS
DATA:
  SET_A = A1 A2 A3 A4 A5;
      X = 5  1  3  4  6;
ENDDATA
```

To find the minimum and maximum values of *X*, all one need do is add the two expressions:

```
THE_MIN_OF_X = @MIN( SET_A( J): X( J));
THE_MAX_OF_X = @MAX( SET_A( J): X( J));
```

As with the *@SUM* example above, we can use an implied index list since the attributes are defined on the index set. Using implied indexing, we can recast our expressions as:

```
THE_MIN_OF_X = @MIN( SET_A: X);
THE_MAX_OF_X = @MAX( SET_A: X);
```

In either case, when we solve this model, LINGO returns the expected minimum and maximum values of *X*:

```
    Variable        Value
THE_MIN_OF_X     1.000000
THE_MAX_OF_X     6.000000
```

For illustration purposes, suppose we had just wanted to compute the minimum and maximum values of the first three elements of *X*. As with the *@SUM* example, all we need do is add the conditional qualifier *J* `#LE#` *3*. We then have:

```
THE_MIN_OF_X_3 = @MIN( SET_A( J) | J #LE# 3: X( J));
THE_MAX_OF_X_3 = @MAX( SET_A( J) | J #LE# 3: X( J));
```

with solution:

```
      Variable       Value
THE_MIN_OF_X_3    1.000000
THE_MAX_OF_X_3    5.000000
```

5.4.3 *@FOR* Set Looping Function

The *@FOR* function is used to generate constraints across members of a set. Whereas scalar based modeling languages require you to explicitly enter each constraint, the *@FOR* function allows you to enter a constraint just once and LINGO does the work of generating an occurrence of the constraint for each of the set members. As such, the *@FOR* statement provides the set based modeler with a very powerful tool.

To illustrate the use of *@FOR*, consider the following:

```
SETS:
   TRUCKS : HAUL;
ENDSETS
DATA:
   TRUCKS = MAC, PETERBILT, FORD, DODGE;
ENDDATA
```

Specifically, we have a primitive set of four trucks with a single attribute titled *HAUL*. If the attribute *HAUL* is used to denote the amount a truck hauls, then we can use the *@FOR* function to limit the amount hauled by each truck to 2,500 pounds with the following expression:

```
@FOR( TRUCKS( T): HAUL( T) <= 2500);
```

In this case, it might be instructive to view the constraints that LINGO generates from our expression. You can do this by using the *LINGO | Generate* command under Windows or by using the *GENERATE* command on other platforms. Running this command, we find that LINGO generates the following four constraints:

```
      HAUL( MAC) <=   2500;
HAUL( PETERBILT) <=   2500;
     HAUL( FORD) <=   2500;
    HAUL( DODGE) <=   2500;
```

As we anticipated, LINGO generated one constraint for each truck in the set to limit them to a load of 2,500 pounds.

Here is a model that uses an *@FOR* statement (listed in bold) to compute the reciprocal of any five numbers placed into the *GPM* attribute:

```
SETS:
   OBJECT: GPM, MPG;
ENDSETS
DATA:
   OBJECT =   A     B      C      D      E;
      GPM = .0303 .03571 .04545 .07142 .10;
ENDDATA
   @FOR( OBJECT( I):
      MPG( I) = 1 / GPM( I)
       );
```

Solving this model gives the following values for the reciprocals:

```
Variable           Value
 MPG( A)        33.00330
 MPG( B)        28.00336
 MPG( C)        22.00220
 MPG( D)        14.00168
 MPG( E)        10.00000
```

Since the reciprocal of zero is not defined, we could put a conditional qualifier on our *@FOR* statement that causes us to skip the reciprocal computation whenever a zero is encountered. The following *@FOR* statement accomplishes this:

```
@FOR( OBJECT( I) | GPM( I) #NE# 0:
   MPG( I) = 1 / GPM( I)
);
```

The conditional qualifier (listed in bold) tests to determine if the *GPM* is not equal (`#NE#`) to zero. If so, the computation proceeds.

This was just a brief introduction to the use of the *@FOR* statement. There will be many additional examples in the sections to follow.

5.4.4 Nested Set Looping Functions

The simple models shown in the previous section use *@FOR* to loop over a single set. In larger models, you may need to loop over a set within another set looping function. When one set looping function is used within the scope of another, we call it *nesting*. LINGO allows nesting.

The following is an example of an *@SUM* loop nested within an *@FOR*:

```
! The demand constraints;
   @FOR( VENDORS( J):
     @SUM( WAREHOUSES( I): VOLUME( I, J)) = DEMAND( J);
       );
```

Specifically, for each vendor, we sum up the shipments going from all the warehouses to that vendor and set the quantity equal to the vendor's demand.

@SUM, *@MAX*, and *@MIN* can be nested within any set looping function. *@FOR* functions, on the other hand, may only be nested within other *@FOR* functions.

5.5 Set Based Modeling Examples

Recall, four types of sets can be created in LINGO:

- primitive,
- dense derived,
- sparse derived - explicit list, and
- sparse derived - membership filter.

This section will help develop your talents for set based modeling by building and discussing four models. Each of these four models will introduce one of the set types listed above.

5.5.1 Primitive Set Example

The following staff scheduling model illustrates the use of a primitive set. This model may be found in the *SAMPLES* subdirectory off the main LINGO directory under the name *STAFFDEM.LNG*.

The Problem

Suppose you run the popular Pluto Dog's hot dog stand that is open seven days a week. You hire employees to work a five-day workweek with two consecutive days off. Each employee receives the same weekly salary. Some days of the week are busier than others and, based on past experience, you know how many workers are required on a given day of the week. In particular, your forecast calls for these staffing requirements:

Day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Staff Req'd	20	16	13	16	19	14	12

You need to determine how many employees to start on each day of the week in order to minimize the total number of required employees, while still meeting or exceeding staffing requirements each day of the week.

The Formulation

The first question to consider when building a set based model is, "What are the relevant sets and their attributes?". In this model, we have a single primitive set, the days of the week. We will be concerned with two attributes of the *DAYS* set. The first is the number of staff required on each day. The second is the decision variable of the number of staff to start on each day. If we call these attributes *REQUIRED* and *START*, then we might write the SETS section and DATA sections as:

```
SETS:
  DAYS : REQUIRED, START;
ENDSETS
DATA:
      DAYS = MON TUE WED THU FRI SAT SUN;
  REQUIRED = 20  16  13  16  19  14  12;
ENDDATA
```

We are now at the point where we can begin entering the model's mathematical relations (i.e., the objective and constraints). Let's begin by writing the objective: minimize the total number of employees we start during the week. In standard mathematical notation, we might write:

Minimize: $\sum_{i} START_i$

The equivalent LINGO statement is very similar. Substitute "MIN=" for "Minimize:" and "@SUM(DAYS(I):" for $\sum_i$ and we have:

```
MIN = @SUM( DAYS( I): START( I));
```

Now, all that's left is to deduce the constraints. There is only one set of constraints in this model. Namely, we must have enough staff on duty each day to meet or exceed staffing requirements. In words, what we want is:

for each day: Staff on duty today ≥ Staff required today, The right-hand side of this expression, *Staff required today,* is given. It is simply the quantity `REQUIRED( I)`. The left-hand side, *Staff on duty today* takes a little thought. Given that all employees are on a five-day on/two day off schedule, the number of employees working today is:

Number working today = Number starting today +
Number starting 1 *day ago + Number starting* 2 *days ago +*
Number starting 3 *days ago + Number starting* 4 *days ago*.

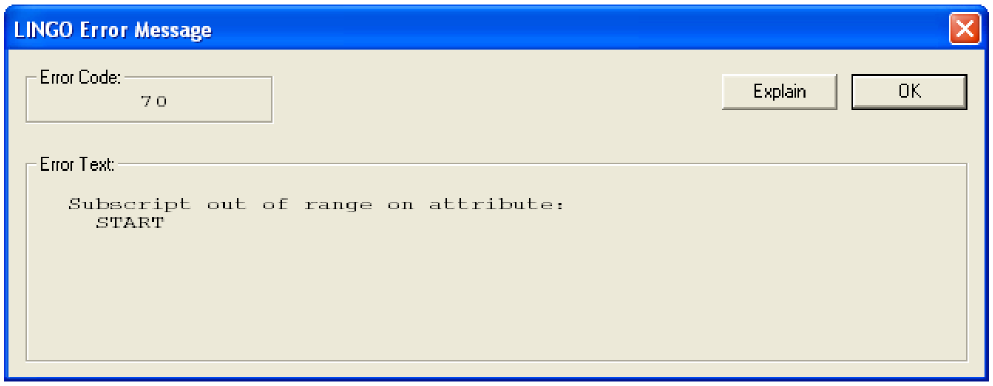
In other words, to compute the number of employees working today, we sum up the number of people starting today plus those starting over the previous four days. The employees starting five and six days back don't count because they are on their days off. Therefore, using mathematical notation, what one might consider doing is adding the constraint:

$$\sum_{i = j - 4, j} \text{START}_i \geq \text{REQUIRED}_j, \text{ for } j \in \text{DAYS}$$

Translating into LINGO notation, we can write this as:

```
@FOR( DAYS( J):
   @SUM( DAYS( I) | I #LE# 5: START( J - I + 1))
    >= REQUIRED( J)
);
```

In words, the LINGO statement says, for each day of the week, the sum of the employees starting over the five-day period beginning four days ago and ending today must be greater-than-or-equal-to the required number of staff for the day. This sounds correct, but there is a slight problem. If we try to solve our model with this constraint, we get the error message:


LINGO Error Message

Error Code:
70

Error Text:

```
Subscript out of range on attribute:
  START
```

To see why we get this error message, consider what happens on Thursday. Thursday has an index of 4 in our set *DAYS*. As written, the staffing constraint for Thursday will be:

```
START( 4 - 1 + 1) + START( 4 - 2 + 1) +
START( 4 - 3 + 1) + START( 4 - 4 + 1) +
START( 4 - 5 + 1) >= REQUIRED( 4);
```

Simplifying, we get:

```
START( 4) + START( 3) +
START( 2) + START( 1) +
START( 0) >= REQUIRED( 4);
```

It is the *START(0)* term that is at the root of our problem. *START* is defined for days 1 through 7. *START(0)* does not exist. An index of 0 on *START* is considered "out of range".

What we would like to do is to have any indices less-than-or-equal-to 0, *wrap around* to the end of the week. Specifically, 0 would correspond to Sunday (7), -1 to Saturday (6), and so on. LINGO has a function that does just this, and it is called *@WRAP*.

The *@WRAP* function takes two arguments - call them *INDEX* and *LIMIT*. Formally speaking, *@WRAP* returns *J* such that *J* = *INDEX* - *K* × *LIMIT*, where *K* is an integer such that *J* is in the interval [1,*LIMIT*]. Informally speaking, *@WRAP* will subtract or add *LIMIT* to *INDEX* until it falls in the range 1 to *LIMIT*, and, therefore, is just what we need to "wrap around" an index in multi-period planning models.

Incorporating the *@WRAP* function, we get the corrected, final version of our staffing constraint:

```
@FOR( DAYS( J):
 @SUM( DAYS( I) | I #LE# 5:
  START( @WRAP( J - I + 1, 7))) >= REQUIRED( J)
);
```

The Solution

Below is our staffing model in its entirety:

```
SETS:
  DAYS : REQUIRED, START;
ENDSETS
DATA:
      DAYS = MON TUE WED THU FRI SAT SUN;
  REQUIRED = 20  16  13  16  19  14  12;
ENDDATA
MIN = @SUM( DAYS( I): START( I));
@FOR( DAYS( J):
  @SUM( DAYS( I) | I #LE# 5:
     START( @WRAP( J - I + 1, 7))) >= REQUIRED( J)
     );
```

Solving this model, we get the solution report:

```
Optimal solution found at step:         8
Objective value:                 22.00000

      Variable           Value        Reduced Cost
REQUIRED( MON)        20.00000           0.0000000
REQUIRED( TUE)        16.00000           0.0000000
REQUIRED( WED)        13.00000           0.0000000
REQUIRED( THU)        16.00000           0.0000000
REQUIRED( FRI)        19.00000           0.0000000
REQUIRED( SAT)        14.00000           0.0000000
REQUIRED( SUN)        12.00000           0.0000000
   START( MON)         8.00000           0.0000000
   START( TUE)         2.00000           0.0000000
   START( WED)         0.00000           0.0000000
   START( THU)         6.00000           0.0000000
   START( FRI)         3.00000           0.0000000
   START( SAT)         3.00000           0.0000000
   START( SUN)         0.00000           0.0000000
```

```
      Row    Slack or Surplus      Dual Price
        1        22.00000            1.000000
        2       0.0000000          -0.2000000
        3       0.0000000          -0.2000000
        4       0.0000000          -0.2000000
        5       0.0000000          -0.2000000
        6       0.0000000          -0.2000000
        7       0.0000000          -0.2000000
        8       0.0000000          -0.2000000
```

The objective value of 22 means we need to hire 22 workers.

We start our workers according to the schedule:

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Start	8	2	0	6	3	3	0

If we look at the surpluses on our staffing requirement rows (rows 2 - 7), we see the slack values are 0 on all of the days. This means there are no extra workers on any day.

5.5.2 Dense Derived Set Example

The following model illustrates the use of a dense derived set in a blending model. This model may be found in the *SAMPLES* subdirectory off the main LINGO directory under the name *CHESS.LNG*.

The Problem

The Chess Snackfoods Co. markets four brands of mixed nuts. The four brands of nuts are called the *Pawn*, *Knight*, *Bishop*, and *King*. Each brand contains a specified ratio of peanuts and cashews. The table below lists the number of ounces of the two nuts contained in each pound of each brand and the price at which the company can sell a pound of each brand:

	Pawn	Knight	Bishop	King
Peanuts (oz.)	15	10	6	2
Cashews (oz.)	1	6	10	14
Selling Price ($/lb.)	2	3	4	5

Chess has contracts with suppliers to receive per day: 750 pounds of peanuts and 250 pounds of cashews. Our problem is to determine the number of pounds of each brand to produce each day to maximize total revenue without exceeding the available supply of nuts.

The Formulation

The primitive sets in this model are the nut types and the brands of mixed nuts. The *NUTS* set has the single attribute *SUPPLY* that is the daily supply of nuts in pounds. The *BRANDS* set has *PRICE* and *PRODUCE* attributes, where *PRICE* stores the selling price of the brands and *PRODUCE* represents the decision variables of how many pounds of each brand to produce each day.

We need one more set, however, in order to input the brand formulas. We need a two dimensional table defined on the nut types and the brands. To do this, we will generate a derived set from the cross of the *NUTS* and *BRANDS* sets. Adding this derived set, we get the complete SETS section:

```
SETS:
   NUTS : SUPPLY;
   BRANDS : PRICE, PRODUCE;
   FORMULA( NUTS, BRANDS): OUNCES;
ENDSETS
```

We have titled the derived set *FORMULA*, and it has the single attribute *OUNCES*, which will be used to store the ounces of nuts used per pound of each brand. Since we have not specified the members of this derived set, LINGO assumes we want the complete, dense set that includes all pairs of nuts and brands.

Now that our sets are defined, we can move on to building the DATA section. We initialize the three attributes *SUPPLY*, *PRICE*, and *OUNCES* in the DATA section as follows:

```
DATA:
    NUTS = PEANUTS, CASHEWS;
  SUPPLY =   750      250;
  BRANDS = PAWN, KNIGHT, BISHOP, KING;
     PRICE =  2      3      4      5;
    OUNCES = 15     10      6      2  !(Peanuts);
              1      6     10     14; !(Cashews);
ENDDATA
```

With the sets and data specified, we can enter our objective function and constraints. The objective function of maximizing total revenue is straightforward:

```
MAX = @SUM( BRANDS( I): PRICE( I) * PRODUCE( I));
```

Our model has only one class of constraints. Namely, we can't use more nuts than we are supplied with on a daily basis. In words, we would like to ensure that:

For each nut type i, the number of pounds of nut i used must be less-than-or-equal-to the supply of nut i.

We can express this in LINGO as:

```
@FOR( NUTS( I):
   @SUM( BRANDS( J):
   OUNCES( I, J) * PRODUCE( J) / 16) <= SUPPLY( I)
);
```

We divide the sum on the left-hand side by 16 to convert from ounces to pounds.

The Solution
Our completed nut-blending model is:

```
SETS:
   NUTS : SUPPLY;
   BRANDS : PRICE, PRODUCE;
   FORMULA( NUTS, BRANDS): OUNCES;
ENDSETS
DATA:
    NUTS = PEANUTS, CASHEWS;
  SUPPLY =   750      250;
  BRANDS = PAWN, KNIGHT, BISHOP, KING;
   PRICE =   2      3      4      5;
   OUNCES = 15     10      6      2  !(Peanuts);
             1      6     10     14; !(Cashews);
ENDDATA
MAX = @SUM( BRANDS( I):
 PRICE( I) * PRODUCE( I));
 @FOR( NUTS( I):
   @SUM( BRANDS( J):
     OUNCES( I, J) * PRODUCE(J)/16) <= SUPPLY(I)
     );
```

An abbreviated solution report to the model follows:

```
Optimal solution found at step:         0
Objective value:                 2692.308
        Variable        Value   Reduced Cost
  PRODUCE( PAWN)      769.2308     0.0000000
PRODUCE( KNIGHT)     0.0000000     0.1538461
PRODUCE( BISHOP)     0.0000000     0.7692297E-01
  PRODUCE( KING)      230.7692     0.0000000
           Row  Slack or Surplus  Dual Price
             1        2692.308      1.000000
             2       0.0000000      1.769231
             3       0.0000000      5.461538
```

This solution tells us Chess should produce 769.2 pounds of the Pawn mix and 230.8 of the King for total revenue of \$2692.30. The dual prices on the rows indicate Chess should be willing to pay up to \$1.77 for an extra pound of peanuts and \$5.46 for an extra pound of cashews. If, for marketing reasons, Chess decides it must produce at least some of the Knight and Bishop mixes, then the reduced cost figures tell us revenue will decrease by 15.4 cents with the first pound of Knight produced and revenue will decline by 76.9 cents with the first pound of Bishop produced.

5.5.3 Sparse Derived Set Example - Explicit List

In this example, we will introduce the use of a sparse derived set with an explicit listing. When using this method to define a sparse set, we must explicitly list all members of the set. This will usually be some small subset of the dense set resulting from the full Cartesian product of the parent sets.

For our example, we will set up a PERT (*Program Evaluation and Review Technique)* model to determine the *critical path* of tasks in a project involving the roll out of a new product. PERT is a simple, but powerful, technique developed in the 1950s to assist managers in tracking the progress of large projects. Its first official application was to the fleet submarine ballistic missile project, the so-called Polaris project. In fact, PERT proved so successful the Polaris project was completed eighteen months ahead of schedule! PERT is particularly useful at identifying the critical activities within a project, which, if delayed, will delay the project as a whole. These time critical activities are referred to as the critical path of a project. Having such insight into the dynamics of a project goes a long way in guaranteeing it won't get sidetracked and become delayed. PERT, and a closely related technique called CPM (Critical Path Method), continues to be used successfully on a wide range of projects. The formulation for this model is included in the *SAMPLES* subdirectory off the main LINGO directory under the name *PERTD.LNG*.

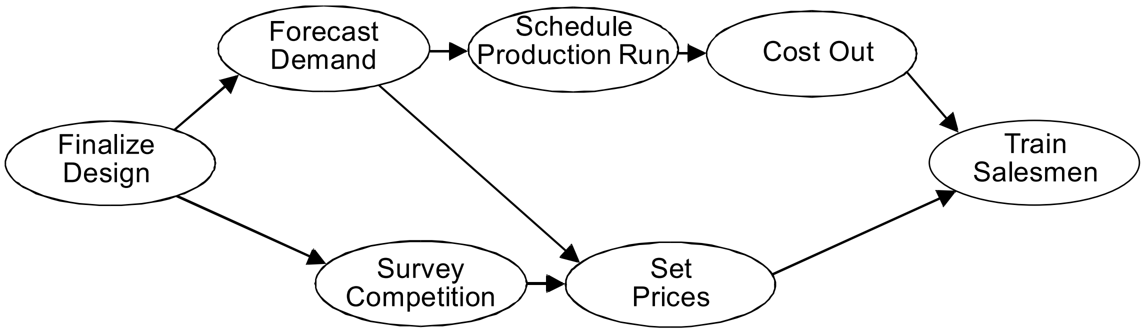
The Problem

Wireless Widgets is about to launch a new product — the Solar Widget. In order to guarantee the launch will occur on time, WW wants to perform a PERT analysis of the tasks leading up to the launch. Doing so will allow them to identify the critical path of tasks that must be completed on time in order to guarantee the Solar Widget's timely introduction. The tasks that must be accomplished before introduction and their anticipated times for completion are listed in the table below:

Task	Weeks
Finalize Design	10
Forecast Demand	14
Survey Competition	3
Set Prices	3
Schedule Production Run	7
Cost Out	4
Train Salesmen	10

Certain tasks must be completed before others can commence. These precedence relations are shown in Figure 5.2:

Figure 5.2 Product Launch Precedence Relations



For instance, the two arrows originating from the *Forecast Demand* node indicate the task must be completed before the *Schedule Production Run* and the *Set Prices* tasks may be started.

Our goal is to construct a PERT model for the Solar Widget's introduction in order to identify the tasks on the critical path.

The Formulation

We will need a primitive set to represent the tasks of the project.

We have associated four attributes with the *TASKS* set. The definitions of the attributes are:

TIME	Time duration to complete the task, given
ES	Earliest possible start time for the task, to be computed,
LS	Latest possible start time for the task, to be computed
SLACK	Difference between LS and ES for the task, to be computed.

If a task has a 0 slack, it means the task must start on time or the whole project will be delayed. The collection of tasks with 0 slack time constitutes the critical path for the project.

In order to compute the start times for the tasks, we will need to examine the precedence relations. Thus, we will need to input the precedence relations into the model. The precedence relations can be viewed as a list of ordered pairs of tasks. For instance, the fact the *DESIGN* task must be completed before the *FORECAST* task could be represented as the ordered pair (*DESIGN*, *FORECAST*). Creating a two-dimensional derived set on the *TASKS* set will allow us to input the list of precedence relations. Therefore, our DATA section will look as follows:

```
DATA:
 TASKS : TIME, ES, LS, SLACK;
PRED( TASKS, TASKS);
```

Notice that the *PRED* set has no attributes. Its purpose is only to provide the information about the precedence relationships between tasks.

Next, we can input the task times and precedence pairs in the DATA section thus:

```
DATA:
 TASKS= DESIGN, FORECAST, SURVEY, PRICE, SCHEDULE, COSTOUT, TRAIN;
 TIME =   10,      14,      3,      3,      7,        4,       10;
 PRED =
    DESIGN,FORECAST,
    DESIGN,SURVEY,
    FORECAST,PRICE,
    FORECAST,SCHEDULE,
    SURVEY,PRICE,
    SCHEDULE,COSTOUT,
    PRICE,TRAIN,
    COSTOUT,TRAIN;
ENDDATA
```

Keep in mind that the first member of the *PRED* set is the ordered pair (*DESIGN*, *FORECAST*) and not just the single task *DESIGN*. Therefore, this set has a total of 8 members. Each of which corresponds to an arc in the precedence relations diagram.

The feature to note from this example is that the set *PRED* is a sparse derived set with an explicit listing of members. The set is a subset derived from the cross of the *TASKS* set upon itself. The set is sparse because it contains only 8 out of a possible 49 members found in the complete cross of *TASKS* on *TASKS*. The set has an explicit listing because we have included a listing of the members we want included in the set. Explicitly listing the members of a sparse set may not be convenient in cases where there are thousands of members to select from, but it does make sense whenever set membership conditions are not well-defined and the sparse set size is small relative to the dense alternative.

Now, with our sets and data established, we can turn our attention to building the formulas of the model. We have three attributes to compute: earliest start (*ES*), latest start (*LS*), and slack time (*SLACK*). The trick is computing *ES* and *LS*. Once we have these times, *SLACK* is merely the difference of the two. Let's start by deriving a formula to compute *ES*. A task cannot begin until all its predecessor tasks are completed. Thus, if we find the latest finishing time of all predecessors to a task, then we have also found its earliest start time. Therefore, in words, the earliest start time for task *t* is equal to the maximum of the sum of the earliest start time of the predecessor plus its completion time over all predecessors of task *t*. The corresponding LINGO notation is:

```
@FOR( TASKS( J)| J #GT# 1:
   ES( J) = @MAX( PRED( I, J): ES( I) + TIME( I)));
```

Note, we skip the computation for task 1 by adding the conditional qualifier *J #GT#* 1. We do this because task 1 has no predecessors. We will give the first task an arbitrary start time of 0 below.

Computing *LS* is similar to *ES*, except we must think backwards. In words, the latest time for task *t* to start is the minimum, over all successor tasks *j*, of *j*'s latest start minus the time to perform task *t*. If task *t* starts any later than this, it will force at least one successor to start later than its latest start time. Converting into LINGO syntax gives:

```
@FOR( TASKS( I)| I #LT# LTASK:
   LS( I) = @MIN( PRED( I, J): LS( J) - TIME( I)));
```

Here, we omit the computation for the last task, since it has no successor tasks.

Computing slack time is just the difference between *LS* and *ES* and may be written as:

```
@FOR( TASKS( I): SLACK( I) = LS( I) - ES( I));
```

We can set the start time of task 1 to some arbitrary value. For our purposes, we will set it to 0 with the statement:

```
ES( 1) = 0;
```

We have now input formulas for computing the values of all the variables with the exception of the latest start time for the last task. It turns out, if the last project were started any later than its earliest start time, the entire project would be delayed. So, by definition, the latest start time for the last project is equal to its earliest start time. We can express this in LINGO using the equation:

```
LS( 7) = ES( 7);
```

This would work, but it is not a very general way to express the relation. Suppose you were to add some tasks to your model. You'd have to change the 7 in this equation to whatever the new number of tasks was. The whole idea behind LINGO's set based modeling language is the equations in the model should not need changing each time the data change. Expressing the equation in this form violates data independence. Here's a better way to do it:

```
LTASK = @SIZE( TASKS);
LS( LTASK) = ES( LTASK);
```

The *@SIZE* function returns the size of a set. In this case, it will return the value 7, as desired. However, if we changed the number of tasks, *@SIZE* would also return the new, correct value. Thus, we preserve the data independence of our model's structure.

The Solution

The entire PERT formulation and portions of its solution appear below:

```
SETS:
   TASKS : TIME, ES, LS, SLACK;
   PRED( TASKS, TASKS);
ENDSETS
DATA:
 TASKS= DESIGN, FORECAST, SURVEY, PRICE, SCHEDULE, COSTOUT, TRAIN;
 TIME =   10,      14,      3,      3,      7,       4,       10;
 PRED =
    DESIGN,FORECAST,
    DESIGN,SURVEY,
    FORECAST,PRICE,
    FORECAST,SCHEDULE,
    SURVEY,PRICE,
    SCHEDULE,COSTOUT,
    PRICE,TRAIN,
    COSTOUT,TRAIN;
ENDDATA

@FOR( TASKS( J)| J #GT# 1:
   ES( J) = @MAX( PRED( I, J): ES( I) + TIME( I))
    );
@FOR( TASKS( I)| I #LT# LTASK:
   LS( I) = @MIN( PRED( I, J): LS( J) - TIME( I));
    );
@FOR( TASKS( I): SLACK( I) = LS( I) - ES( I));
ES( 1) = 0;
LTASK = @SIZE( TASKS);
LS( LTASK) = ES( LTASK);
```

The interesting part of the solution is:

```
             Variable           Value
                LTASK        7.000000
          ES( DESIGN)        0.000000
        ES( FORECAST)       10.000000
          ES( SURVEY)       10.000000
           ES( PRICE)       24.000000
        ES( SCHEDULE)       24.000000
         ES( COSTOUT)       31.000000
           ES( TRAIN)       35.000000
          LS( DESIGN)        0.000000
        LS( FORECAST)       10.000000
          LS( SURVEY)       29.000000
           LS( PRICE)       32.000000
        LS( SCHEDULE)       24.000000
         LS( COSTOUT)       31.000000
           LS( TRAIN)       35.000000
       SLACK( DESIGN)        0.000000
     SLACK( FORECAST)        0.000000
       SLACK( SURVEY)       19.000000
        SLACK( PRICE)        8.000000
     SLACK( SCHEDULE)        0.000000
      SLACK( COSTOUT)        0.000000
        SLACK( TRAIN)        0.000000
```

The interesting values are the slacks for the tasks. *SURVEY* and *PRICE* have respective slacks of 19 and 8. The start time of either *SURVEY* or *PRICE* (but not both) may be delayed by as much as these slack values without delaying the completion time of the entire project. The tasks *DESIGN*, *FORECAST*, *SCHEDULE*, *COSTOUT*, and *TRAIN*, on the other hand, have 0 slack. These tasks constitute the critical path. If any of their start times are delayed, the entire project will be delayed. Management will want to pay close attention to these critical path activities to be sure they start on time and complete within the allotted time. Finally, the *ES*(*TRAIN*) value of 35 tells us the estimated time to the start of the roll out of the new Solar Widget will be 45 weeks: 35 weeks to get to the start of training, plus 10 weeks to complete training.

5.5.4 A Sparse Derived Set Using a Membership Filter

In this example, we introduce the use of a sparse derived set with a membership filter. Using a membership filter is the third method for defining a derived set. When you define a set using this method, you specify a logical condition each member of the set must satisfy. This condition is used to filter out members that don't satisfy the membership condition.

For our example, we will formulate a *matching* problem. In a matching problem, there are *N* objects we want to match into pairs at minimum cost. Sometimes this is known as the roommate selection problem. It is a problem faced by a university at the beginning of each school year as incoming first year students are assigned to rooms in dormitories. The pair *(I,J)* is indistinguishable from the pair *(J,I)*. Therefore, we arbitrarily require *I* be less than *J* in the pair. Formally, we require *I* and *J* make a set of ordered pairs. In other words, we do not wish to generate redundant ordered pairs of *I* and *J,* but only those with *I* less than *J*. This requirement that *I* be less than *J* will form our membership filter.

The file containing this model may be found in the *SAMPLES* subdirectory off the main LINGO directory under the name *MATCHD.LNG.*

The Problem

Suppose you manage your company's strategic planning department. There are eight analysts in the department. Your department is about to move into a new suite of offices. There are four offices in the new suite and you need to match up your analysts into 4 pairs, so each pair can be assigned to one of the new offices. Based on past observations, you know some of the analysts work better together than they do with others. In the interest of departmental peace, you would like to come up with a pairing of analysts that results in minimal potential conflicts. To this goal, you have come up with a rating system for pairing your analysts. The scale runs from 1 to 10, with a 1 rating for a pair meaning the two get along fantastically, whereas all sharp objects should be removed from the pair's office in anticipation of mayhem for a rating of 10. The ratings appear in the following table:

Analysts	1	2	3	4	5	6	7	8
1	-	9	3	4	2	1	5	6
2	-	-	1	7	3	5	2	1
3	-	-	-	4	4	2	9	2
4	-	-	-	-	1	5	5	2
5	-	-	-	-	-	8	7	6
6	-	-	-	-	-	-	2	3
7	-	-	-	-	-	-	-	4

Analysts' Incompatibility Ratings

Since the pairing of analyst *I* with analyst *J* is indistinguishable from the pairing of *J* with *I*, we have only included the above diagonal elements in the table. Our problem is to find the pairings of analysts that minimizes the sum of the incompatibility ratings of the paired analysts.

The Formulation

The first set of interest in this problem is the set of eight analysts. This primitive set can be written simply as:

```
ANALYSTS;
```

The final set we want to construct is a set consisting of all the potential pairings. This will be a derived set we will build by taking the cross of the *ANALYST* set. As a first pass, we could build the dense derived set:

```
PAIRS( ANALYSTS, ANALYSTS);
```

This set, however, would include both *PAIRS(I, J)* and *PAIRS(J, I)*. Since only one of these pairs is required, the second is wasteful. Furthermore, this set will include "pairs" of the same analyst of the form *PAIRS(I, I)*. As much as each of the analysts might like an office of their own, such a solution is not feasible. The solution is to put a membership filter on our derived set requiring each pair *(I,J)* in the final set to obey the condition *J* be greater than *I*. We do this with the set definition:

```
PAIRS( ANALYSTS, ANALYSTS) | &2 #GT# &1;
```

The start of the membership filter is denoted with the vertical bar character (|). The *&1* and *&2* symbols in the filter are known as *set index placeholders*. Set index placeholders are valid only in membership filters. When LINGO constructs the *PAIRS* set, it generates all combinations in the cross of the *ANALYSTS* set on itself. Each combination is then "plugged" into the membership filter to see if it passes the test. Specifically, for each pair *(I,J)* in the cross of set *ANALYSTS* on itself, *I* is substituted into the placeholder *&1* and *J* into *&2* and the filter is evaluated. If the filter evaluates to true, *(I,J)* is added to the pairs set. Viewed in tabular form, this leaves us with just the above diagonal elements of the *(I,J)* pairing table.

We will also be concerned with two attributes of the *PAIRS* set. First, we will need an attribute that corresponds to the incompatibility rating of the pairings. Second, we will need an attribute to indicate if analyst *I* is paired with analyst *J*. We will call these attributes *RATING* and *MATCH*. We append them to the *PAIRS* set definition as follows:

```
PAIRS( ANALYSTS, ANALYSTS) | &2 #GT# &1: RATING, MATCH;
```

We will simply initialize the *RATING* attribute to the incompatibility ratings listed in the table above using the DATA section:

```
DATA:
  ANALYSTS = 1..8;
   RATING =
      9  3  4  2  1  5  6
         1  7  3  5  2  1
            4  4  2  9  2
               1  5  5  2
                  8  7  6
                     2  3
                        4;
ENDDATA
```

We will use the convention of letting *MATCH(I, J)* be 1 if we pair analyst *I* with analyst *J*, otherwise 0. As such, the *MATCH* attribute contains the decision variables for the model.

Our objective is to minimize the sum of the incompatibility ratings of all the final pairings. This is just the inner product on the *RATING* and *MATCH* attributes and is written as:

```
MIN = @SUM( PAIRS( I, J):
   RATING( I, J) * MATCH( I, J));
```

There is just one class of constraints in the model. In words, what we want to do is:

For each analyst, ensure the analyst is paired with exactly one other analyst.

Putting the constraint into LINGO syntax, we get:

```
@FOR( ANALYSTS( I):
  @SUM( PAIRS( J, K) | J #EQ# I #OR# K #EQ# I:
    MATCH( J, K)) = 1
    );
```

The feature of interest in this constraint is the conditional qualifier *J #EQ# I #OR# K #EQ# I* on the *@SUM* function. For each analyst *I*, we sum up all the *MATCH* variables that contain *I* and set them equal to 1. In so doing, we guarantee analyst *I* will be paired up with exactly one other analyst. The conditional qualifier guarantees we only sum up the *MATCH* variables that include *I* in its pairing.

One other feature is required in this model. We are letting *MATCH*(*I*, *J*) be 1 if we are pairing *I* with *J*. Otherwise, it will be 0. Unless specified otherwise, LINGO variables can assume any value from 0 to infinity. Since we want *MATCH* to be restricted to being only 0 or 1, we need to add one other feature to our model. What we need is to apply the *@BIN* variable domain function to the *MATCH* attribute. *Variable domain functions* are used to restrict the values a variable can assume. Unlike constraints, variable domain functions do not add equations to a model. The *@BIN* function restricts a variable to being *binary* (i.e., 0 or 1). When you have a model that contains binary variables, it is said to be an *integer programming* (IP) model. IP models are much more difficult to solve than models that contain only continuous variables. Carelessly formulated IPs (with several hundred integer variables or more) can literally take *forever* to solve! Thus, you should limit the use of binary variables whenever possible. To apply *@BIN* to all the variables in the *MATCH* attribute, add the *@FOR* expression:

```
@FOR( PAIRS( I, J): @BIN( MATCH( I, J)));
```

The Solution

The entire formulation for our matching example and parts of its solution appears below:

```
SETS:
   ANALYSTS;
   PAIRS( ANALYSTS, ANALYSTS) | &2 #GT# &1:
    RATING, MATCH;
ENDSETS
DATA:
   ANALYSTS = 1..8;
   RATING =
      9  3  4  2  1  5  6
         1  7  3  5  2  1
            4  4  2  9  2
               1  5  5  2
                  8  7  6
                     2  3
                        4;
ENDDATA
MIN = @SUM( PAIRS( I, J):
   RATING( I, J) * MATCH( I, J));
@FOR( ANALYSTS( I):
  @SUM( PAIRS( J, K) | J #EQ# I #OR# K #EQ# I:
                              MATCH( J, K)) = 1
     );
@FOR( PAIRS( I, J): @BIN( MATCH( I, J)));
```

A solution is:

```
     Variable           Value
 MATCH( 1, 2)       0.0000000
 MATCH( 1, 3)       0.0000000
 MATCH( 1, 4)       0.0000000
 MATCH( 1, 5)       0.0000000
 MATCH( 1, 6)        1.000000
 MATCH( 1, 7)       0.0000000
 MATCH( 1, 8)       0.0000000
 MATCH( 2, 3)       0.0000000
 MATCH( 2, 4)       0.0000000
 MATCH( 2, 5)       0.0000000
 MATCH( 2, 6)       0.0000000
 MATCH( 2, 7)        1.000000
```

```
MATCH( 2, 8)        0.0000000
MATCH( 3, 4)        0.0000000
MATCH( 3, 5)        0.0000000
MATCH( 3, 6)        0.0000000
MATCH( 3, 7)        0.0000000
MATCH( 3, 8)         1.000000
MATCH( 4, 5)         1.000000
MATCH( 4, 6)        0.0000000
MATCH( 4, 7)        0.0000000
MATCH( 4, 8)        0.0000000
MATCH( 5, 6)        0.0000000
MATCH( 5, 7)        0.0000000
MATCH( 5, 8)        0.0000000
MATCH( 6, 7)        0.0000000
MATCH( 6, 8)        0.0000000
MATCH( 7, 8)        0.0000000
```

Notice from the objective value, the total sum of incompatibility ratings for the optimal pairings is 6. Scanning the *Value* column for 1's, we find the optimal pairings: (1,6), (2,7), (3,8), and (4,5).

5.6 Domain Functions for Variables

Variable domain functions were briefly introduced in this chapter when we used *@BIN* in the previous matching model. Variable domain functions allow one to put restrictions on the values allowed for decision variables. Examples of the four domain functions available are:

@BIN (*Y*);
@GIN (*X*);
@BND (100, *DELIVER*, 250);
@FREE (*PROFIT*);.

The statement *@BIN* (*Y*) restricts the variable *Y* to be a binary variable. That is, it can take on only the values 0 and 1.

The statement *@GIN* (*X*) restricts the variable *X* to be a general integer variable. That is, it can take on only the values 0, 1, 2, …

The *@BND* () specification allows one to specify simple upper and lower bounds. The statement *@BND* (100, *DELIVER*, 250) restricts the variable *DELIVER* to be in the interval [100, 250]. The same effect could be achieved by the slightly more verbose:

DELIVER >= 100;
DELIVER <= 250;

LINGO, by default, gives a lower bound of zero to every decision variable. The statement *@FREE* (*PROFIT*) overrides this default lower bound for the variable *PROFIT* and says that (unfortunately) *PROFIT* can take on any value between minus infinity and plus infinity. Each of the domain functions can appear inside *@FOR* loops, just like any other constraint.

5.7 Spreadsheets and LINGO

In this chapter, we have seen how LINGO can be useful for modeling very large problems. The most widely used method for modeling of any sort is undoubtedly spreadsheet models. When is which approach more appropriate?

The major advantages of doing a model in a spreadsheet are:

- Excellent report formatting features available,
- Large audience of people who understand spreadsheets, and
- Good interface capability with other systems such as word processors.

The major advantages of doing a model in LINGO are:

- Flexibility of various kinds.
- Scalability--It is easy to change the size of any set (e.g., add time periods, products, customers, suppliers, transportation modes, etc.) without having to worry about copying or editing formulae. There is no upper limit of 255(as in a spreadsheet) on the number of columns, or 65536 on the number of rows.
- Sparse sets are easily represented.
- Auditability and visibility--It is easy to see the formulae of a LINGO model in complete, comprehensive form. Truly understanding the model formulae underlying a complex spreadsheet is an exercise in detective work.
- Multiple dimensions are easily represented. A spreadsheet handles two dimensions very well, three dimensions somewhat well, and four or more dimensions not very well.

One can get most of the benefits of both by using LINGO in conjunction with spreadsheets. One can place "hooks" in a LINGO model, so it automatically retrieves and inserts data from/to spreadsheets, databases, and ordinary files. Under Microsoft Windows, the hooks used are the OLE (Object Linking and Embedding) and ODBC (Open Database Connectivity) interfaces provided as part of Windows. Using the OLE capability to connect an Excel spreadsheet to a LINGO model requires two steps:

a) In the spreadsheet, each data area that is to be either a supplier to or a receiver of data from the LINGO model must be given an appropriate range name. This is done in the spreadsheet by highlighting the area of interest with the mouse, and then using the *Insert | Name | Define* command. The most convenient name to give to a range is the same name by which the data are referenced in the LINGO model.

b) In the LINGO model, each attribute (vector) (e.g., plant capacities) that is to be retrieved from a spreadsheet, must appear in a LINGO DATA section in a statement of the form:

    CAPACITY = @OLE('C:\MYDATA.XLS');

Each attribute (e.g., amount to ship) to be sent to a spreadsheet must appear in a LINGO DATA section in a statement of the form:

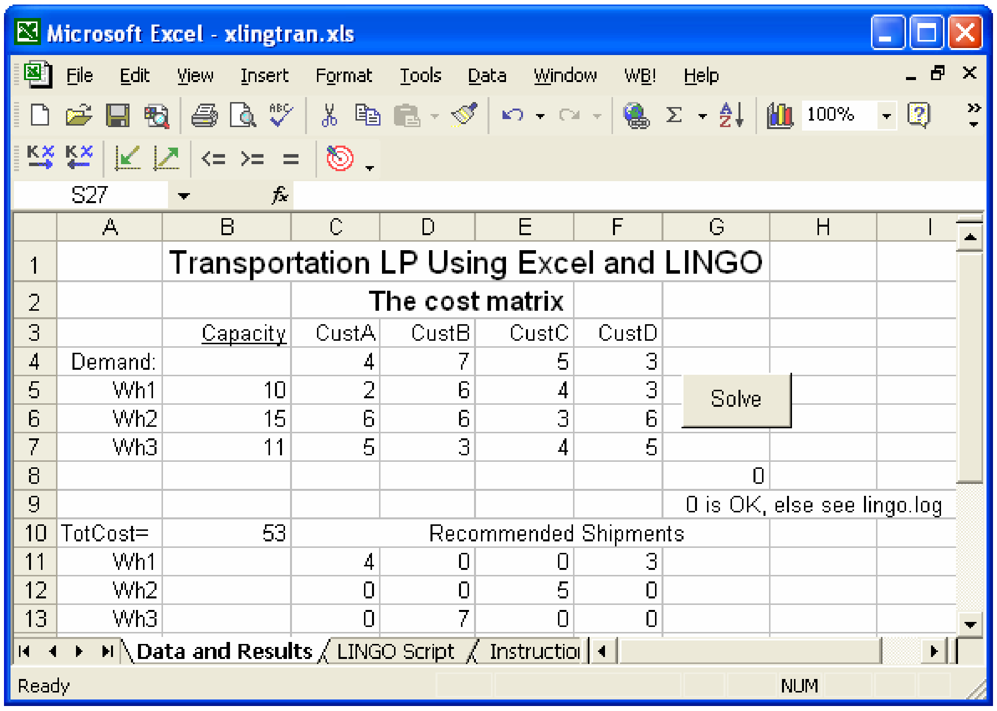
@OLE('C:\MYDATA.XLS') = AMT_SHIPPED;

If only one spreadsheet is open in Excel, this connection can be simplified. You need only write:

CAPACITY = @OLE();

LINGO will look in the only open spreadsheet for the range called CAPACITY. This “unspecified spreadsheet” feature is very handy if you want to apply the same LINGO model to several different spreadsheet data sets.

This spreadsheet connection can be pushed even further by embedding the LINGO model in the spreadsheet for which it has a data connection. This is handy because the associated LINGO model will always be obviously and immediately available when the spreadsheet is opened. The screen shot below shows a transportation model embedded in a spreadsheet. To the casual user, it looks like a standard spreadsheet with a special solve button.

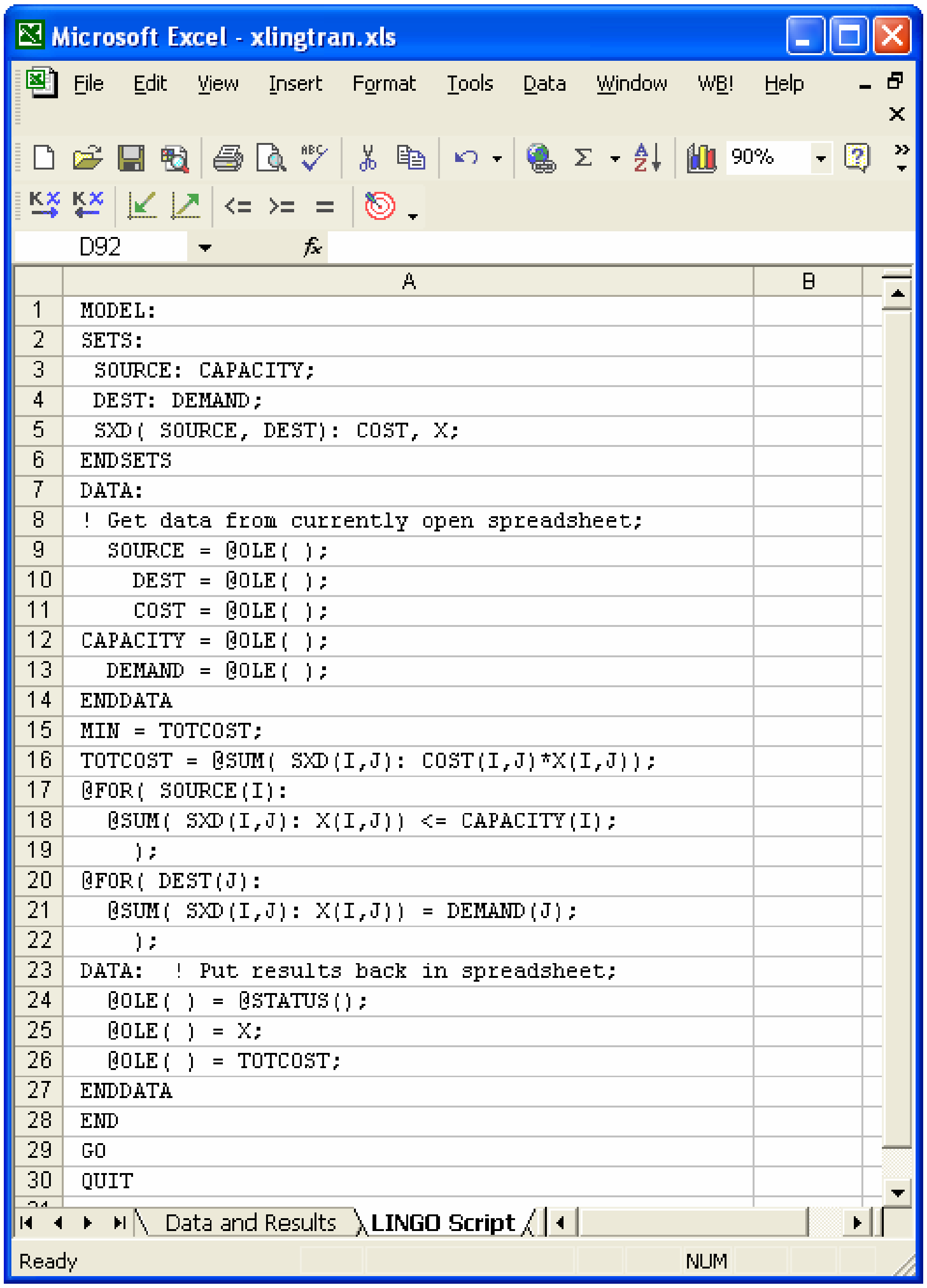

Microsoft Excel - xlingtran.xls
File Edit View Insert Format Tools Data Window WB! Help
100%
S27
Solve
Data and Results
LINGO Script
Instructio
Ready
NUM

Transportation LP Using Excel and LINGO

The cost matrix

	A	B	C	D	E	F	G
3		Capacity	CustA	CustB	CustC	CustD	
4	Demand:		4	7	5	3	
5	Wh1	10	2	6	4	3	
6	Wh2	15	6	6	3	6	
7	Wh3	11	5	3	4	5	
8							0
9							0 is OK, else see lingo.log
10	TotCost=	53	Recommended Shipments				
11	Wh1		4	0	0	3	
12	Wh2		0	0	5	0	
13	Wh3		0	7	0	0	

The data and results are stored on the first tab/sheet of the spreadsheet file. Not so obvious is the LINGO model that is stored on another tab in the same spreadsheet (see below). Completely hidden is a small VBA program in the spreadsheet that causes the LINGO model on the second tab to be solved whenever the Solve button is clicked on the first tab. The complete example can be found in the file xlingtran.xls.


Microsoft Excel - xlingtran.xls
File Edit View Insert Format Tools Data Window WB! Help
90%
D92
A
B
Data and Results
LINGO Script
Ready
NUM

```
MODEL:
SETS:
 SOURCE: CAPACITY;
 DEST: DEMAND;
 SXD( SOURCE, DEST): COST, X;
ENDSETS
DATA:
! Get data from currently open spreadsheet;
  SOURCE = @OLE( );
    DEST = @OLE( );
    COST = @OLE( );
CAPACITY = @OLE( );
  DEMAND = @OLE( );
ENDDATA
MIN = TOTCOST;
TOTCOST = @SUM( SXD(I,J): COST(I,J)*X(I,J));
@FOR( SOURCE(I):
  @SUM( SXD(I,J): X(I,J)) <= CAPACITY(I);
    );
@FOR( DEST(J):
  @SUM( SXD(I,J): X(I,J)) = DEMAND(J);
    );
DATA:  ! Put results back in spreadsheet;
  @OLE( ) = @STATUS();
  @OLE( ) = X;
  @OLE( ) = TOTCOST;
ENDDATA
END
GO
QUIT
```

Just as *@OLE*() is used to connect a LINGO model to a spreadsheet and *@ODBC*() is used to connect a LINGO model to a database, the *@TEXT*() statement is available to connect a LINGO model to a simple text file. You can send the value(s) of attribute *X* to a file called "myfile.out" with:

```
DATA:
  @TEXT( 'MYFILE.OUT') = X;
ENDDATA
```

The following will send the value of *X* to the screen, along with an explanatory message:

```
@TEXT() = 'The value of X=',  X;
```

Still one more way that LINGO can be incorporated into an application is by way of a subroutine call. A regular computer program, say in C/C++ or Visual Basic, can make a regular call to the LINGO DLL (Dynamic Link Library). The model is passed as a string variable to the LINGO DLL. See the LINGO manual for more details.

5.8 Summary

In this chapter, we've discussed the concept of sets, defined sets, and demonstrated the power and flexibility of set-based modeling. You should now have a foundation of knowledge in the definition and use of both primitive and derived sets.

5.9 Problems

1. You wish to represent the status of an academic institution during a specific teaching term. The major features to be represented are that instructors teach courses and students are registered for courses. You want to keep track of who is teaching which course, who is registered for each course, and which courses a given student is taking. What sets would you recommend if each course is taught by exactly one instructor?
2. Suppose we take into account the additional complication of team teaching. That is, two or more instructors teach some courses. How would you modify your answer to the previous question?