

Chapter 5

Modeling with

Xpress-Mosel

Overview

In this chapter, you will:

- construct simple models using the Mosel model programming language;
- develop versatile models through the use of subscripted variables, and constants;
- learn about separation of model data from model structure;
- learn about the use of data files and parameters.

Introduction

Over the course of the last few chapters we have seen how each of the interfaces for Xpress-MP can be used to load and solve a simple model, obtaining the optimal solution to the linear problem. For this chapter, however, we begin to explore the Mosel model programming language in greater detail, learning how to construct new models and to interpret the solution produced. At the same time, the desirability of flexible modeling will also be explored, as well as that of separating

model structure from the data which makes up a particular instance of that model.

The Xpress-Mosel model programming language can often be used in many ways to model the same problem and the examples given here will represent just one approach to this. Through these, we will aim to introduce a large part of the language, although foremost in our minds will always be the development of good modeling practice.

Constructing our First Model

The Burglar Problem

Burglar Bill breaks into a house one night with a sack to carry away items of interest to him. He identifies a number of items which have the following weights and estimated values:

	Weight	Value
Camera	2	15
Necklace	20	100
Vase	20	90
Picture	30	60
TV	40	40
Video	30	15
Chest	60	10
Brick	10	1

Bill can only carry items up to a total weight of 102 pounds. Subject to this, his aim is to maximize the estimated value of the items that he takes.

Problem Specification

Formulating the Burglar Problem mathematically is relatively simple. Suppose that we have a decision variable, *camera*, which has the value 1 if Bill takes the camera and 0 otherwise. Suppose also that we have a similar set of variables for the other items. The Burglar Problem may then be expressed as:

Problem 1

Maximize: $15*camera + 100*necklace + 90*vase + 60*picture + 40*tv + 15*video + 10*chest + 1*brick$

Subject to: $2*camera + 20*necklace + 20*vase + 30*picture + 40*tv + 30*video + 60*chest + 10*brick \leq 102$

$camera, necklace, vase, picture, tv, video, chest, brick \in \{0, 1\}$

"The decision variables..."

This problem has a number of parts which should already be familiar to you. The variables *camera*, *necklace* etc. are also often known as *decision variables* as it is the value of these that must be decided on during the solution process. In standard linear programming (LP) problems, the decision variables can take any non-negative real values and it was a problem of this kind which you will have solved in the past three chapters. In the case of the Burglar Problem, however, it makes little sense to take half a video, or even five necklaces if only one is there. Instead this may be described as a (mixed) integer programming (MIP) problem, with the decision variables only allowed to be either 0 or 1. They are, in fact, *binary* variables.

"The constraint" The total weight of all the items taken is given by

$$2*camera + 20*necklace + 20*vase + 30*picture + 40*tv + 30*video + 60*chest + 10*brick$$

The value of this expression is *constrained* by the total weight of items that Bill can carry, or in other words, this must be less than or equal to 102. LP or MIP problems can have any number of constraints such as this, although for the purposes of this model only one is required.

"The objective function..."

The *objective function* is the value of the items taken. The objective of the modeling exercise is to maximize this function subject to the various constraints.

The problem name forms the basis for a number of files that are generated by the Optimizer and Mosel.

Entering the Model into Xpress-Mosel

Models specified using the Mosel model programming language are divided up into a number of *blocks*, the outermost being the `model` block. It is here that the problem is given a name, which we will choose to be `burglar` for the purposes of this exercise. All blocks must be explicitly ended using an accompanying `end-<blockname>` keyword when the statements contained in them are complete.

The second main block that we will use initially is the `declarations` block in which the variables are declared, along with their type. Decision variables are of type `mpvar`. We demonstrate this in Listing 5.1.

Listing 5.1 Declaring decision variables

```
model burglar
  declarations
    camera, necklace, vase, picture: mpvar
    tv, video, chest, brick: mpvar
  end-declarations

end-model
```

model

The `model` block defines the problem name and contains all statements to be considered part of the model. Any text appearing after the `end-model` is treated as if it were a comment and ignored.

declarations

In the `declarations` block are defined variables and their type, sets and scalars.

The decision variables must also be specified as being binary-valued. This is in some sense another constraint on the problem and so it is stated outside of the `declarations` block along with the other constraints. The syntax for this is simply

Similarly, variables can be described as integral using `is_integer`.

```
variable is_binary
```

Finally the constraint and objective function must be added. These can be assigned a name and take the form:

```
name := linear_expression
```

"Continuation lines..."

If the linear expression is particularly long, it may be split over several lines as necessary. There is no character or command which denotes a split line to Mosel, but the line is assumed to continue if it ends in such a way that more input is expected. An example might be if the line ends in a binary operator such as '+', following which another term is expected. This is demonstrated in Listing 5.2 where the full model is given.

Listing 5.2 Adding constraints to the model

```
model burglar
  declarations
    camera, necklace, vase, picture: mpvar
    tv, video, chest, brick: mpvar
  end-declarations

  camera    is_binary
  necklace  is_binary
  vase      is_binary
  picture   is_binary
  tv        is_binary
  video     is_binary
  chest     is_binary
  brick     is_binary

  TotalWeight := 2*camera + 20*necklace + 20*vase +
                 30*picture + 40*tv + 30*video +
                 60*chest + 10*brick <= 102
```

Library module names take a `mm` prefix to denote 'Mosel Module'.

This model is on the Xpress-MP CD as file `burglar1.mos`.

Listing 5.2 Adding constraints to the model

```
TotalValue := 15*camera + 100*necklace + 90*vase +
              60*picture + 40*tv + 15*video +
              10*chest + 1*brick
```

```
end-model
```

The Optimizer Library Module

With the model constructed, it just remains to solve it to find the maximum value of items that Bill can carry away with him. This can also be done from within Mosel. In addition to the standard Mosel syntax, additional functionality can be loaded into it in the form of library modules allowing interaction with external applications. Using the Optimizer library module, `mmxprs`, Mosel can call the Optimizer to solve the problem and return the solutions for display. It does this with the `uses` keyword. The objective function can then be maximized using the `maximize` command. With both modeling and solution components together, this is perhaps most accurately described as a *model program*. A full model program for the problem is given in Listing 5.3, with additions highlighted in bold face.

Listing 5.3 The full burglar model program

```
model burglar1
  uses "mmxprs"

  declarations
    camera, necklace, vase, picture: mpvar
    tv, video, chest, brick: mpvar
  end-declarations

  camera is_binary
  necklace is_binary
  vase is_binary
  picture is_binary
  tv is_binary
  video is_binary
  chest is_binary
  brick is_binary
```

Listing 5.3 The full burglar model program

```
TotalWeight := 2*camera + 20*necklace + 20*vase +
              30*picture + 40*tv + 30*video +
              60*chest + 10*brick <= 102

TotalValue := 15*camera + 100*necklace + 90*vase +
             60*picture + 40*tv + 15*video +
             10*chest + 1*brick

maximize(TotalValue)

writeln("Objective value is ", getobjval)

end-model
```

The final part of this model program involves how we deal with the solution once found. Using the `writeln` command, Mosel can be instructed to output information such as the objective function value, obtainable using `getobjval`.

uses

Allows for the loading of library modules into Mosel.

maximize / minimize

Optimization commands calling Mosel to optimize the objective function declared as their argument.

writeln

Instructs Mosel to send information to standard output. It can take any number of arguments, and each such is returned in the stated order.

getobjval

Returns the objective function value following solution.



Exercise *Input the model program of Listing 5.3 and compile, load and run it in Mosel using your chosen interface. What is the maximum value of the haul that Bill can make? Which items does he take to achieve this?*

Modeling Using Arrays

Array Variables and Indexing

Creating models in the form described above is convenient for small numbers of decision variables, but as the size of the problem increases it quickly becomes very cumbersome to work with. Already in our first model specification it was untidy having to explicitly state that each variable was binary and would be equally so if we were to use Mosel to output the values of the decision variables. It is usual in modeling even small problems like this to make use of *array variables*, or *subscripted variables* as they are sometimes called.

Listing 5.4 shows an altered declarations section for the Burglar Problem, introducing the various arrays that will be used.

Listing 5.4 declaring and entering data into arrays

```
declarations
  Items = 1..8
  WEIGHT: array(Items) of real
  VALUE:  array(Items) of real
  x: array(Items) of mpvar
end-declarations

WEIGHT := [ 2, 20, 20, 30, 40, 30, 60, 10]
VALUE  := [ 15,100, 90, 60, 40, 15, 10, 1]
```

The first thing defined here is an indexing set for the array elements, called `Items`. `Items` is assigned to hold eight integer values, effectively numbering the elements of the arrays. Following this two more arrays are defined, `WEIGHT` and `VALUE` holding respectively the weights and values for the items to be chosen from. They are declared as arrays indexed by `Items` with entries that are of type `real`. The Mosel type `real` corresponds to double precision floats in C. Mosel also supports types `integer` and `string` which have direct analogs in other programming languages. Outside of the `declarations` block these two arrays have their data entered as a comma-separated list.

The final array defined in the `declarations` block is `x` which will act as our decision variable. It is an array, also indexed by `Items`, with entries of type `mpvar`.

Looping and Summation

Using arrays does not just make the variable declaration tidier. By looping through the array of decision variables we can also declare the variables as binary in a single statement. Mosel supports simple looping with the help of the `forall` statement.

Multiple statements may be executed with a `forall` loop by placing them in a `do / end-do` block. For details of this and further details of `forall` structure, see page 165.

forall

Allows looping through an index set. Its structure is:

```
forall(var in set) statement
```

where *var* is a dummy variable, *set* is an indexing set and *statement* is to be executed on each such entry.

Mosel also supports summation notation with the `sum` command, used extensively in constructing linear expressions for constraints and the objective function.

sum

Allows summation over an index set. Its structure is:

```
sum(var in set) expression
```

where *var* is a dummy variable, *set* is an indexing set and *expression* is the generic term in the sum.

Using these to define the decision variables as binary and to set up the constraint and objective function results in the statements in Listing 5.5.

Listing 5.5 Looping and summation in the Burglar Problem

```
forall(i in Items) x(i) is_binary
TotalValue := sum(i in Items) x(i)*VALUE(i)
TotalWeight:= sum(i in Items) x(i)*WEIGHT(i)<=102
```

Comments

With the use of arrays and subscripting, a better model for the Burglar Problem is almost complete. However, as with all programming, it is good practice to ensure that your model is adequately commented to make it easier for both yourself and other to read. It has already been mentioned that any amount of commentary can be added following the `end-model` statement, but what happens when you want to place comments in the body of the model itself? For exactly this purpose, the Mosel model programming language supports two types of comments. Single line comments are introduced with the `!` character. Any text following this character is ignored as comment to the end of the line. For multiline commentary, the pair `(! and !)` must be used.

With comments added, a model program for the Burglar Problem is given in Listing 5.6.

Listing 5.6 Second attempt at the Burglar Problem

This model is on the Xpress-MP CD as file `burglar2.mos`.

```
model burglar2 (! Modeling with arrays, subscripts,
                summations, looping and comments !)
uses "mmxprs"

declarations
  Items = 1..8           ! 8 items
  WEIGHT: array(Items) of real
  VALUE:  array(Items) of real
  x: array(Items) of mpvar
end-declarations

! Item:      1,  2,  3,  4,  5,  6,  7,  8
WEIGHT := [  2, 20, 20, 30, 40, 30, 60, 10]
VALUE  := [ 15,100, 90, 60, 40, 15, 10,  1]
```

Listing 5.6 Second attempt at the Burglar Problem

```
! All x are binary
forall(i in Items) x(i) is_binary

! Objective: maximize the haul
TotalValue := sum(i in Items) x(i)*VALUE(i)

! Constraint: weight restriction
TotalWeight:= sum(i in Items) x(i)*WEIGHT(i)<=102

maximize(TotalValue)

writeln("Objective value is ", getobjval)
forall(i in Items) writeln(" x(",i,") = ",
    getsol(x(i)))
writeln
end-model
```

The additional lines at the bottom of the model instruct Mosel to output the values of the decision variables as well as the objective function value. For this another `forall` loop has been used to go through the variable array returning the solution value for each element using `getsol`.

getsol

Returns optimal values of the decision variables following solution of the problem. Its single argument is the variable whose value should be returned.



Exercise Alter your model program to use array variables, such as in Listing 5.6. Compile, load and run it. Which items are taken by Bill?

Using String Indices

The model program developed above is considerably simpler to use than our first attempt, but interpreting the solution is made more difficult since we have to manually match up the index number for a variable with the item that it represents. It would be far more useful

if all this information could be presented together. Mosel allows for this with the use of string indices rather than numerical ones.

The use of index sets in the Mosel model programming language is perhaps more simple and natural even than using numerical arrays. Defining `Items` as a string set is the only change necessary and this is demonstrated in Listing 5.7.

Listing 5.7 Using string indices in the Burglar Problem

This model is on the Xpress-MP CD as file `burglar3.mos`.

```

model burglar3
  uses "mmxprs"

  declarations
    Items = {"camera", "necklace", "vase",
            "picture", "tv", "video",
            "chest", "brick"}
    WEIGHT: array(Items) of real
    VALUE: array(Items) of real
    x: array(Items) of mpvar
  end-declarations

  ! Item:      ca, ne, va, pi, tv, vi, ch, br
  WEIGHT := [ 2, 20, 20, 30, 40, 30, 60, 10]
  VALUE  := [ 15,100, 90, 60, 40, 15, 10, 1]

  ! All x are binary
  forall(i in Items) x(i) is_binary

  ! Objective: maximize the haul
  TotalValue := sum(i in Items) x(i)*VALUE(i)

  ! Constraint: weight restriction
  TotalWeight:= sum(i in Items) x(i)*WEIGHT(i)<=102

  maximize(TotalValue)

  writeln("Objective value is ", getobjval)
  forall(i in Items) writeln(" x(",i,") = ",
    getsol(x(i)))
  writeln
end-model

```



Exercise Alter your model program to make use of string index sets. Compile, load and run it and compare the output with before.

Versatility in Modeling

Generic and Instantiated Models

The Burglar Problem is just one example of a number of similar modeling problems, known also as knapsack problems. Whilst the above model represents a good first step toward modeling the Burglar Problem, it could still be made easier to generalize should the number of items need to be increased, or even if the problem is changed to a different knapsack problem. With the problem already specified in terms of array variables, the remaining difficulties are largely due to the fact that the data are currently 'hard-wired' into the model — there is no separation between model data and structure. For the current section we shall discuss ways of overcoming these limitations.

Models are generally formulated using symbols to represent the various decision variables, with the relationship between these variables described by a set of equations and inequalities — the constraints. The systematic description of these constraints forms the *generic* model. Combining this with a given set of data produces a particular *numerical* model, the *model instance*, which can then be optimized. Separating off the model structure from the numerical data results in more flexible models which are easier to maintain. For this reason, our aim over the next few pages is to impose such a distinction on the model just created.

Separation also allows you to distribute models as BIM files, protecting intellectual property.

Scalar Declarations

Looking at the model of Listing 5.7, the constraints are already almost in a suitably flexible form. However, the specific information about the maximum weight that Bill can take is still embedded within the body of the model. We can easily get around this by defining a scalar,

`MAXWT`, assigning this the value `102` and then using that throughout the model instead.

Additional quantities such as this may be assigned values from within the `declarations` block at the head of the model file. Any quantity which is assigned a value in this way is assumed to be a constant.

Inputting Data From Text Files

The other change we shall make to the model program at this stage is to separate out the model array data and input it from an external data file. Mosel does this through use of the `initializations` block.

initializations

Enables data to be input from a stated text file. Arrays and constants may be specified in any order and it will be read in at run time.

Suppose that the file `burglar4.dat` in the `Data` subdirectory contains the following information:

Listing 5.8 Structure of an external data file

Note that the array values are no longer comma-separated.

```
! Data file for burglar4.mos
WEIGHT: [ 2  20  20  30  40  30  60  10]
VALUE:  [15 100  90  60  40  15  10  1]
```

Then, placing the arrays `WEIGHT` and `VALUE` in the `initializations` block, Mosel will read in the data to fill the arrays at run time. A full listing for the model program implementing this is given in Listing 5.9, with changes highlighted in bold face.



Enter the data of Listing 5.8 into a file `burglar4.dat` in the `Data` subdirectory and alter your previous model to input data from this file. Compile, load and run to check this has worked smoothly.

Listing 5.9 Inputting data from sources files

This model is on the Xpress-MP CD as file `burglar4.mos`.

```
model burglar4
  uses "mmxprs"
  declarations
    Items = {"camera","necklace","vase","picture",
            "tv","video","chest","brick"}
    WEIGHT: array(Items) of real
    VALUE: array(Items) of real
    MAXWT = 102
    x: array(Items) of mpvar
  end-declarations

  ! Read in data from external file
  initializations from 'Data/burglar4.dat'
  WEIGHT
  VALUE
  end-initializations

  ! All x are binary
  forall(i in Items) x(i) is_binary

  ! Objective: maximize the haul
  TotalValue :=sum(i in Items) x(i)*VALUE(i)

  ! Constraint: weight restriction
  TotalWeight:=sum(i in Items) x(i)*WEIGHT(i)<=MAXWT

  maximize(TotalValue)

  writeln("Objective value is ", getobjval)
  forall(i in Items) writeln(" x(",i,") = ",
    getsol(x(i)))
  writeln
end-model
```

Completing the Burglar Problem

If you have completed the last exercise, the obvious question to ask is 'how much more can we input from the data file?' Certainly the value of the scalar `MAXWT` need not be specified in the model program and can be saved to the data file. The same is actually true of the index set, since the number and names of the items to choose from are really

model data rather than part of the model structure. Suppose, then, that the data file was modified to include the following information:

Listing 5.10 Revised data file for the Burglar Problem

The set is defined as a string array.

```
! Data file for burglar5.mos
Items:["camera" "necklace" "vase" "picture" "tv"
       "video" "chest" "brick"]
MAXWT:102
WEIGHT:[ 2  20  20  30  40  30  60  10]
VALUE:[15 100  90  60  40  15  10  1]
```

If the model program could input all this information at run time, we would finally have separated all the model data from its structure. Listing 5.11 shows the changes to the model program to handle input of this.

Listing 5.11 Completing the Burglar model program

This model is on the Xpress-MP CD as file burglar5.mos.

```
model burglar5
  uses "mmxprs"

  parameters
    WeightFile = 'Data/burglar5.dat'
  end-parameters

  declarations
    Items: set of string
    MAXWT: real
    WEIGHT: array(Items) of real
    VALUE: array(Items) of real
    x: array(Items) of mpvar
  end-declarations

  ! Read in data from external file
  initializations from WeightFile
    Items MAXWT WEIGHT VALUE
  end-initializations

  forall(i in Items) create(x(i))
```

Listing 5.11 Completing the Burglar model program

```
! All x are binary
forall(i in Items) x(i) is_binary

! Objective: maximize the haul
TotalValue :=sum(i in Items) x(i)*VALUE(i)

! Constraint: weight restriction
TotalWeight:=sum(i in Items) x(i)*WEIGHT(i)<=MAXWT

maximize(TotalValue)

writeln("Objective value is ", getobjval)
forall(i in Items) writeln(" x(",i,") = ",
    getsol(x(i)))
writeln
end-model
```

"Understanding Listing 5.11..."

Jumping to the declarations block, the first thing to note is that `Items` and `MAXWT` have to have their type declared if their data is to be read in subsequently. `Items` is a (index) set comprising elements which are strings, and we have defined `MAXWT` to have the same type as the elements of the arrays `VALUE` and `WEIGHT`.

If the model is run with just these changes applied, it will fail at run time, claiming that the model is empty. The problem arises from the fact that our decision variable, `x`, has been declared as indexed by a set of, as yet, undetermined size. Since the indexing set is not input until later, the decision variable array is dynamic and is initialized with no entries. In this case, the array *must* be generated explicitly, which we do here using the `create` command.

create

Explicitly creates a variable that is part of a previously declared dynamic array.



In fact, this is not the only way to get around this problem. A second declarations block could be included immediately following the

initializations block and the `x` variable could be declared here instead. An example of the relevant parts of the model is given in Listing 5.12.

Listing 5.12 Using two declarations sections

```

declarations
  Items: set of string
  MAXWT: real
  WEIGHT: array(Items) of real
  VALUE: array(Items) of real
end-declarations

! Read in data from external file
initializations from WeightFile
  Items MAXWT WEIGHT VALUE
end-initializations

declarations
  x: array(Items) of mpvar
end-declarations

```

The linear nature of the way in which Mosel statements are read and interpreted from a model file means that `Items` is well-defined for all statements in the second set of declarations, since the size and entries in `Items` are known immediately following the `initializations` block. For the current model either of these two methods will work perfectly well, although we will work with the former, keeping the blocks together purely for clarity.



Exercise *Alter your model program to look like Listing 5.11. Compile, load and run it as before to solve the problem.*

Using Parameters with Mosel

The final change in Listing 5.11, which has not yet been discussed, is right at the top of the model program and involves the use of the `parameters` block. Using this, parameters may be defined whose values are then input throughout the model program as they are encountered. The values may be numeric or of string type, so if we had wanted to run the problem with a selection of different maximum weights, `MAXWT` could have been defined here as a parameter rather than a scalar. In this example we show how the data file name can be defined initially and then used in the `initializations` block later to input data to create a model instance.

parameters

This block contains a list of parameter symbols along with their default values to be input into the model program at run time if no alternative values are specified.

We illustrate this point with the introduction of a second example.

The Hiker Problem

Henry decides to go on a hiking holiday carrying all the items that he will need in his rucksack. He has five items from which to choose what he will take and each item that he takes represents a certain saving on his holiday. For example, by taking a tent and sleeping bag, he saves on accommodation costs. The items and their relative savings are as follows:

	Weight	Saving
Tent	60	100
Sbag	28	50
Camera	20	35
Clock	8	10
Food	40	50

However, he can only carry items up to a maximum weight of 100 pounds. What items should Henry take to maximize his saving?

Solving the Hiker Problem

This problem is another example of a knapsack problem and as such the model structure is identical to that which we have been creating in this chapter. To solve the problem, therefore, we only need run the same model program again with a different set of model data. This is illustrated in Listing 5.13, where the saving will be loaded into the array `VALUE`.

Listing 5.13 Data file for the Hiker Problem

```
! hiker.dat - Data file for Hiker Problem
Items: ["Tent" "Sbag" "Camera" "Clock" "Food"]
MAXWT: 100
WEIGHT: [ 60  28  20   8  40]
VALUE:  [100  50  35  10  50]
```



Exercise Enter the data of Listing 5.13 into a file `hiker.dat` in the `Data` subdirectory. The compiled file `burglar5.bim` can be loaded into Mosel and run, passing the `WeightFile='Data/hiker.dat'` parameter as an argument to `run`. Which items should Henry take and what is his maximum saving?

A full listing demonstrating this using Mosel as a Console application is given in Listing 5.14.

Listing 5.14 Solving the Hiker Problem

```
C:\Mosel Files>mosel
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-zzzz
>load burglar5
>run WeightFile='Data/hiker.dat'
Objective value is 160
x(Tent) = 1
x(Sbag) = 1
x(Camera) = 0
x(Clock) = 1
x(Food) = 0

Returned value: 0
>
```

Getting Help



Over the course of this chapter we have developed a relatively simple knapsack problem to introduce some of the basic elements of the Mosel model programming language. Over successive iterations of this process we saw the benefits of separating model structure from its data in terms of versatility in modeling. However a presentation such as this can only really scratch the surface and there are many features of the Mosel language which have not been touched on. In the following chapter a few of these will be briefly discussed to provide a feel for what else is possible, but for full details you should consult the Mosel Reference Manual. This gives a detailed description of all the possibilities in the language as well as describing the functions in some of the standard library modules accompanying the software. The various models described in this chapter can all be found on the installation CD-ROM along with many other examples of using the Mosel language.

Summary

In this chapter we have learnt how to:

- ✓ declare variables, arrays, constants and indexing sets;
- ✓ enter constraints and the objective function;
- ✓ employ the Optimizer library module to solve problems and to use `writeln` to return solution information;
- ✓ use subscripted variables, summation and simple looping constructs;
- ✓ input data from external files, separating model data from the structure.

Chapter 6

Further Mosel Topics

Overview

In this chapter you will:

- learn about using sets and arrays;
- learn how to import and export data using data files and spreadsheets;
- learn about conditional variables and constraints;
- meet the basic programming structures of the Mosel language;
- learn about writing subroutines in your models.

Introduction

In the previous chapter we saw how models can be described in a number of different ways using the Mosel model programming language, going beyond the trivial examples which had been used as a basis for learning the interfaces in Chapters 2–4. Despite the variety of ideas covered there, a number of topics related to the Mosel language are worth a brief discussion. Each topic described here is a self-contained unit and can be read independently of the others if you have a particular interest for your own modeling.

The topics that will be described here are the following:

- constant sets, dynamic sets and set operations;
- multi-dimensional, dynamic and sparse arrays;
- importing data from files and ODBC-enabled products;
- conditional generation of variables and constraints;
- selection and looping constructs;
- writing your own procedures and functions.

Further details of these may be found in the Mosel Reference Manual.

Working With Sets

Sets are collections of objects of the same type, where an ordering is not imposed on their elements. Mosel sets may be defined for any of the elementary types: the basic types (*integer*, *real*, *string* and *boolean*) and the Xpress-MP types (*mpvar* and *linctr*). They can be initialized in three different ways, which we briefly recall.

Constant Sets: Sets whose elements have been declared within a `declarations` section in a model are *constant* sets — their contents cannot subsequently be changed. An example of this would be:

```
declarations
  SnowGear = { 'hat', 'coat', 'scarf', 'gloves', 'boots' }
end-declarations
```

Data importing is covered in detail in “Importing and Exporting Data” on page 148.

File Initialization: Elements of sets may also be stored in data files and imported into a model using an `initializations` block:

```
declarations
  FIB: set of integer
end-declarations

initializations from 'datafile.dat'
  FIB
end-initializations
```

where the file `datafile.dat` contains data such as:

```
FIB: [1 1 2 3 5 8 13 21]
```

Implicit File Initialization: Sets used to index arrays may also be initialized indirectly during initialization of the array. For a model

```
declarations
  REGION: set of string
  DEMAND: array(REGION) of real
end-declarations

initializations from 'transport.dat'
  DEMAND
end-initializations
```

where the file `transport.dat` might contain data such as:

```
DEMAND: [(North) 240 (South) 159 (East) 52 (West) 67]
```

the set `REGION` will be initialized as:

```
{'North', 'South', 'East', 'West'}
```

Dynamic, Fixed and Finalized Sets

Sets which are not constant are considered by Mosel to be *dynamic*, that is, elements may be added to or removed from the set at any point. Once a set has been used to index an array, it becomes *fixed* and elements may no longer be deleted, although further elements can still be added. In many cases, however, the contents of a set do not change once it has been initialized and in such circumstances there is little reason to prefer dynamic sets over constant ones. Rather the opposite is true, for the following reason: Arrays indexed by dynamic sets are themselves created dynamic in Mosel. Since Mosel handles static arrays (those indexed by constant sets) slightly more efficiently than dynamic arrays, it is preferable to employ static arrays in models where possible. For this reason, Mosel provides the `finalize` statement allowing you to turn dynamic sets into constant ones.

In a continuation of the previous example, this might be used as follows:

```
finalize(FIB)

declarations
  x: array(FIB) of mpvar
end-declarations
```

Set Operations

In all examples so far in this manual, sets have been employed purely for the indexing of other modeling objects. However, this need not be the case and we provide details here of a few of the other possibilities available. Some of these are demonstrated in Listing 6.1.

Listing 6.1 Using set operators

```
model Sets
  declarations
    Cits = {"Rome", "Bristol", "London", "Paris",
           "Liverpool"}
    Ports = {"Plymouth", "Bristol", "Glasgow",
            "London", "Calais", "Liverpool"}
    Caps = {"Rome", "London", "Paris", "Madrid"}
  end-declarations

  writeln("Union of all places: ", Cits+Ports+Caps)
  writeln("Intersection of all three: ",
         Cits*Ports*Caps)
  writeln("Cities that are not capitals: ", Cits-Caps)

end-model
```

This example illustrates the use of the following three set operations:

- set union ('+');
- set intersection ('*');
- set difference ('-').

The first and last of these have associated operators '+=' and '-=' which act on sets in much the same way as they do on numbers, modifying a set subject to the elements of another.



Exercise Type in and run the example of Listing 6.1 to find the union, intersection and difference of the sets of places. Alter your example to use the operators '+=' and '-='.

Mosel supports a number of other operators for use with sets. The `in` operator has already been seen in several examples of the `forall` structure, allowing us to loop over all elements of a set. Comparison operators may also be used and include: subset ('<='), superset ('>='), difference ('<>') and equality ('='), returning boolean expressions. Their use is illustrated in Listing 6.2.

Listing 6.2 Using set comparison operators

```
model "Set Comparisons"

  declarations
    RAINBOW = {"red", "yellow", "orange", "green",
              "blue", "indigo", "violet"}
    BRIGHT = {"orange", "yellow"}
    DARK = {"blue", "brown", "black"}
  end-declarations

  writeln("BRIGHT is included in RAINBOW: ",
         BRIGHT<=RAINBOW)
  writeln("BRIGHT is not the same as DARK: ",
         BRIGHT<>DARK)
  writeln("RAINBOW contains DARK: ", RAINBOW>=DARK)

end-model
```



Exercise Enter the model of Listing 6.2, altering it to include the other comparison operators in output statements. What is the output produced?

Working with Arrays

In contrast to sets, arrays are *ordered* collections of objects of the same type and may be defined for any of the elementary types: the basic types (`integer`, `real`, `string` and `boolean`) and the Xpress-MP types (`mpvar` and `linctr`). Arrays may be indexed by a set, examples of which have been seen in the last section, or more simply using the natural numbers without reference to a defined set.

Multi-Dimensional Arrays

Initializing and entering data into arrays in one dimension may be simply achieved in the following way.

```
declarations
  OneDim: array(1..10) of real
end-declarations

OneDim:= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We have already seen many examples of setting up one-dimensional arrays in Chapter 5. Initializing multi-dimensional arrays can be achieved in exactly the same manner, although in this case formatting can be used advantageously to make the result more intuitive. In two dimensions, this could be done as follows:

```
declarations
  TwoDim: array(1..2,1..3) of real
end-declarations

TwoDim:= [11, 12, 13,
          21, 22, 23]
```

which is, of course, syntactically the same as

```
TwoDim:= [11, 12, 13, 21, 22, 23]
```

These statements are interpreted by Mosel by placing the values into the array `TwoDim` row-wise — it is the last subscript which varies most rapidly. Thus, the values in the arrays are the following:

```
TwoDim[1] [1] = 11
TwoDim[1] [2] = 12
TwoDim[1] [3] = 13
TwoDim[2] [1] = 21
TwoDim[2] [2] = 22
TwoDim[2] [3] = 23
```

In higher dimensions, the same principle may be applied. Listing 6.3 describes a small model which enters data into a three-dimensional array and then prints out the array elements along with their values.

Listing 6.3 Initializing a three-dimensional array

```
model ThreeDimEx
  declarations
    ThreeDim: array(1..2,1..3,1..4) of integer
  end-declarations

  ThreeDim:= [111, 112, 113, 114,
             121, 122, 123, 124,
             131, 132, 133, 134,
             211, 212, 213, 214,
             221, 222, 223, 224,
             231, 232, 233, 234]

  forall(i in 1..2, j in 1..3, k in 1..4) do
    writeln("ThreeDim(",i,",",j,",",k,") = ",
           ThreeDim(i,j,k) )
  end-do
end-model
```

See page 165 for details of the 'forall' structure.



Exercise Enter the model of Listing 6.3 and compile, load and run it. Check that the array element indices correspond to the element values.

For details of the possible set types, see “Working With Sets” previously.

Fixed and Dynamic Arrays

Arrays in the Mosel language may have either a fixed size, or be dynamically sized as the model program is run. By default an array is created of fixed size if all its indexing sets are of fixed size, in other words if they are constant, or have had their sizes *finalized*. Fixed arrays have space for all their cells created at the point of declaration with uninitialized cells given the value 0 or '' (the empty string) depending on the array type.

By contrast, if the size of an array is not known at the point of declaration, it is created *dynamic*. This might occur if one of its index sets does not have a fixed size, possibly because it is yet to be read in from an external file. Dynamic arrays are created empty and have cells added as they are assigned values, allowing the array to grow as necessary. An array may further be forced to be dynamic by using the `dynamic` qualifier.



If an array of type `mpvar` is either declared as dynamic, or becomes implicitly so, with the size of at least one of its indexing sets unknown at declaration, the corresponding variables are not created. In such circumstances, the individual elements must all be created by hand using the `create` command. If this is not done, the array will have zero size, no decision variables will exist and the problem will be empty. An example of this was seen in Listing 5.11 in the previous chapter and we shall encounter it again in the section “Conditional Variables and Constraints” on page 159.

Sparsity

Almost all large scale LP and MIP problems have a property known as *sparsity*, that is each variable appears with a nonzero coefficient in a very small fraction of the total set of constraints. Often this property is reflected in the data arrays used in the model, with many zero values. When this happens, it is more convenient to provide just the nonzero values of the data array rather than listing all the values. This is also the easiest way to input data into data arrays with more than two dimensions. An added advantage is that less memory is used.

Suppose that we have a data file, `SparseEx.dat`, which contains the information in Listing 6.4.

Listing 6.4 A sparse data format array example

In this file format, the bracketed terms denote the place in the array where the value is to be stored.

```
! SparseEx.dat - Data for SparseEx.mos
COST: [(Depot1 Customer1) 11
       (Depot2 Customer1) 21
       (Depot3 Customer2) 32
       (Depot2 Customer2) 22
       (Depot1 Customer3) 13]
```

In this example an element of the array `COST` has a nonzero value assigned for sending a product from `Depot1` to `Customer1`, from `Depot2` to `Customer1` and so on. However, since no cost is defined for sending a product from `Depot2` to `Customer3`, the array is considered sparse. In Listing 6.5, the model reads in the data contained in this file and prints a list of all possible `DEPOT-CUSTOMER` pairs with their associated costs. Those elements which had no cost assigned are given the value 0.

Listing 6.5 Inputting data in sparse data format

```
model SparseEx
  declarations
    DEPOT, CUSTOMER: set of string
    COST: array(DEPOT,CUSTOMER) of integer
  end-declarations

  initializations from 'SparseEx.dat'
    COST
  end-initializations

  forall(d in DEPOT, c in CUSTOMER)
    writeln(d," -> ",c," : ",COST(d,c))
  end-model
```



In this example, the array `COST` is dynamically sized since the sizes of neither of its indexing sets were known when it was declared. The only space that is used to hold data corresponds to the number of entries added — space for the extra zero elements is not created. This can be seen by using the command `display COST` at the console.



Exercise Enter the above model and run it in Mosel. By displaying the 'value' of the array `COST`, check that this array is dynamic.

Importing and Exporting Data

There are obvious benefits to be gained from separating the form, or structure, of a model from the particular data that make up a model instance. The Mosel model programming language encourages this modeling principle and incorporates a powerful set of facilities for importing and exporting data.

It is possible to enter data into Mosel model program arrays in four main ways:

- directly in the model program file;
- by use of the `initializations` block;
- by use of ODBC;
- by use of the `readln` command.

The latter three ways also provide corresponding methods for the export of data following solution. In this section we cover each of these and discuss their relative benefits.

Model Data Entry

Perhaps the simplest method of entering data in Mosel arrays involves embedding data directly in the model itself. We have seen a number of examples of this in the previous chapter, but a further example of how data may be entered in this way is provided in Listing 6.6.

In this example a one-dimensional array of size five is created and data is entered: `A(1)` is assigned the value 1, `A(2)` the value 2.5 and so on. Finally the array is displayed to the console using a `writeln` statement.

Listing 6.6 Native data entry into arrays

```
model NativeEx
  declarations
    A: array(1..5) of real
  end-declarations

  A:=[1, 2.5, -6.1, 10, 77]

  writeln("A is: ",A)
end-model
```

Whilst quick and easy to use, the downside to this method is that it does not in any way separate model structure from its data. Its use is therefore perhaps limited to the creation of small test examples or the development of prototype models.

Data Transfer Using the `initializations` Block

The `initializations` block may be used to initialize basic objects such as scalars, arrays or sets from external data files and can be used to export solution data later in the model program.

Importing Data from ASCII Files: An initialization data file must contain one or more records of the form:

```
label: value
```

where `label` is a text string and `value` is either a constant, or a collection of values separated by spaces and enclosed in square brackets. Collections of values are used to initialize sets or arrays. During data input, each object to be initialized is associated to a `label` in the external file. This is typically the same label as the object name, but may be different if the modifier `as` is used. When an `initializations` block is executed, the given file is opened and the requested labels are searched for in this file to initialize the corresponding objects.

No particular formatting is required in the file: spaces, tabs and line breaks are all normal separators. Moreover, single line comments are

also supported within the file. An example of such a file is given in Listing 6.7.

Listing 6.7 ASCII file format to use with initializations

```
! InitEx.dat: Using the initialization block
Item: 25
albatross: [12.9 76 1.55 0.99]
A1: [23 15 43 29 90 180]
```

In this file one integer (`Item`) is defined along with two arrays (`A1` and `albatross`). Listing 6.8 provides an example model file which reads these values into memory.

Listing 6.8 Model file to demonstrate input from ASCII files

```
model InitEx
  declarations
    Item: integer
    A1: array(1..6) of integer
    A2: array(1..4) of real
  end-declarations

  initializations from 'InitEx.dat'
    Item A1
    A2 as "albatross"
  end-initializations

  writeln("Item is: ",Item)
  writeln("A1 is: ",A1)
  writeln("A2 is: ",A2)
end-model
```

Several things are worth noting in this example. Firstly, the name of the data file should appear on the same line as the beginning of the `initializations` block, quoted and following the word `from`. Since the `initializations` block can also be used for data export,

the `from` modifier specifies the sense of the data transport. In this case we are obtaining information from an external source.

Second, since items in the `initializations` block are separated by spaces, several of these may be placed on the same line. In the example, we have done this with `Item` and `A1`. It should also be noted that objects here need not be in the same order as the data in the accompanying data file, although it is often sensible to maintain the order, if only for the sake of clarity.

Finally, the array data associated with the label `albatross` in the data file is to be read into the array `A2`. This is achieved in the model file by use of the `as` modifier. Since the label in the data file must be quoted in the model file when this construct is used, labels in the model file may consist of more than one word if necessary.



Exercise *Type in the example above, or create your own to read in and display data. Your example should demonstrate all of the possibilities described above.*

Exporting Data to ASCII Files: An equivalent method may also be used to export data from Mosel to external data files following solution of the problem, with the format for the `initializations` block being:

```
initializations to filename
  identifier [as label]
end-initializations
```

When this form is executed, the values of all provided labels in the file are updated with the current value of the corresponding identifier. If a label cannot be found, a new record is appended to the end of the file and the file is created if it does not exist.



Exercise *Alter your model from the exercise above to export the data to a separate file using this format. Consult the new file with a text editor or similar to see how data has been entered.*

Data Transfer Using ODBC

Creating and maintaining data in text files is quite a simple process, but for many people a considerably more efficient and useful format is provided by spreadsheets and databases. A facility exists in the Mosel language whereby data may be imported from, and exported to, ODBC-enabled spreadsheet and database programs using the structured query language, SQL. To do so requires the use of the library module `mmodbc`, in addition to an extra authorization in your Xpress-MP license.

If you are going to work through the examples here, you will need access to Microsoft Excel.

Setting Up ODBC: Suppose that in a spreadsheet called `Myss.xls` the following data has been entered into the stated cells and the

	A	B	C
1			
2		First	Second
3		6.2	1.3
4		-1.0	16.2
5		2.0	-17.9

range `B2:C5` has been called `MyRange`. When data is imported into Xpress-MP from a spreadsheet, the first row in the marked range is always assumed to contain column headings and is consequently ignored. In this example, data from `MyRange` would fill an array of size `3*2` and in the following, ODBC will be used to extract this data into an array `A(3,2)`.

A spreadsheet range must have a top row with column titles.

Using ODBC

In Windows' Control Panel, select *32-bit ODBC* and set up a User Data Source called `MyExcel` by clicking *Add*, selecting *Microsoft Excel Driver (*.xls)* and filling in the ODBC Microsoft Excel Setup dialog. Click *Options >>* and clear the *Read Only* check box.



Exercise Set up a user data source for Excel, as described in the box 'Using ODBC'.

Importing Data From ODBC-Enabled Products: In Listing 6.9 we demonstrate how data may be read into Mosel from an Excel spreadsheet such as this. Notable here is that SQL syntax must be used to obtain the required data from the spreadsheet and the statement 'SELECT * FROM MyRange' says, quite simply, that everything in the range MyRange is to be returned. This is then placed into the array A as required.

Listing 6.9 Reading in data from an Excel spreadsheet

```
model ODBCImpEx
  uses "mmodbc"

  declarations
    A: array(1..3,1..2) of real
    CSTR: string
  end-declarations

  CSTR:= 'DSN=MyExcel; DBQ=Myss.xls'

  setparam("SQLndxcol",false)
  SQLconnect(CSTR)
  SQLexecute("SELECT * FROM MyRange", [A])
  SQLdisconnect

  forall(i in 1..3)do
    writeln("Row(",i,"): ",A(i,1)," ",A(i,2))
  end-do
end-model
```



Exercise Construct a model (or alter one of your previous ones) which obtains its data from Excel in this way.

Exporting Data to ODBC-Enabled Products: Exporting data back to spreadsheets can be achieved in much the same manner. An example of this is provided in Listing 6.10, where a new sheet is created, *New*, into which data from the array A is entered.

Listing 6.10 Writing data to an Excel spreadsheet

```
model ODBCExpEx
  uses "mmodbc"

  declarations
    A: array(1..3,1..2) of real
  end-declarations

  forall(i in 1..3,j in 1..2) A(i,j) := i*j

  setparam("SQLndxcoll",false)
  SQLconnect('DSN=MyExcel; DBQ=Myss.xls')
  SQLexecute("CREATE TABLE New(Col1 integer, Col2
    integer)")
  SQLexecute("INSERT INTO New(Col1,Col2) VALUES
    (?,?)",A)
  SQLdisconnect

end-model
```

Since SQL is used for any communication between Mosel and ODBC-enabled applications, far more complicated statements than this are possible. You should consult any standard text on SQL for details of the possibilities.



Exercise *Alter your previous model to output data back into Excel following solution of a problem, or alter one of the previous examples to do this. If you encounter problems, consult the following section.*



Opening and Using Microsoft Excel Tables: A number of points should be considered when writing data to Microsoft Excel from Mosel. Since Excel is a spreadsheet application and ODBC was primarily designed for databases, special rules have to be followed to read and write Excel data using ODBC:

- named ranges must be used in an Excel worksheet to refer to tables of data;
- column names must be used as field names;

- the data type of each field should be defined using a row of specimen data below the column headings.

When using ODBC with Excel, it is important that the top row of each range contains the column headings — otherwise errors will occur and data will not be transferred correctly to and from the worksheet. A row of specimen data is also required in the row below the column headings to identify the data type of each column. This specimen row must also be included in the range.

Users should also be aware that when writing to database tables specified by a named range in Excel, the range will increase in size if new data is added. Now suppose that we wish to write further data over the top of data that has already been written to a range using ODBC. Within Excel it is not sufficient to delete the previous data by selecting it and hitting the Delete key. If this is done, further data will be added after a blank rectangle where the deleted data used to reside. Instead, it is important to use *Edit, Delete, Shift cells up* within Excel, which will eliminate all traces of the previous data, and the enlarged range.

Microsoft Excel tables can be created and opened by only one user at a time. However, the 'Read Only' option available in the Excel driver options allows multiple users to read from the same .xls files.



Sizing Arrays for Spreadsheet Data: Since Mosel evaluates objects that it encounters in the order in which they are encountered, the sizes of tables may be adjusted dynamically. In practice, this allows for the possibility that, having written a model, the size of the region in the spreadsheet may actually change if additional data is entered. Since this is a particularly useful trick, we will briefly describe its usage here.

Suppose that we are continuing to work with the spreadsheet `Myss.xls` and we are concerned that the size of `MyRange` may change as more data are added. This can be dealt with by constructing an additional region of the spreadsheet, which we will name `Sizes`. Into this we will put the numbers that characterize the problem as follows:

Reducing the number of rows by 1 allows for the row containing just the column names.

Number of Rows	Number of Columns
----------------	-------------------

=ROWS (MyRange) - 1	=COLUMNS (MyRange)
---------------------	--------------------

Naming the range formed by the two cells in the first column as `NRows` and the range formed by the two cells in the second column as `NCols`, the commands in Listing 6.11 may then form the introductory part of a model.

Listing 6.11 Dynamic table sizing with spreadsheets

```

model SizingEx
  uses "mmodbc"

  declarations
    NRows, NCols: integer
  end-declarations

  SQLconnect ('DSN=MyExcel;DBQ=Myss.xls')
  NRows:=SQLreadinteger("select NRows from Sizes")
  NCols:=SQLreadinteger("select NCols from Sizes")

  declarations
    A: array(1..NRows,1..NCols) of real
  end-declarations

  SQLexecute("select * from MyRange", [A])
  SQLdisconnect

  forall(i in 1..NRows,j in 1..NCols) do
    writeln("A(",i,",",j,") = ", A(i,j))
  end-do
end-model

```



Exercise *Alter your previous model to allow for dynamic resizing of the tables. Now change the number of rows with data to be read in and run the new model in Mosel. Check that the extra data is imported.*

Data Transfer Using `readln` and `writeln` Commands

A considerably more general method for reading data from ASCII files is provided by the `readln` and `read` commands. These commands

assign the data read in from the active input stream to given symbols, or attempt to match a given expression with what is read in. The command `readln` expects all symbols to be recognized to be contained in a single line, whereas `read` allows this to flow over multiple lines.

Use of these commands is perhaps best illustrated by way of an example. Suppose we have an input data file containing information such as that described in Listing 6.12.

Listing 6.12 File format for use with `readln`

```
! File ReadlnEx.dat
read_format( 1 and 1)= 12.5
read_format( 2 and 3)= 5.6
read_format(10 and 9)= -7.1
```

This file, `ReadlnEx.dat`, is then read in by the model program which is described in Listing 6.13. The model begins in the usual way by declaring variables necessary for the problem. The data in the file are to be read into the array `A`, which is sparse and to be sized dynamically.

Listing 6.13 Reading in data with `readln`

```
model ReadlnEx
  declarations
    A: array(range,range) of real
    i, j: integer
  end-declarations

  fopen("ReadlnEx.dat",F_INPUT)
  readln("!")
  repeat
    readln("read_format(",i,"and",j,")=",A(i,j))
  until getparam("nbread") < 6
  fclose(F_INPUT)

  writeln("A is: ",A)
end-model
```

Note here that the dynamic array `A` is indexed by a dynamic set, `range`.

See page 167 for details of the 'repeat' structure.

Using the `fopen` command, the data file is opened and assigned to the active input stream, `F_INPUT`. We then read a line containing the `!` character, since the first line of our file will contain a comment. Following this, the repeat loop reads in as many lines as it can in the format described in Listing 6.12, assigning the value after the `=` sign to an array element indexed by the numerical values spanning the word 'and'. Finally the input stream is closed and the array is printed to the console.

During a run of the program, the commands `read` and `readln` set a control parameter, `nbread`, to the number of items actually recognized in a line. By consulting this, it becomes evident when information has been read in which does not match a given string. In our example, after the first line we might expect six items per line to be recognized by the parser. This can be used to end the repeat loop when there is no further data to be read in.



Exercise Create your own model which uses `readln` and `read` to input data from an external source file, displaying the data that it has read.

Exporting Data With `write` and `writeln`: Array and solution data may be written to file in much the same way using the `writeln` and `write` commands. By opening a file and assigning it the active output stream, `F_OUTPUT`, any subsequent `write` or `writeln` commands will direct output to the file rather than to the console. An example of this is given in Listing 6.14 where the simple problem of Chapters 2 – 4 is again solved and solution values exported to a data file.

It should be noted that when information is exported in this manner, any information currently in the file being written to is lost and the file is completely overwritten with the new data. When this is undesirable, files may instead be opened for appending using:

```
fopen (Filename , F_APPEND)
```



Exercise Enter the model of Listing 6.14 and run it, writing data to the file `WritelnEx.dat`. Now alter the model to append data and run the program again. Consult the output file with a text editor.

Listing 6.14 Outputting solution data with writeln

```
model WritelnEx
  uses "mmxprs"

  declarations
    a,b: mpvar
  end-declarations

  first:= 3*a + 2*b <= 400
  second:= a + 3*b <= 200
  profit:= a + 2*b

  maximize (profit)

  fopen("WritelnEx.dat",F_OUTPUT)
  writeln("Profit = ",getobjval)
  writeln("a=",getsol(a),": b=",getsol(b))
  fclose(F_OUTPUT)
end-model
```

Conditional Variables and Constraints

Conditional Bounds

Suppose that we wish to apply an upper bound to some, but not all members of a set of N variables, x_j . The upper bound that will be applied varies for each variable and these are given by the set U_j . However, they should only be applied if the entry in the data table $COND_j$ is greater than some other amount, say 20. If the bound *did not* depend on the value of $COND_j$, as has been usual up to now, then this might have been expressed using:

```
forall(i in 1..N) x(i) <= U(i)
```

For the conditional bound, however, this must be altered slightly:

```
forall(i in 1..N | COND(i) > 20) x(i) <= U(i)
```

The vertical bar (|) character, followed by a logical expression denotes a conditional operator and should be read as “to be done whenever the following expression is true”, or “such that”.

Thus the line in the second of these examples reads “for $i=1$ to N , the variable $x(i)$ must be less than or equal to $U(i)$ whenever $COND(i)$ is greater than 20”.

Conditional Variables

The existence of variables can also be made conditional by declaring a dynamic array of variables and then creating only those which satisfy a certain condition. An example of this is given in Listing 6.15. The only variables which will actually be defined here are $x(1)$, $x(2)$, $x(3)$, $x(6)$, $x(8)$ and $x(10)$. By constructing a small model and outputting it to the console, this is evident.

We have already seen an example of this in Chapter 5, where the indexing set was also input from an external file. For details, see Listing 5.11 on page 132.

Listing 6.15 Conditional generation of variables

```
model ConditionalEx
  declarations
    Elements = 1..10
    COND: array(Elements) of integer
    x: dynamic array(Elements) of mpvar
  end-declarations

  COND:= [1, 2, 3, 0, -1, 6, -7, 8, -9, 10]
```

Listing 6.15 Conditional generation of variables

```
forall(i in Elements | (COND(i) = i)) create(x(i))

! build a little model to show what's there
obj:= sum(i in Elements) x(i)
c:= sum(i in Elements) i*x(i) >= 10
exportprob(0,"",obj)
end-model
```



Exercise Create and run the model of Listing 6.15. Check that only six variables exist by checking the LP format matrix of the constructed problem.



Use of `exportprob` in this way provides a nice way of seeing directly the problem that has just been created. Without specifying a file name for output, the matrix is displayed on screen.

Basic Programming Structures

The Mosel model programming language provides all the possibilities of a high-level programming language in addition to commands and procedures for model specification. This provides powerful additions to the modeling environment, some of which we shall now discuss.

if Statements

The general form of the `if` statement is:

```
if expression_1
then commands_1
[ elif expression_2
then commands_2 ]
[ else commands_3 ]
end-if
```

The selection is executed by evaluating the boolean *expression_1*. If this is `true`, then *commands_1* are executed and control passes to after the `end-if` statement. If optional `elif` sections are included, then the boolean *expression_2* is evaluated and if this is `true`, then *commands_2* are executed, before control passes to after the `end-if` statement. If none of the expressions prefixed with either `if` or `elif` evaluate to `true`, then any *commands_3* following an optional `else` statement are executed. The program continues from after the `end-if` statement. An example of this is given in Listing 6.16.

Listing 6.16 Using `if` statements in modeling

The `parameters` block lists quantities which may be changed at run time, along with their default values.

The command sequence `\n` is interpreted as a newline character only when enclosed in double quotes.

```

model IfEx
  parameters
    DEBUG=0
    File='InitEx.dat'
  end-parameters

  declarations
    A: array(1..6) of integer
    Item: integer
  end-declarations

  initializations from File
    Item A as "A1"
  end-initializations

  if(DEBUG=1) then
    writeln("Item is ",Item,"\nA is ",A)
  else
    writeln("Data read in from ",File,"...")
  end-if
end-model

```

In this example an integer `DEBUG` is used to control output during a run of a model. For normal use, `DEBUG` is set to 0, so the only statement output by Mosel is to tell us when and from where the data have been read in. However, since `DEBUG` is a parameter, its value can be changed at run time and if it is set to 1, the data read in will also be displayed to confirm that everything is set correctly. Listing 6.17 shows an example session with Mosel in which this model program is

run first as normal and then subsequently with the `DEBUG` parameter set to 1.

Listing 6.17 Using the model debugging feature

```
C:\Mosel Files>mosel
** Mosel **
(c) Copyright Dash Associates 1998-zzzz
>cload IfEx
Compiling `IfEx'...
>run
Data read in from InitEx.dat...
Returned value: 0
>run DEBUG=1
Item is 25
A is [23,15,43,29,90,180]
Returned value: 0
>
```

Another example like this can be found in Listing 6.22 on page 170.



Using this facility provides a particularly convenient way of debugging long models. It is a common trick to use the `writeln` command during debugging to output values of variables when trying to detect the cause of an error. If any statements added for this purpose are enclosed in an `if` statement of this form, then they can be left in the model when debugging has been completed, rather than having to identify and remove the extra statements. If further details are to be subsequently added to the model, then additional debugging may be required. The next time around, your debugging statements will already be in place to help.



Exercise Enter and run the example of Listing 6.16 both as normal and with the `DEBUG` feature enabled. Now alter the model program using an `elif` statement to add an extra `DEBUG` level of 2 which additionally prints out the file name from which the data is collected.

case Statements

The general form of the `case` statement is:

```
case Expression_0 of
  Expression_1 : Statement_1
or
  Expression_1 : do Statement_list_1 end-do
[
  Expression_2 : Statement_2
or
  Expression_2 : do Statement_list_2 end-do
... ]
[ else Statement_list_3 ]
end-case
```

The selection is executed by evaluating the boolean *Expression_0* and sequentially comparing it with each *Expression_i* until a match is found. At this point either *Statement_i* or *Statement_list_i* is executed (depending on the form of the construction) and control passes to after the `end-case` statement. If none of the expressions match and the `else` statement is present, then the statement(s) *Statement_list_3* are executed and control passes to after the `end-case`. A simple example demonstrating this is given in Listing 6.18.

In this example the entries in an array *A* are searched through and categorized according to a (somewhat bizarre) rule: it is interesting to know whether they are either 0, in the range 1 to 3 inclusive, or else either 8 or 10. Anything falling outside of this is declared as 'not interesting' to us. Each element in *A* is categorized and a statement about its value is printed to the screen.



Exercise Construct your own model program which makes use of the `case` construction. Alternatively, adapt your previous program to use the `case` construction to handle multiple debugging levels in code.

Listing 6.18 A simple case example

```
model CaseEx
  declarations
    A: array(1..10) of integer
  end-declarations

  A:=[1,2,3,0,-1,1,3,8,2,10]

  forall(i in 1..10) do
    case A(i) of
      0:   writeln('A(',i,',') is 0')
      1..3: writeln('A(',i,',') is between 1 and 3')
      8,10: writeln('A(',i,',') is either 8 or 10')
      else writeln('A(',i,',') is not interesting')
    end-case
  end-do
end-model
```

forall Loops

The general form of the `forall` statement is:

```
forall (Iterator_list) Statement
or
forall (Iterator_list) do Statement_list end-do
```

Here the *Statement* or *Statement_list* is repeatedly executed for each possible index tuple generated by the *Iterator_list*. An example of this is provided in Listing 6.19.

In this example, Mosel fills the array `F` with the first 20 numbers in the Fibonacci sequence, printing them to the console as it goes. In this sequence the first two numbers are 1 and subsequent numbers are generated as the sum of the two preceding. By looping through the indexing set `Elms`, a simple `if` statement determines if we are initializing the first two terms and, if not, applies the algorithm to determine the next term in the sequence.

Listing 6.19 A simple forall example

```
model ForallEx
  declarations
    Elms=1..20
    F: array(Elms) of integer
  end-declarations

  forall(i in Elms) do
    if(i=1 or i=2) then
      F(i):=1
    else F(i):= F(i-1) + F(i-2)
    end-if
    writeln("F(",i,")\t= ",F(i))
  end-do
end-model
```

The command sequence `\t` is interpreted as a tab character only when enclosed in double quotes.

Other examples that we have seen involving the `forall` loop include setting decision variables as binary in Listing 5.6 and explicitly creating variables in Listing 5.11, both in the previous chapter.



Exercise Create your own model program making use of the `forall` loop, or enter and run the example given here.

while Loops

The general form of the `while` statement is:

```
while (Expression) Statement
or
while (Expression) do Statement_list end-do
```

With this construction, the boolean *Expression* is evaluated and the *Statement* or *Statement_list* is executed as long as *Expression* is `true`. If *Expression* evaluates to `false`, the `while` statement is completely skipped. An example is given in Listing 6.20.

The `while` statement is used here to construct a ‘times table’, printing the product of the row and column numbers. At the end of each row

Listing 6.20 A simple `while` example

```
model WhileEx
  declarations
    i,j: integer
  end-declarations

  i:=1
  j:=1

  while(i <= 10) do
    while(j <= 10) do
      write(i*j)
      if(j=10) then writeln
      else write("\t")
      end-if
      j+=1
    end-do
    j:=1
    i+=1
  end-do
end-model
```

a newline character is printed, whilst between numbers in a row the tab character is used.



Exercise Either enter the example in the listing or create your own example making use of the `while` statement. Compile, load and run it to check output.

repeat Loops

The final looping structure is the `repeat` loop, which has the following general form:

```
repeat Statement_1
[ Statement_2...]
until Expression
```

Here *Statement_1* (and any further statements) are repeatedly executed until the boolean *Expression* evaluates to *false*. By contrast with the `while` loop, statements in a `repeat` loop are guaranteed to be executed at least once, since the execution takes place before the *Expression* is evaluated. An example is given in Listing 6.21.

Listing 6.21 A simple repeat example

```
model RepeatEx
  declarations
    i: integer
    number: integer
  end-declarations

  i:=1
  writeln("Input a positive integer:")
  readln(number)

  repeat i+=1
  until ((number mod i =0) or i>sqrt(number))

  if(i>sqrt(number)) then writeln(number," is prime!")
  else writeln(number," is divisible by ",i)
  end-if
end-model
```

This example provides a short program testing numbers to see if they are prime. When run, the user is prompted to enter a number which is then repeatedly tested until either a divisor is found or it is found to be prime.



Exercise Enter the example of Listing 6.21 and run it. Now alter it to test numbers input to make sure they are integral and bigger than zero. You may need to consider separately how to cope with the input of 1.

Procedures and Functions

It is possible to group together sets of statements and declarations in the form of subroutines which can be called several times during the execution of a model. Mosel supports two kinds of subroutines: procedures and functions. Both of these can take parameters, define local data and call themselves recursively.

Procedures

Procedures consist of a collection of statements which should be run together in a number of places. On execution the statements in the body are run and no value is returned.

procedure

The procedure block takes the form:

```
procedure proc_name [ (param_list) ]  
  proc_body  
end-procedure
```

where *proc_name* is the procedure's name and *param_list* its list of formal parameters, separated by commas. When the procedure is called, statements in *proc_body* are executed.

An example of a procedure is given in Listing 6.22, printing a banner at the beginning of a model run. For models using parameters, an example such as this can provide a useful way of classifying output, particularly if used for information sent to file rather than to the screen. Listing 6.23 shows the use of this program from Console Xpress.



Create a simple procedure outputting a banner at the beginning of a model run. If your model makes use of parameter or other settings read in from external files, you could have these output as well.

Listing 6.22 A simple procedure

Procedures may be placed together at the end of the model as long as they are declared before use using the forward keyword. See the following sections for details.

```
model ProcEx
  parameters
    DEBUG=0
    File='InitEx.dat'
  end-parameters

  procedure banner(DEBUG:integer,File:string)
    writeln("This is the ProcEx model, release 1")
    if(DEBUG = 1) then
      writeln("\tDebugging on...")
    end-if
    if(File <> 'InitEx.dat') then
      writeln("\tInput file ",File," in use...")
    end-if
    writeln
  end-procedure

  banner(DEBUG,File)
end-model
```

Listing 6.23 Running the procedure with parameter input

```
C:\Mosel Files>mosel
** Mosel **
(c) Copyright Dash Associates 1998-zzzz
>cloud ProcEx
Compiling `ProcEx'...
>run
This is the ProcEx model, release 1

Returned value: 0
>run 'DEBUG=1,File=data.dat'
This is the ProcEx model, release 1
      Debugging on...
      Input file data.dat in use...

Returned value: 0
>
```

Extra debugging information is also made available by using the `g` flag with `compile` or `XPRMcompmod`.

Functions

A function is a collection of statements which should be run together in one or more places, and returns a value.

function

The function block takes the form:

```
function func_name [ (param_list) ]: type  
    func_body  
end-function
```

where *func_name* is the function's name, *param_list* its list of formal parameters, separated by commas and *type* is the basic type of the return value. When the procedure is called, statements in *func_body* are executed and a value returned.

Listing 6.24 provides an example of using three functions to find *perfect numbers*. A perfect number is one for which the sum of its divisors is equal to itself. The first function, *isPrime*, checks the primality or otherwise of a number, whilst the second calculates positive integral powers of a number. Finally, the third calls these to generate *Mersenne Primes*, from which it calculates perfect numbers using Euclid's formula.

Notable in this example is the keyword `returned` which must be present in any function description. Its value is sent back to the caller when the function exits, so it should be set to a value of basic type *type*, as declared at the top of the function. In our example, integer values are assigned to `returned` either by each possibility from an `if` statement, or from the power calculation.



Exercise Enter the example of Listing 6.24 and run it to calculate the first four or five perfect numbers. Adapt the code to make the functions more general, rejecting invalid arguments.

Listing 6.24 Calculating perfect numbers with functions

Forward declaration of functions is also permitted. See the following sections for details.

```
model PerfectEx
  function isPrime(number: integer): boolean
    i := 1
    repeat i+= 1
    until ((number mod i = 0) or i > sqrt(number))
    if(i > sqrt(number)) then returned := true
    else returned := false
    end-if
  end-function

  function power(a,b: integer): integer
    pow := 1
    while(b > 0) do
      pow := pow*a
      b-=1
    end-do
    returned := pow
  end-function

  function Perfect(n: integer): integer
    Mn := power(2,n)-1
    if(isPrime(Mn)) then
      returned := power(2,n-1)*Mn
    else returned := 0
    end-if
  end-function

  i := 1; k := 1
  while(k<5) do
    i+=1
    if(Perfect(i) > 1) then
      write(Perfect(i), " = ")
      forall(j in 0..i-1) write(power(2,j), "+")
      write(power(2,i)-1)
      forall(j in 1..i-2)
        write("+", (power(2,i)-1)*power(2,j))
      writeln; k+=1
    end-if
  end-do
end-model
```

Recursion

Both functions and procedures may be used recursively, either calling themselves directly, or indirectly. This can be particularly useful, for which reason we provide an example. In Listing 6.25 the function `hcf` is called recursively to determine the highest common factor of two numbers.

Listing 6.25 Recursion of functions

```
model HcfEx
  function hcf(a,b: integer): integer
    if(a=b) then
      returned:=a
    elif(a>b) then
      returned:=hcf(b,a-b)
    else
      returned:=hcf(a,b-a)
    end-if
  end-function

  declarations
    A,B: integer
  end-declarations

  write("Enter two integer numbers:\n A: ")
  readln(A)
  write(" B: ")
  readln(B)

  writeln("Highest common factor: ",hcf(A,B))
end-model
```



Exercise *Type in the example of Listing 6.25 and experiment with it on a few examples.*

Forward Declaration

We have already hinted that Mosel allows not just for recursion, but also *cross recursion*, where two subroutines alternately call each other. However, since all functions and procedures must be declared before they can be called, the subroutine defined first will of necessity call another which has not been defined. The solution is to use the `forward` keyword to *declare* one or both of the subroutines before their commands are *defined*.

"Declaration v. definition..."

An important distinction should be made at this point between the various vocabulary used. The *declaration* of a subroutine states its name, the parameters (type and name) and, in the case of a function, the type of the return value. The *definition* of the subroutine that will follow later in the model program contains the body of the subroutine, that is, the commands that will be executed when the subroutine is called.

This difference is perhaps best illustrated by way of an example. In Listing 6.26 we implement a quick sort algorithm for sorting a randomly-generated array of numbers into ascending order. The procedure `start` that starts the sorting algorithm is defined at the very end of the program, so it needs to be declared at the beginning before it is called.

Listing 6.26 Forward declaration of subroutines

```

model "Quick Sort"
  parameters
    LIM=50
  end-parameters

  forward procedure start(L:array(range) of integer)

  declarations
    T: array(1..LIM) of integer
  end-declarations

  forall(i in 1..LIM) T(i):=round(.5+random*LIM)
  writeln(T)
  start(T)
  writeln(T)

```

Listing 6.26 Forward declaration of subroutines

```
! swap the positions of two numbers in an array
procedure swap(L:array(range) of integer,
  i,j:integer)
  k:=L(i)
  L(i):=L(j)
  L(j):=k
end-procedure

! main sorting routine
procedure qsort(L:array(range) of integer,
  s,e:integer)
! Determine partitioning value
V:=L((s+e) div 2)
i:=s; j:=e
repeat      ! Partition into two subarrays
  while(L(i)<V) i+=1
  while(L(j)>V) j--1
  if(i<j) then
    swap(L,i,j)
    i+=1; j--1
  end-if
until i>=j

      ! Recursively sort the two subarrays:
if(j<e and s<j) then
  qsort(L,s,j)
end-if

if(i>s and i<e) then
  qsort(L,i,e)
end-if
end-procedure

! start of the sorting process
procedure start(L:array(r:range) of integer)
  qsort(L,getfirst(r),getlast(r))
end-procedure

end-model
```

The idea of the quick sort algorithm is to partition the array that is to be sorted into two parts. One of these contains all values smaller than the partitioning value and the other all the values that are larger than this value. The partitioning is then applied to the two subarrays recursively until all the values are sorted.



Exercise Type in the model of Listing 6.26, or adapt your previous examples to place all subroutine definitions at the end of the program.



Use of the `forward` keyword is particularly useful in providing structure to your model files and making them easier to follow. By placing all detail of a model into subroutines and providing the descriptions of these at the end, the main flow of the program is evident.



Having reached this point, you have encountered enough features of the Mosel model programming language to enable you to create your own, more sophisticated models and solve them using the interface of your choice. As you develop these, you may need to use further features of the Mosel language, all of which may be found in the Mosel Reference Manual. The following (and final) chapter of this introduction provides a glossary of some of the terms that have been used.

Summary

In this chapter we have learnt how to:

- ✓ initialize fixed-size and dynamic array in several dimensions;
- ✓ import and export model data using external sources and applications;
- ✓ create conditional expressions and variables;
- ✓ create selection and looping structures;
- ✓ write functions and procedures.